Universidad de Sevilla

Departamento de Electrónica y Electromagnetismo

Modular and scalable implementation of AER neuromorphic systems

Memoria presentada por:

Carlos Zamarreño Ramos

para optar al grado de Doctor por la Universidad de Sevilla

Sevilla, 2011

Modular and scalable implementation of AER neuromorphic systems

Memoria presentada por

Carlos Zamarreño Ramos

para optar al grado de Doctor por la Universidad de Sevilla

Sevilla, 2011

Directores:

Dra. Teresa Serrano Gotarredona

Dr. Bernabé Linares Barranco

Dr. Alejandro Linares Barranco

Tutor por la Universidad de Sevilla:

Dr. Antonio José Acosta Jiménez

Departamento de Electrónica y Electromagnetismo
Universidad de Sevilla

Agradecimientos

En primer lugar quiero agradecer a mis directores de tesis Teresa Serrano Gotarredona, Bernabé Linares Barranco y Alejandro Linares Barranco por su inestimable ayuda durante estos años de duro trabajo. Siempre han estado dispuestos a ayudarme en los numerosos problemas que han ido surgiendo en el camino y me han guiado durante el mismo hasta llegar a escribir estas líneas. Siempre estaré en deuda por todas las cosas aprendidas estos años que me han introducido en el difícil campo de la microelectrónica.

Las palabras no son suficientes para mostrarle la gratitud merecida a mis padres. Ellos me han apoyado en los momentos más difíciles impidiéndome tirar la toalla y haciéndome ver las cosas en perspectiva. Sin ellos, esta tesis no hubiera sido posible. Es también de justicia agradecer a mis amigos por haberme acompañado en este camino y hacer que la vida merezca la pena.

Sería injusto no nombrar en este agradecimiento a mis compañeros del Instituto de Microelectrónica de Sevilla. Gracias por las interminables discusiones técnicas y todo lo que he aprendido gracias a ellas, pero, sobre todo, gracias por haberme hecho sentir como en mi casa. Sois tantos que sería imposible enumeraros sin dejarme a alguno. Muchos habéis dejado de ser compañeros de trabajo para convertiros en amigos.

I must also thank to professor Jose Silva Martinez and people from the Analog and Mixed Signal Center at Texas A&M University for letting me be part of their group for two months. I also have to mention Raghavendra Kulkarni for sharing his experience in the high speed I/O design with me and other members in the group for fruitful discussions on different topics and their warm hospitality.

También es hora de agradecer al Ministerio de Educación y Ciencia a través de las becas FPU (Formación del Profesorado Universitario) por su apoyo financiero durante estos últimos cuatro años. De la misma forma, es necesario también nombrar al CSIC (Consejo Superior de Investigaciones Científicas) por el apoyo brindado a través de las becas de iniciación a la investigación que disfruté durante mi penúltimo y último año de los estudios de Ingeniería de Telecomunicación.

Contents

1 Introduction

Artificial vision systems aim to extract high level information from complex non-structured scenes. Once reality is captured by some general or specific purpose platform, sophisticated algorithms establish relationships between high dimensional data and a low dimensional space where information can be easily retrieved. In robotic systems, the loop can be closed by using relevant features extracted from the computer vision algorithm to act over the observed reality and produce a control action. Achieving real time operation in the execution of these algorithms becomes relevant in these situations where artificial systems have to interact with their environment. Surveillance, industrial process control, automatic robot navigation or even modern video games are examples about fields of application of vision systems.

Hardware implementations of computer vision algorithms use to rely on high performance digital systems. The development of CMOS (Complementary Metal Oxide Semiconductor) technology has enabled the implementation of high performance vision processing engines which have been successfully introduced in the mass market. Moore's law established that the number of transistors on a chip will double about every two years. Processing power, measured in millions of instructions per second (MIPS), has steadily risen because of increased transistor density coupled with improved multiple core processor micro-architectures. Moore's Law also means decreasing costs per transistor since more transistors can be built on the same silicon wafer. The market of vision systems has been driven by this technological development during the last years.

However, Moore's law does not seem to be enough to satisfy the real time and low power requirements of modern vision systems. As CMOS technology is reaching the quantum limits of materials, it is believed that scaling down transistor dimensions will not drive the technological development of computing capabilities in the long run. Strong research efforts are being focused on architectural improvements of computer systems (e.g. multi-core technology) and exploring the post-CMOS era (e.g. nanotechnologies). Limitations of mainstream solutions to keep delivering more and more computing power throughout the next years pave the way for the research in new approaches for high performance vision processing systems.

The main drawback of using traditional computing architectures in vision systems is their intrinsic sequential nature. Conventional digital systems are intended to store an algorithm program and to execute a sequence of instructions one after the other. Due to the mathematical complexity and high parallelism of computer vision algorithms, this is a non-efficient solution in terms of computing resources utilization. Limitations of conventional computing infrastructures have led researchers to seek inspiration in nature. Human brain clearly outperforms artificial systems in sensory information processing tasks due to its massively parallel organization.

The essential block of a traditional computer is the Central Processing Unit (CPU). This circuit reads instructions and data from dedicated memories, performs operations encoded by the instructions over the fetched data and stores results back in proper locations. By contrast, the human brain is composed by a huge amount of low computing power processing units, neurons, which are connected between them through synapses, which form up a highly dense connection network. This architecture is very well suited for highly parallel tasks, such as sensory information processing. The main goal of neuromorphic engineering is to mimic this organization to build artificial systems which can carry out sensory processing tasks with similar performance than the human brain.

However, translating the massive parallelism of living brains into hardware systems is cumbersome due to the limited number of on-chip metal layers existing in modern VLSI (Very Large Scale of Integration) technologies and package pins to connect with off-chip components. Neuromorphic engineers have tried to develop large scale systems which emulate the brain organization and overcome this connectivity limitations of manufacturing by using the advantages of modern VLSI technologies in terms of communication bandwidth. As typical firing rates of neurons are in the order of a few Hz, a lot of them can be multiplexed in high speed communication channels which can work in the GHz range. Exploiting this fact, there is a strong research interest in extending the integration densities of current neural systems and opening the door for the implementation of complex vision tasks with them.

In this Chapter, we will further discuss the advantages of the spike-based bio-inspired vision processing and information encoding over the frame-based approach of conventional systems. However, the computational power of neural system to achieve all these advantages comes from the massive integration of neurons and synapses. Neuromorphic engineers have developed protocols to communicate spikes between large arrays of neurons using minimum hardware resources to overcome the connectivity limitations of CMOS technologies. In this field, the AER (Address Event Representation) protocol has been extensively used by the neuromorphic community and it will be analyzed in a dedicated Section. Finally, we will briefly present a hardware infrastructure capable of assembling a large amount of neuromorphic chips which communicate between them using the AER protocol. We will discuss the research challenges imposed by this architecture in order to achieve a reliable and low power consumption operation, while keeping the performance figures of spike-based vision processing systems.

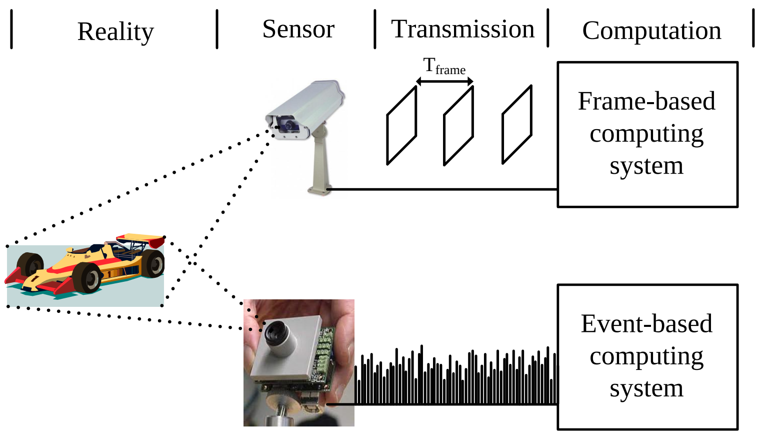


Figure 1.1: Conceptual illustration of frame-constraint (top) vs. a frame-free event-based (bottom) vision sensing and processing system.

1.1 Frame-constraint vs. Frame-free Event-based Vision Sensing and Processing

Artificial vision systems capture and process sequences of frames. For example, a video camera captures images at about 25-30 frames per second. Then they are processed frame by frame, pixel by pixel, to extract, enhance and combine features. Finally, operations in feature space would be performed until a desired recognition is achieved. This frame-based processing is slow, specially if many operations need to be computed for each input image or frame [1, 2]. Living brains do not operate on a frame by frame basis. In the retina, each pixel sends spikes (also called events) to the cortex when its activity level reaches a threshold. Pixels are not read by an external scanner, but they decide when to send an event. All these spikes are transmitted as they are being produced, and do not wait for an artificial "frame time" before sending them to the next processing layer.

Fig. 1.1 illustrates the conceptual difference between a frame and an event-based sensing and processing system. In the top row, a frame-constraint camera captures a sequence of frames, each of which is transmitted to the computing system. Each frame is processed by a sophisticated image processing algorithm for achieving some recognition. The computing system needs to have all pixels values of a frame before starting any computation. In the bottom row, an event-based vision sensor operates without frames. Each pixel sends an event (usually its own x,y coordinate) when it senses some property (change in intensity [3], contrast with respect to neighboring pixels [4], . . .).

Events are sent out to the computing system as they are produced, without waiting for a frame time and the computing system updates its state after each event. Fig. 1.2

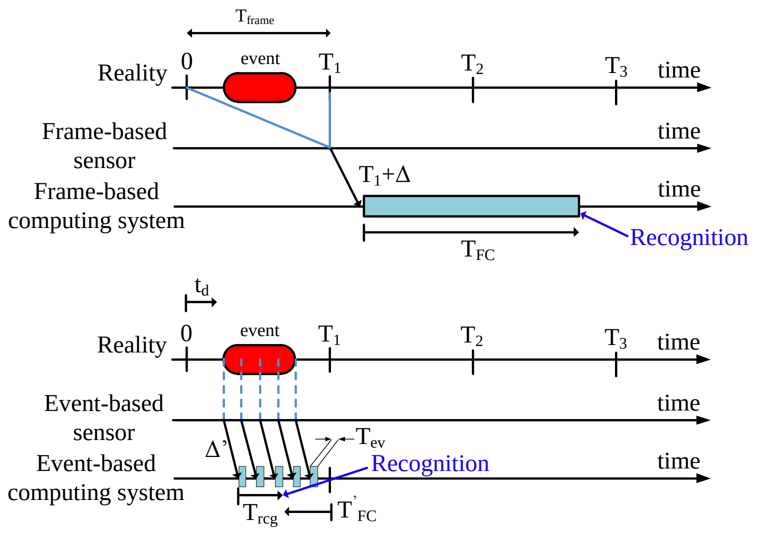


FIGURE 1.2: Comparison of timing issues between a (top) frame constraint and a (bottom) frame-free event-based sensing and processing system.

illustrates the inherent difference in timings between both concepts. In the top, (frame-constraint) reality is binned into compartments of duration T_{frame}. During the first frame T_1 an event happens (such as a flashing shape), but the information produced by this event does not reach the computing system until the full frame is captured (at T_1) and transmitted (with an additional delay Δ). Then the computing system has to process the full frame, handling a large amount of data and requiring a long "frame computation time" T_{FC} before the "recognition" information is available.

In the bottom of Fig. 1.2, pixels "see" directly the event in real time and send out their own events with a delay Δ' to the computing system. Events are processed as they flow with an event latency T_{ev} (in the order of ns). More relevant events usually come out first or with higher frequency. As soon as sufficient meaningful events are received and processed, it is possible to achieve recognition. Consequently, recognition time T_{rcg} can be smaller than the total time of the events produced. Note that the recognition is possible before frame time T_1, resulting in negative T'_{FC} when compared to the recognition delay in a frame-constraint system [5].

1.2 The AER protocol

AER is a promising emergent hardware technology which shows potential for providing the computing requirements of large scale multi-layer systems. AER was first proposed in 1991 in one of the CalTech research labs [6] and has been used since then by a wide community of neuromorphic hardware engineers. AER has been used fundamentally

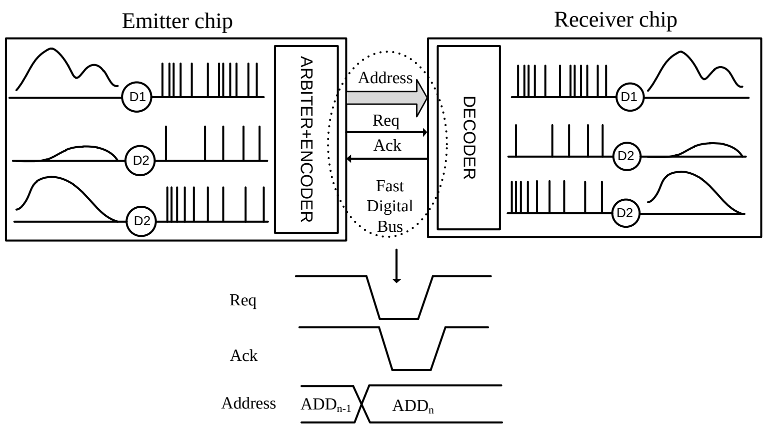


Figure 1.3: Rate-encoded point-to-point AER inter-chip communication link.

in image sensors, for simple light intensity to frequency transformations [3], time-to-first-spike coding [7, 8], foveated sensors [9], spatial contrast [4, 10] and more elaborate transient detectors [11]. But AER has also been used for auditory systems [12–14], competition and winner-take-all networks [15, 16], convolution processors [17–23] and even for systems distributed over wireless networks [24].

Fig. 1.3 shows the event communication in a point-to-point AER link. The aim of this link is to achieve unidirectional virtual connectivity from a neuron ensemble placed in one chip onto another one placed in a different chip. There are many spike coding schemes [7, 25, 26]. For example, neurons in the emitter chip can code their output as trains of spikes whose instantaneous frequency is proportional to their activation level. Other more efficient coding schemes have been proposed, such as rank-order coding [25] where the order of the events carries the information, instead of neuron spiking frequency. The continuous-time states of neurons D_i in the emitter chip are transformed into sequences of fast digital pulses (spikes or events) of minimal width (in the order of ns) but with much longer inter-spike intervals (in the order of ms). Spikes produced by all neurons are collected by a circuitry which arbitrates them to avoid collisions, and encodes the spikes as digital addresses corresponding with the destination neuron. Communication between chips is carried out using an asynchronous classic four phase handshaking protocol.

In this protocol, the emitter chip generates a request signal *Req* when it wants to transmit a new spike event. After latching the incoming event address, the receiver chip acknowledges this request with signal *Ack*. The transmitter disables the request signal, causing the same action at the receiver side and allowing the beginning of a new communication cycle. The receiver chip decodes the input address and sends a spike to the corresponding neuron within the array. Therefore, the AER protocol enables the connectivity between two sets of n neurons placed in different chips using only $log_2(n) + 2$ physical wires. The receiver neurons just have to integrate the input spikes to reconstruct the sender's activation state.

1.3 Multiple AER chips assembly

Let us suppose that we come out with a new spike based neural network architecture that we want to implement in hardware to carry out some high level information processing task. This network will have some neural computational units (neurons, synapses, complex neural circuits,. . .) which will be implemented in AER chips. To map the neural network topology onto a general purpose hardware infrastructure, we require a communication layer to efficiently distribute the spikes among chips. Fig. 1.4-(a) shows an example of a hierarchical multi-layer feed-forward neural system. The connectivity between blocks will vary from one application to another, but the neural computational units remain the same as long as they can be customized by defining proper configuration parameters. This programmability will define the neuron and synapses dynamics needed to carry out the desired computation.

It would be therefore desirable to have a hardware infrastructure where arbitrary connectivity patterns can be mapped. This way, a lot of different applications can be targeted using the same hardware by simply defining the architecture and blocks configuration parameters. In this dissertation, we envision a 2D mesh assembly of AER chips over the same PCB (Printed Circuit Board) where individual units only exchange events with their neighbors, as shown in Fig. 1.4-(b). As shown in Fig. 1.4-(c), each chip in the network would have an AER spike-based processor (analog or digital) and a routing engine which would provide the network intelligence. When a new event arrives at any network node, this router should either decide the next hop for the event to reach the final destination, or identify it as having reached its destination and forward it to the local spike processor. By separating the event communication layer from the event processing layer we achieve the connection programmability required to map a great variety of neural systems within the same hardware infrastructure.

The spike processor will also have configuration parameters to adapt the neural computation characteristics to the target application. Apart from the spike processor and the router programmability, other important aspects of this architecture are the scalability and the expandability issues. The 2D mesh grid can be easily expanded by connecting other AER systems or units around the multi-chip PCB neighborhood. As the network layer in charge of the event communication would be fully programmable, the system can be configured to send out events to other boards if required. As neuromorphic engineering is still an active research field, there is still room for new AER chips and concepts. Consequently, this expandability and scalability property is very interesting for adapting the system to the future developments of this research field.

Building this multi-chip assembly of AER chips over the same PCB to test new spike based neural network concepts is a long term research goal in our group. Several technical challenges arise when translating these system level ideas to a hardware implementation. The circuit level design must be carefully taken into account in order to ensure the system reliability and an affordable power consumption as a large amount of chips want to be assembled over the same PCB. This dissertation will address some of this implementation challenges as a first step of a long walk towards a fully operative multi-chip AER system. The infrastructure should be capable of mapping large scale

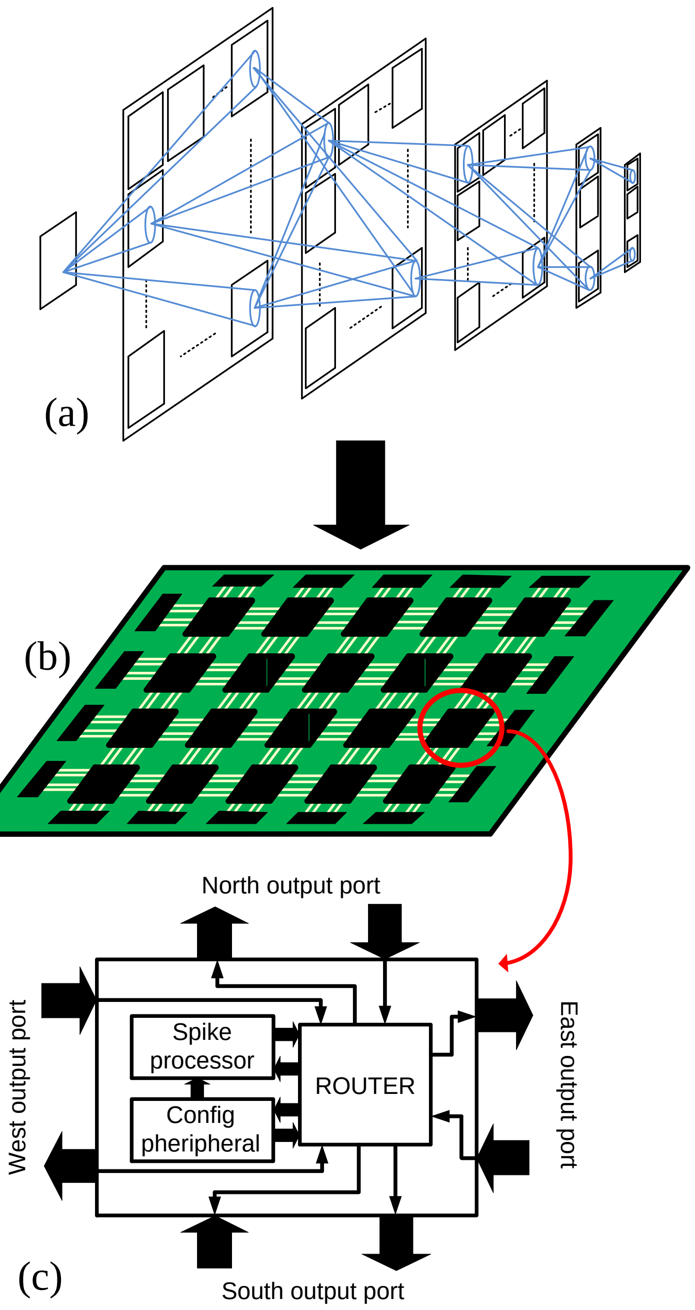


Figure 1.4: Hierarchical Multi-Module Neural System mapped onto a 2D grid of AER Hardware modules (a) Conceptual schematic topology architecture of target neural system. (b) 2D grid of AER hardware modules interconnected through PCB traces. (c) Schematic diagram of each module, including an event router plus event processor and configuration circuitry.

spike based neural networks for complex vision processing tasks.

1.4 Structure of this dissertation

This dissertation is divided into 6 Chapters, apart from this introductory part. Chapter 2 describes the existing large scale neuromorphic hardware platforms. Moreover, we introduce some basic concepts about convolutional neural networks and spike-based implementation of convolution processors. That will be useful in subsequent Chapters where this neural network paradigm will be used as an example of system integration on the proposed multi-module architecture. The other Chapters are focused on the research challenges that the large scale of integration of neuromorphic chips impose:

- **Communication layer design:** in Chapter 3 the routing algorithms, the addressing scheme and the routers circuit design for a 2D grid assembly will be extensively discussed, taking into account the particular characteristics of the AER traffic. In order to validate the suitability of this communication layer for the neuromorphic system implementation, we implemented a large scale assembly of spike based convolution modules in a commercial FPGA hardware infrastructure. We present a system level study of this particular example and experimental results to validate the performance of the proposed communication layer.
- **AER chips miniaturization:** classic AER links use parallel buses as the physical communication layer. However, this is not a scalable solution when hundreds of these chips need to be integrated on the same hardware infrastructure. For low pin-count, low cost and low power chip design it is a good choice to use LVDS (Low Voltage Differential Signaling) serial-bit AER links, instead of parallel ones. Conventional LVDS links with embedded clock need to transmit continuously to keep the link sender and receiver synchronized. However, in AER systems events are asynchronous and sparse. Consequently, considerable extra power savings can be achieved if the links are turned OFF during inter-event pauses, and quickly back ON when a new event needs to be transmitted. In Chapter 4, a Manchester-encoding Serializer/Deserializer scheme is proposed to overcome the limitations of mainstream serial link solutions. This circuit can turn the link OFF and back ON with zero-bit acquisition time, exploiting the asynchronous nature of the AER traffic and scaling the power consumption with the transmitted event rate. Chapter 4 describes the circuit design and presents experimental results for a $0.35\mu m$ CMOS prototype.
- **Low power consumption:** the burst mode SerDes proposed in Chapter 4 uses conventional high speed LVDS drivers and pads. These circuits consume several mWs per link even when no data is transmitted. Power down modes on such drivers' large current sources can be very slow, thus losing the benefits of burst mode AER SerDes links. Chapters 5 and 6 describe two switchable implementations of the I/O circuits to increase the power reduction benefits of turning off the communication circuits activity during pauses. These Chapters deal with the circuit design techniques required to keep the I/O circuitry performance in

terms of speed and latency, but scaling down the power consumption with the transmitted event rate. Extensive experimental results are also provided to check the feasibility of the applied design techniques.

- **Prototyping hardware infrastructures:** Chapter 7 presents the design of a prototyping board intended to easily and quickly assemble multi-chip AER systems, the Node Board. This FPGA-based board is intended to implement the network layer tasks of the individual modules shown in Fig. 1.4-(b) and provide a hardware-level configurable architecture to study real large-scale network implementations using existing AER chips. Moreover, users can also implement their own spike processing algorithms on the FPGA to test new functionalities and features on their future chips. The Node Board is a useful tool to easily and quickly develop all the features of the multi-chip system before going to an eventual silicon integration.

2 Existing Large Scale Neuromorphic Hardware Platforms

2.1 Introduction

In the last years we have witnessed the development of new bio-inspired solutions to build high performance large scale vision systems. Large scale hardware implementation of neural systems is becoming a major research area to achieve a similar performance than biology. Advantages of bio-inspired sensory information processing come from the high degree of parallelism existing in living brains. For example, the human brain is made of a huge amount (in the order of 10^{11} [27]) of computational units, the neurons, massively inter-connected (it is believed that there are around 10^{15} connections between neurons in the brain [28]). Building artificial systems that can perform complex information processing, such as in the human brain, will require to assemble millions of artificial neurons and synapses. Present silicon VLSI technologies do not offer enough connectivity resources and neuron integration density to mimic realistic large scale neural systems.

Several current research projects aim at the exploration of novel computational aspects of large scale, biologically inspired neural networks. These architectures integrate over a million neurons, operating in real time or even with a speed-up with respect to the biological archetypes on full custom or modified general purpose hardware. In this Chapter, several approaches to build large scale neural systems will be reviewed. All of them face the problem of integrating a lot of artificial neuron silicon circuits over a spike-based network which provides the communication layer between all computing units. Hardware implementations have to deal with the physical neuron and synapses implementation, event routing through the spike-based network, system programmability in terms of neuron dynamics and synapse connections, observability,... However, this common problem can be addressed from very different point of views, leading to the approaches described in the following Sections.

Once we have drawn the big picture, we will focus the attention over a very promising framework to build scalable and expandable neural vision processing infrastructures: the convolutional neural networks (ConvNet). ConvNets are a very mature tool among the frame-based pattern analysis and machine learning community, and are exploited in numerous commercial products [29–38] and research labs [2, 39, 40]. The interesting scaling properties of these kind of neural networks will be analyzed. A reported hardware accelerated platform [41, 42] to implement large scale ConvNets will be described to establish the state of the art in this particular field. In contrast with this frame-based solution, a spike-based alternative to implement the convolutional modules will be presented at the end of this Chapter. The ConvNet paradigm will be extensively used along this dissertation as an example of large scale spiking neural network suitable to be implemented in scalable and expandable AER multi-module systems.

2.2 SpiNNaker project

The biological processing of a neuron can be modeled by a digital processor. Axon connectivity can be represented by messages, or information packets, transmitted between a large number of processors which emulate the parallel operation of the billions of neurons comprising the brain. Using this philosophy, the *SpiNNaker* project aims to build a large scale fully digital multi-chip system which mimics the human brain's biological structure and functionality. *SpiNNaker* chips do not physically implement artificial neurons, but emulate the neural dynamics by solving differential equations in a numerical domain. In this environment, neurons operate in real time and there is no requirement for explicit synchronization in the computation because neurons independently fire events when their action potential reaches a certain threshold.

Fig. 2.1 shows the mesh topology of the multi-chip *SpiNNaker* system [43]. Neural models, routines written in assembly code, run in software on embedded ARM968 processors. Neurons communicate by means of spike packets directly supported by a multi-cast, packet-switched and self-timed network. Each *SpiNNaker* chip contains several of these ARM processors which communicate with each other asynchronously through a fabric NoC (Network on Chip). The chip can send out spikes to its counterparts through a packet-switched network connected using a 2D toroidal triangular mesh. To emulate biological system's very high connectivity, the on-chip routers provide multicast routing.

Fig. 2.2 shows a block diagram of a multi-core *SpiNNaker* chip [44]. Each one can contain up to 18 identical ARM9 processing cores running at 200MHz. The core is directly connected to on-chip memory blocks which store the synaptic tables. They retrieve the proper synaptic weights every time that an event is received by analyzing the spike address. Moreover, they accelerate the interruption routines execution which calculates the neuron dynamics. One of the cores on each chip is selected to perform

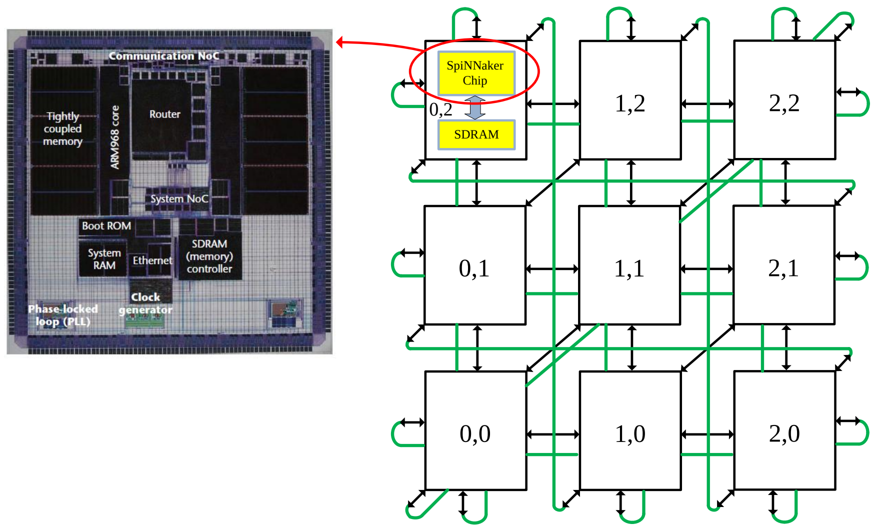


Figure 2.1: Mesh topology of the *SpiNNaker* system.

system management tasks. The other processing cores run independent event-driven neural processes, each of them simulating single neuron dynamics. A direct memory access (DMA) control manages the access of individual cores to the on-chip communication NoC. The neural state, as well as parts of synaptic tables for neurons, are stored in an off-chip SDRAM whose access is granted by an on-chip controller PL340 SDRAM I/F. Ethernet ports are used to provide connectivity for debugging and traffic injection with a host PC.

The key block for large scale processor core assembly is the packet router, which is in charge of the on-chip and off-chip communication management. It has 18 ports for internal use of the ARM cores and six ports to communicate with 6 adjacent chips. All are full-duplex ports which implement asynchronous self-timed protocols. The router is designed to support point-to-point and multicast communications. The multicast engine helps to reduce pressure at the injection ports and reduces significantly the packet traffic through the network compared with a pure point-to-point alternative. To implement high fan-out connections, the routing decisions are based on the spike source neuron identifier. This synaptic connections are embedded in the 1024-word routing tables inside the routers which are configured by the user.

2.3 BrainScaleS project

The BrainScaleS project is a continuation of a previous one, called FACETS (Fast Analog Computing with Emergent Transient States), which aims to build accelerated time large scale spiking neural networks hardware. The acceleration factor ranges from 10^3 to 10^5 compared to the biological real time (BRT). Besides this accelerated time capability, the BrainScaleS system differs in its hardware implementation from the

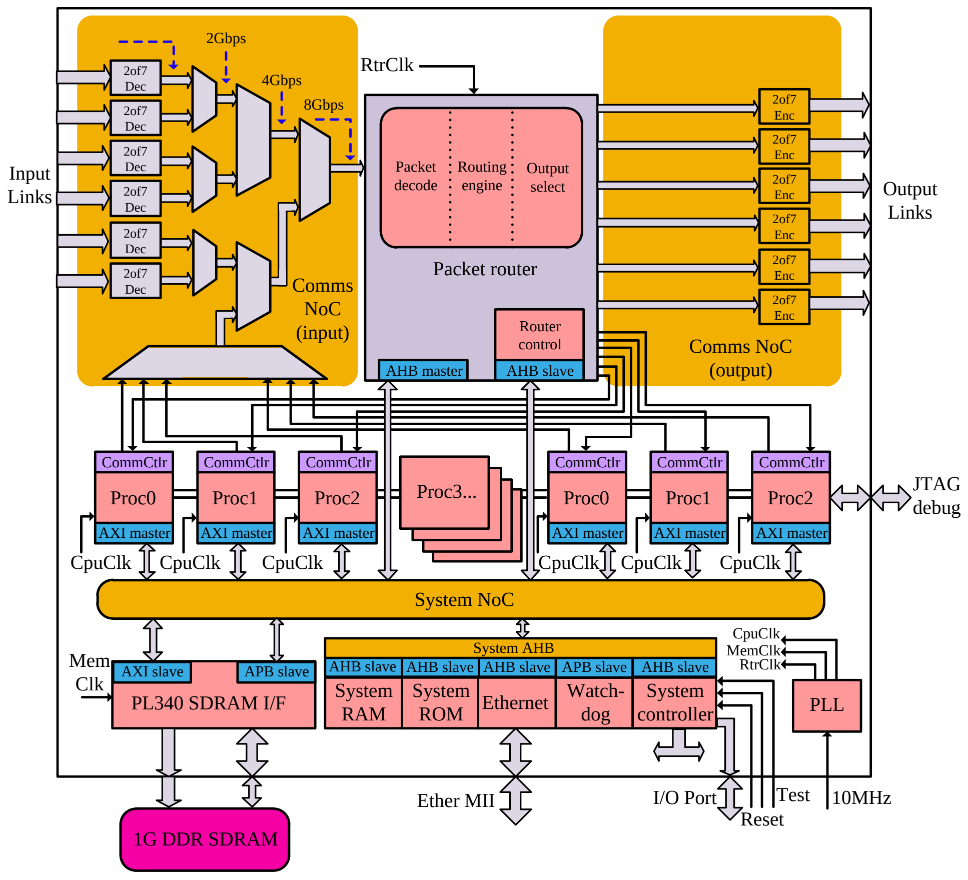


Figure 2.2: *SpiNNaker* chip organization. The top half contains the multicast router that distributes spikes to cores on the same and neighboring chips. The ARM cores and their peripherals occupy the bottom half.

SpiNNaker solution. In this case, the neural computation emulation is carried out through analog circuits which mimic the synapse and neuron dynamics and a large scale spike-based digital network is implemented in a wafer-scale system to enable the necessary high density of neuron connections. Fig. 2.3 shows a 3-D representation of the wafer-scale neuromorphic hardware system developed in the framework of the FACETS project.

Fig. 2.4 shows the integration hierarchy of the wafer-scale system [45]. Due to the high acceleration factor required, communication bandwidth in-between the computation units, the Analog Network Chips (ANC), can exceed 10^{11} neural events per second. Handling this very high throughput is a very demanding task which is accomplished in this case through switch-based wafer-scale interconnections to implement local connectivity, along with a spike-based digital network for long range connections. In this technology, silicon wafers containing the individual chips are not cut into dies. Instead, chips are interconnected directly on the wafer. This integration technique can

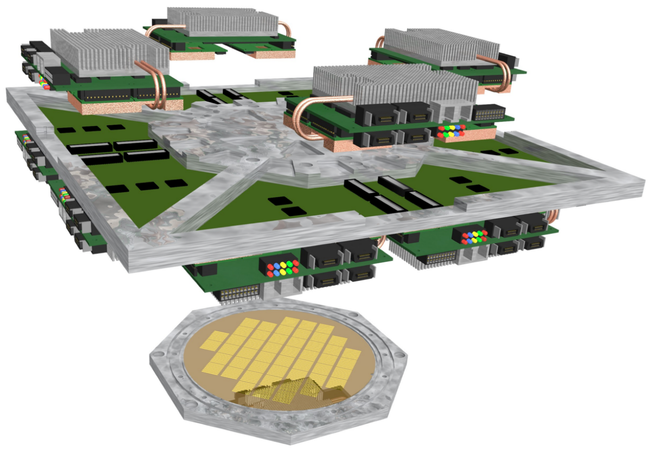


Figure 2.3: Wafer-scale neuromorphic hardware system 3-D assembly.

be designed to have enough fault tolerance and low power consumption for a very large scale neural system implementation.

The HICANN chip, shown in the lower left corner of Fig. 2.4 (hierarchy level 2), is the basic building block of the wafer-scale system. It contains the mixed signal neuron and synapse circuits as well as the necessary support circuits and the host interface logic. The interconnections between the HICANN chips run vertically and horizontally through the chip, with cross-bar switches at their intersections. Users can configure the network topology by specifying on/off states for these switches. The hierarchy level 1 (shown in the top left part of Fig. 2.4) refers to the arrangement of the HICANN chips on the wafer. Eight HICANN chips can be integrated on a single reticle. An extra post-processing step in the regular CMOS process is required to build the horizontal and vertical wires which connect the wafer reticles. This additional step consists of a metal layer deposition atop the fabricated wafer. Apart from the on-wafer chip connection, these wires also link the wafer-scale system with a higher level mother board which manages the spike communication.

The right part of Fig. 2.4 shows the ANNCORE (Analog Neural Network Core) structure. It contains 128K synapses and 512 membrane circuits which can be grouped together to form neurons with up to 16K synapses and 8 neurons. On the other side, using the maximum number of neurons of 512 limits the number of input per neuron to 256. As each neuron has a very high number of input signals, the bandwidth required to handle the spike events throughput is not affordable (it can be estimated in the order of 164Gevent/s [45]). The ANNCORE uses a combination of space and time multiplexing to make this communication demand feasible. The spatial density is achieved thanks to the high density of connections that the wafer-scale integration enables. Up to 256 differential bus lanes can be used to send spikes towards groups of 64x64 synapses. Temporal multiplexing is used as the final step to reach the necessary numbers. Each wire pair carries events from 64 pre-synaptic neurons by serially transmitting 6 bits

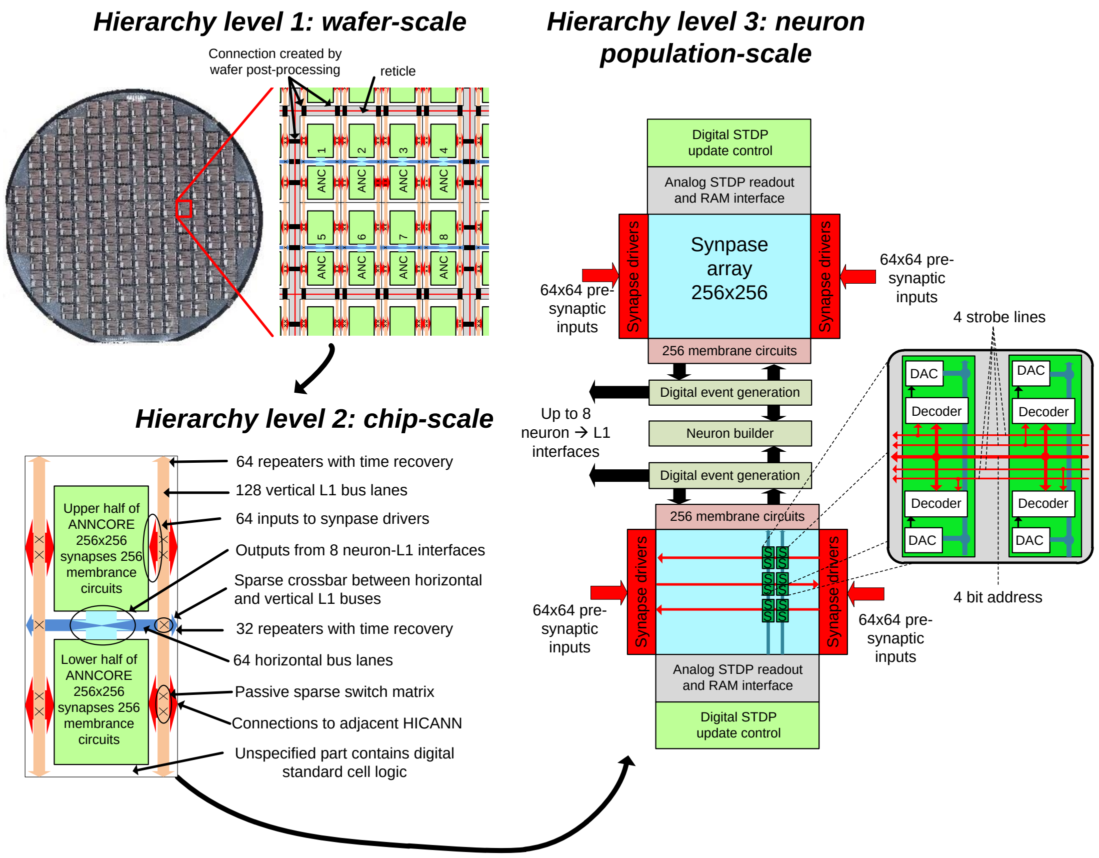

Hierarchy level 1: wafer-scale
Connection created by wafer post-processing
reticle
ANC
Hierarchy level 2: chip-scale
64 repeaters with time recovery
128 vertical L1 bus lanes
64 inputs to synpase drivers
Outputs from 8 neuron-L1 interfaces
Sparse crossbar between horizontal and vertical L1 buses
32 repeaters with time recovery
64 horizontal bus lanes
Passive sparse switch matrix
Connections to adjacent HICANN
Unspecified part contains digital standard cell logic
Upper half of ANNCORE 256x256 synapses 256 membrance circuits
Lower half of ANNCORE 256x256 synapses 256 membrance circuits
Hierarchy level 3: neuron population-scale
Digital STDP update control
Analog STDP readout and RAM interface
Synpase array 256x256
Synapse drivers
64x64 pre-synaptic inputs
256 membrane circuits
Digital event generation
Neuron builder
Up to 8 neuron → L1 interfaces
4 strobe lines
DAC
Decoder
4 bit address

Figure 2.4: Wafer-scale system hierarchy levels of neural elements integration.

neuron numbers. No time stamps are required if the communication links provide enough bandwidth to satisfy the accelerated time requirement.

A custom backplane connects the wafer units to each other. The motherboard contains Digital Network ASICs (DNC) [46] which interface the analog cores on the wafer and several FPGAs (Field Programmable Gate Array) interconnecting the wafer boards between them in a virtual spike-based network. These FPGA chips implement the necessary communication protocols to exchange event packets between the different network wafers and the host computer to analyze results. The intention in a later stage of development is to connect the neuromorphic hardware described in this subsection with high performance numerical computers to further extend the system multi-scale emulation capabilities. The merging of the two computational concepts into a hybrid system provides a new experimental platform where different levels of neural modeling can be integrated over the same experiment. Moreover, new sensory stimuli and motor feedback can be introduced to perform cognitive tasks.

2.4 Multi-chip AER systems

The AER protocol is not only a useful tool to implement point-to-point connectivity between neuromorphic chips, but it can also be used to assemble several AER modules to perform a sensory information processing task. Several research groups have proposed solutions to implement large scale systems where individual units communicate with each other using the AER protocol. An efficient communication layer to control the event transfer and system configuration is needed to assemble a large amount of neuromorphic blocks. Using the asynchronous digital approach provided by the AER protocol, neuromorphic engineers seek novel hardware infrastructures which have low power consumption and good scalability and expandability properties.

The simplest way to implement hierarchical systems using AER modules is using a mapper-based solution, as it is shown in Fig. 2.5. Events traffic is controlled by a central master element which virtually connects the physical elements with the information contained in a mapping table. Chips output events are merged into a single AER bus and the mapper assigns a destination address on the global addressing map defined for the system. Fig. 2.5-(a) shows a multi-chip system intended to be used in a model for orientation selectivity [15]. A PCI-AER board is the key element for the communication because it acts as a bridge between the host PC, a neuromorphic retina (*TMPDIFF* in Fig. 2.5-(a)) and a recurrent competitive network of integrate-and-fire neurons with short-term dynamic synapses (IFWTA chip in Fig. 2.5-(a)).

Fig. 2.5-(b) shows the mapper-based platform which manages a dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses [47]. In this case, an FPGA which implements a controller unit (*MCU*) collects events from an integrate and fire neurons array (*I&F*) and interacts with an external interface (*DIO*), a RAM memory which contains a routing table where the connectivity is described and a digital-to-analog converter (*DAC*) which sets the analog biases of synaptic parameters while the corresponding event indicated by the *MCU* is delivered to the target neurons within the array.

This idea can be extended to a multi-dimensional structure by adding extra levels of routing hierarchy, as shown in Fig. 2.5-(c) [48]. Output events coming out from the local mapper AER block can be re-directed to the same array or to neurons located in other arrays. A higher level router receives events from different arrays. This second level routing algorithm chooses the proper array which must process the event or, if extra layers are needed, events are sent to the next router in the hierarchy level. The infrastructure can be scaled up to N dimensions by adding extra levels of hierarchy.

Fig. 2.6 illustrates another approach to implement large scale AER systems. Spikes are broadcast in a 1-D grid to multiple arrays of silicon neurons in a daisy chain manner. When an AER chip receives an incoming event, it decides if the event must be processed by itself or if it must be transmitted to its neighbor. The grid can be considered expandable because, unlike a bus, its capacity does not decrease as more chips are added because physical connections are point-to-point. An asynchronous communication infrastructure is used to minimize the latency that the event broadcast introduces. Events addresses identify the neuron which generated the spike and

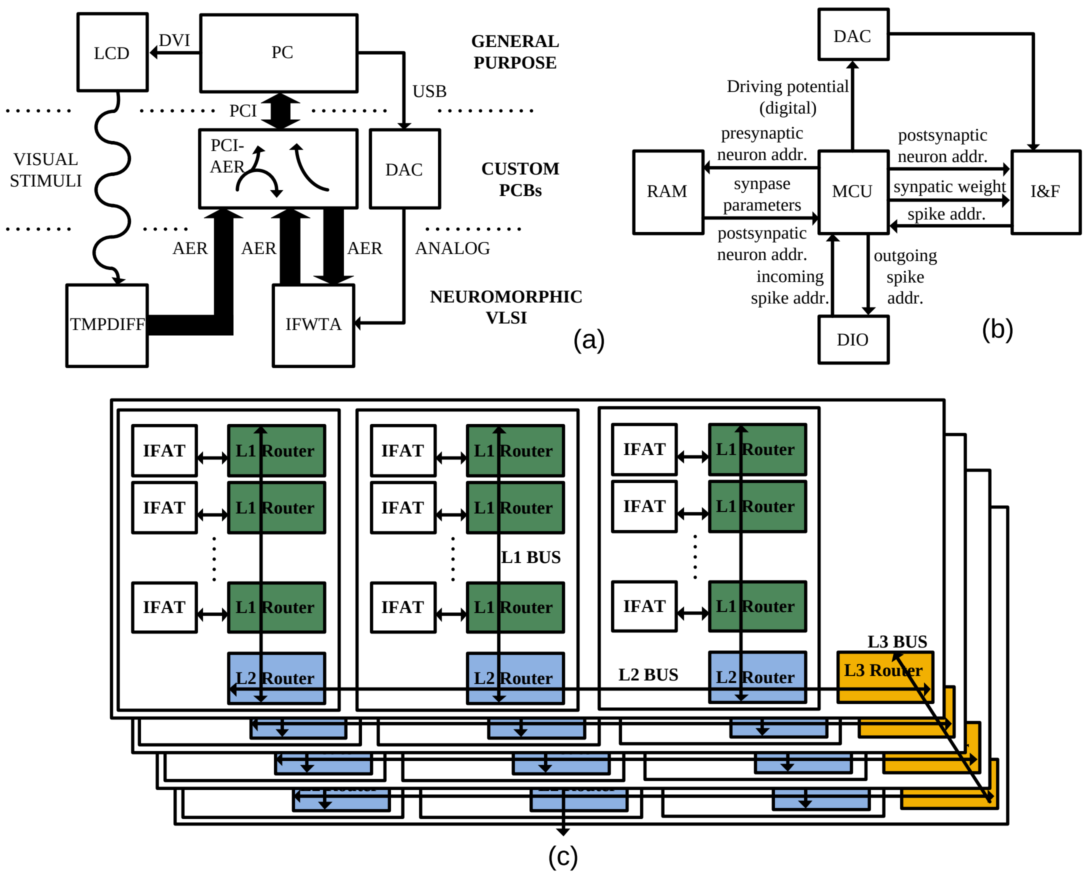


FIGURE 2.5: Mapper based multi-chip AER solutions. (a) Model for orientation selectivity hardware implementation (b) Block diagram of the IFAT (Integrate and Fire Array Transceiver) system. (c) 3-D network extension of the previous concept.

information is transmitted in a word-serial manner through the network [49–51].

Fig. 2.7 shows the experimental set-up of a multi layer vision system for ball tracking built with the CAVIAR philosophy [52]. The approaches discussed until this point implement the physical connections between blocks in a virtual manner. On the contrary, the CAVIAR system uses pre-wired connections between blocks which communicate through point-to-point links. Splitter and merger blocks are used to increase the blocks fan-out by replicating their output traffic without any extra latency penalty or to multiplex different source traffic into a single AER flow, respectively. Mapper elements are also used to sweep between addressing spaces. This arrangement also provides easy ways to monitor the system traffic by inserting monitoring blocks in internal test points. A photograph of the system composed by a DVS retina, 4 convolution chips, one WTA (Winner Take All) chip, one delay line and one learning chip is shown at the bottom part of Fig. 2.7.

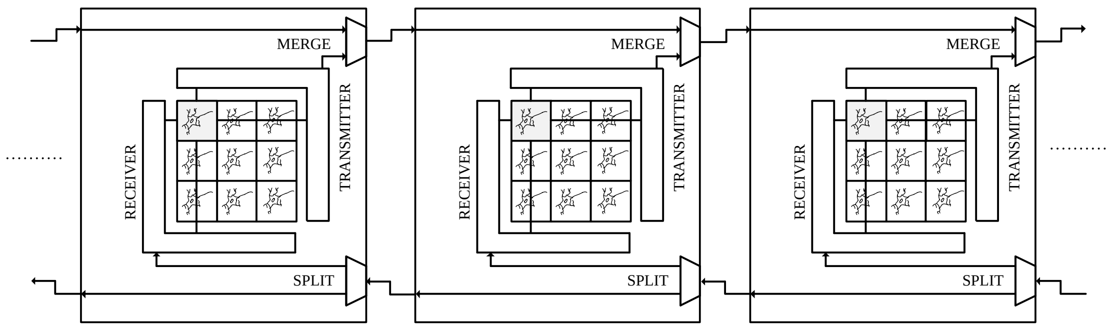


Figure 2.6: The broadcast mesh AER grid for expandable neuromorphic chips networks.

2.5 Convolutional Neural Networks

Architectures presented in previous Sections are designed to connect arrays of artificial neurons and synapses, built as digital or analog silicon circuits, allowing to map any kind of spiking neural network topology (only restricted by the hardware limitations). However, researchers have demonstrated that efficient vision systems can be built exploiting the projection field concept, illustrated in Fig. 2.8. Each neuron in one layer connects to a projection field of neurons in the following layer. The weights of synaptic connections follow a pattern which is independent of neuron position within a sending layer. Thus, the projection field based computation allows to simplify the synaptic connectivity of the neurons and improve the scalability of neural systems without compromising the computational power.

Vision processing based on projection fields is similar to convolution based processing [53], at least for the earlier cortical layers. For example, it is widely accepted that the first layer of the visual cortex V1 performs an operation similar to a bank of 2D Gabor-like filters at different scales and orientations [2] whose actual parameters have been measured [54–56]. This fact has been exploited by many researchers to propose powerful convolution based image processing algorithms [1, 2],[57–62]. In 1959 Hubel and Wiesel reported their findings on projection field processing in early stages of visual cortex, receiving the 1981 Nobel Prize. Based on these findings, ConvNets were originally proposed by Fukushima in 1969 [57]-[58] and further developed by Yann LeCun [59] and other groups, as a type of continuous-time gradient-based learning neural paradigm, with great success in a variety of (industrial) applications as well as research.

Examples of industrial applications and developments are, to mention a few: (1) NEC with products for face/person detection, age and gender recognition for vending machines [37, 38], as well as prototypes for cancer cell detection or mobile phone imaging applications, (2) France Telecom/Orange with face detection and recognition, text detection and recognition [31], various mobile phone applications, (3) Vidient Technologies with products for video surveillance, human detection and tracking, (4) Canon with cameras with embedded video surveillance, (5) Microsoft with handwriting recognition [32–36], (6) AT&T/Lucent-Technologies NCR with products for check

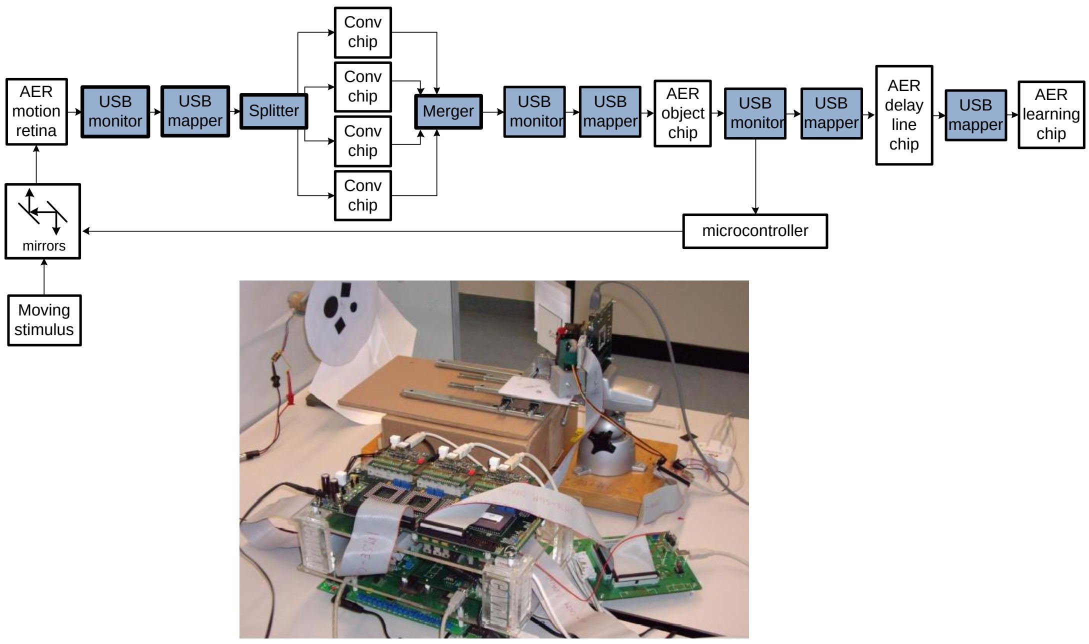


Figure 2.7: Experimental setup of multilayered AER vision system for ball tracking (final demonstrator of CAVIAR FET project).

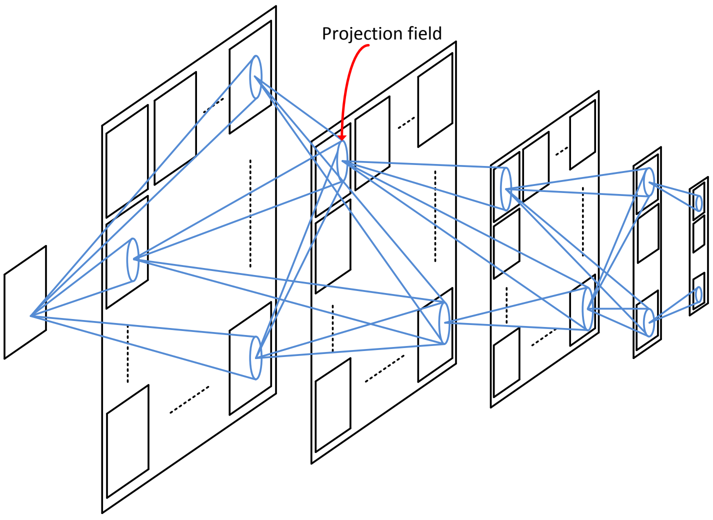


Figure 2.8: Typical structure of a feed forward convolutional neural network.

recognition [30]. Examples of state-of-art research exploiting ConvNets are (1) Poggio at MIT with object recognition and scene analysis [2], (2) Seung at MIT with image segmentation and biological image analysis (brain circuit reconstruction) [39] (3) NEC Labs with natural language processing and understanding [29], (4) NYU with biological image analysis, object recognition and visual navigation for robots [40]. All these developments are based on frame-based image processing, using sensory input from a conventional video camera.

On the other hand, in 1996, Thorpe demonstrated that the human visual system is capable of performing object recognition tasks at such speeds that any neuron involved only had time to fire one spike [63]. Based on this finding, he developed a framework for spiking ConvNets, which is presently being exploited commercially for high speed object recognition software [64].

In the field of VLSI circuit design we have witnessed, during the past years, important developments in the field of spiking neural hardware, and specifically neural spiking hardware for ConvNet processing [20, 21, 52] of visual information sensed by highly efficient spiking sensors [11]. It is now becoming apparent that the combination of ConvNet theories and knowledge, the framework of spiking information sensing and processing and the state-of-the-art hardware technologies will result in highly efficient systems for sophisticated cognitive tasks, similar to the human brain.

2.6 Scalability properties of convolutional neural networks

Fig. 2.8 shows a typical ConvNet architecture. It usually contains a reduced number of sequential layers (4-10), each of which performs several 2D filtering operations in parallel. Early stages extract simple features (such as edge orientation and scale), which are progressively combined into more complex shapes and figures at later stages. Early stages usually operate with small but dense kernels, while later stages use longer range but sparser ones [2]. To increase the knowledge (dictionary of shapes and figures) of the system one simply adds more 2D filters in later layers. Example of ConvNet systems for face and character recognition applications may have several ten to hundreds filters per layer.

What is interesting about ConvNets, compared to other neural networks, is their graceful scaling capability. To increase knowledge one simply has to increase the number of convolutional modules in a layer. Thus, number of neurons (pixels) scales linearly with the number of modules. On the other hand, the latency of the computing structure (if implemented as parallel hardware) is determined mainly by the number of sequential layers, which is a reduced number and does not change for a given application. Therefore, speeds do not degrade by adding more modules per layer (more knowledge) at least in first order. In other neural network architectures, the number of synapses scales quadratically with the number of neurons. Consequently, ConvNets seem very appealing for configurable, modular and scalable spiking hardware implementations.

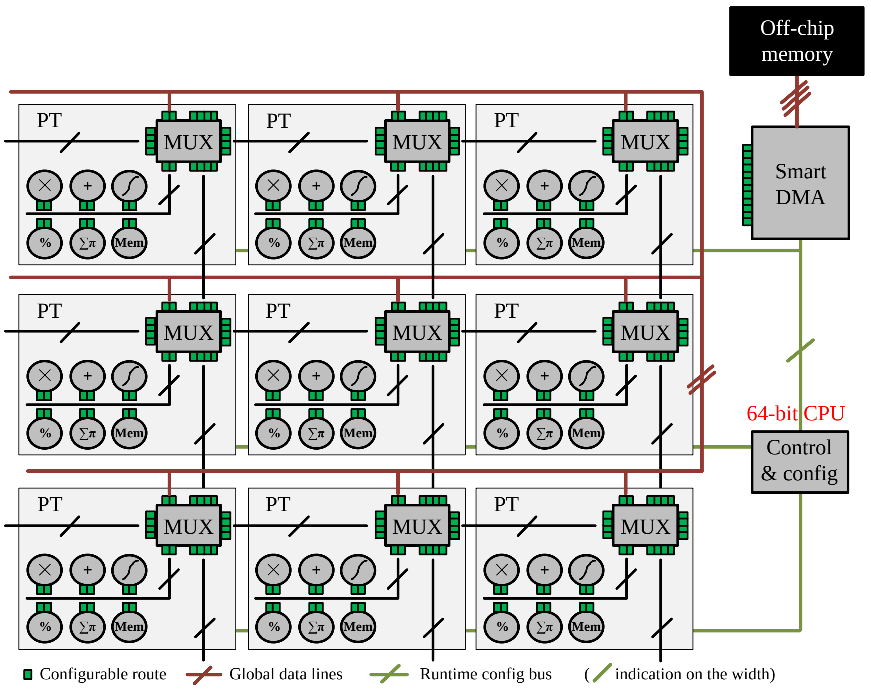


FIGURE 2.9: A runtime reconfigurable dataflow grid of processing tiles (PTs).

2.7 NeuFlow vision system

NeuFlow is a frame-based hardware architecture for large-scale multi-layered synthetic vision systems based on filter banks which exploits the already analyzed advantages of ConvNets. This vision processor is a dataflow engine which can perform real time detection, recognition and localization in mega-pixel images processed as pipelined streams. Fig. 2.9 shows the high level architecture of the vision processor, where its main blocks have been highlighted. The design is a custom single instruction multiple data (SIMD) processor based on a 64 bit CPU with hardware-accelerated instructions. Operations are highly optimized and make use of the parallelism available in hardware.

The key blocks of the computing architecture are described below:

- **Processing tiles (PTs):** they are independent processing tiles laid out on a two-dimensional grid. The PTs contain a routing multiplexer (*MUX*) and local operators, which can range from adders, multipliers or dividers to multiply-and-accumulate arrays (to efficiently compute convolutions) or non-linear mapping engines (to efficiently compute non-linear functions). These operators are fully pipelined to produce one result per clock cycle. Each node has its own configuration parameters, routes or settings, despite of a unique address to identify it within the array.
- **Control Unit:** an agile re-configuration procedure is one of the key aspects of the hardware acceleration capabilities. The grid offers restricted connectivity

Table 2.1: ConvNet implementations comparison: 1-Intel DuoCore: laptop-class CPU, 2.7GHz, optimized C-code, 2-neuFlow on Xilinx Virtex 4/6 (actual measurements) 3-NeuFlow on IBM 65nm process (post place and route simulations), 4-two GPU implementations: low power GT335nm and high-end GTX-480.

	Intel 2Core	neuFlow Virtex4	neuFlow Virtex6	nVidia GT335m	neuFlow IBM 65nm	nVidia GTX480
Peak GOP/sec	10	40	160	182	1200	1350
Actual GOP/sec	1.1	37	147	54	1102.5	294
FPS	1.4	46	182	67	1365	374
Power(W)	30	10	10	30	3	220
Embed(GOPs/W)	0.037	3.7	14.7	1.8	367.5	1.34

because each tile can only be connected to its neighbors and to a few global N-to-N data lines. Instead of a hard-wired solution, the communication relies on a NoC to efficiently broadcast configuration packets. The control unit interfaces the NoC to reconfigure the grid at runtime with reconfiguration times in the order of tens of μs.

- **Smart DMA:** this multi-port memory management block is specifically designed for image manipulations. The streaming engine interfaces any kind of memory module (internal or external) and offers Nx16 bit asynchronous read/write ports on the other side. This design allows simultaneous streams from/to the same memory locations. A dedicated arbiter is used to multiplex/demultiplex access to the external memory with high bandwidth.

The typical execution of an instruction on this system is the following: 1) the CPU configures each tile to be used for the computation and each connection between the tiles and their neighbors and/or the global lines, by sending a configuration commands to each of them, 2) it configures the streaming engine to pre-fetch the data to be processed, and to be ready to write results, 3) when the streaming engine is ready, it triggers the streaming out, 4) each ALU processes its respective incoming data, and passes the results to another tile, or back to the streaming engine, 5) the CPU is notified of the end of the operations when the streaming engine has completed. The behavior of each port in the smart DMA block can be configured separately by using the configuration bus.

Table 2.1 shows a comparison between possible implementations of ConvNets in existing hardware platforms. The results correspond to the computation of $16 \times 10 \times 10$ filter bank over a $4 \times 500 \times 500$ input image. *NeuFlow* implementations on Virtex-4 and Virtex-6 FPGAs have been tested in hardware and results for the IBM 65nm prototype are estimated using data taken from the technology. The *neuFlow* is a good example on how ConvNets can be used for high level processing tasks, such as object or human face recognition, outperforming the conventional hardware platforms for vision system.

2.8 AER convolution chips

Convolutions are computationally expensive. It seems unlikely that the high number of convolutions which might be performed by the brain could be emulated fast enough by software programs running on the fastest of today's computers. *NeuFlow* offers a powerful hardware platform to implement ConvNets which can perform highly sophisticated vision processing tasks. However, it still relies on the frame concept for the computation. Along this Chapter, the event-based neuromophic approach advantages have been presented over the frame-based concept. Exploiting the opportunities given by the event-based computation, ConvNets performance in terms of processing speed and power consumption can be improved. Event-driven hardware convolution units become essential building blocks to assemble ConvNet architectures which operate on spike-driven bases.

Some AER convolution processing chips with hardwired kernels (slightly tunable) have been proposed in the literature [17, 18]. However, it was not until arbitrary-shape-kernel convolution chips became available (with [19] or without kernel symmetry restrictions [20]) that their potential for building large scale AER ConvNets for arbitrary pattern and object recognition applications became apparent. Several AER fully-programmable-kernel convolution chips have been reported; either mixed-mode based on pixel-level charge packet integration [20, 21], or fully digital with in-pixel accumulator and adder to emulate leaky integrate-and-fire neurons [22, 23].

Fig. 2.10 shows the conceptual diagram of a fully digital AER convolution chip [22, 23]. It contains a synchronous controller with an internal clock, a 32x32 4-bit words static kernel-RAM, a kernel parameter lookup table (LUT), a column reader, a 2's complement block, a left/right column shifter, an array of 64x64 pixels, and an asynchronous event read out block [65]. Event-driven convolutions are performed as follows. Pixels (x,y) in the *Pixel Array* (see Fig. 2.10) hold their state in a continuous and dynamic manner. When the module receives an input event described by $(x_{in}, y_{in}, s_{in}, k_{in})$, kernel k_{in} (which is a 2D matrix stored in the *Kernel-RAM*) is added/subtracted to the *Projection Field* of pixels around *Event Address* (x_{in}, y_{in}). Input event sign s_{in} determines whether the kernel is added or subtracted. When a pixel reaches a positive (or negative) threshold, it is reset to its reseting level, and a positively (or negatively) signed output event is sent through the AER output port with the pixel coordinates. Independently to the input event flow, all pixels "suffer" from a constant rate leak which will drive their state to a resting level when no input events modify the pixel state.

Event-driven convolution modules need about 100ns to 1μs to process each event, depending on the kernel size. Each convolution chip needs to collect a given number of space-time correlated input events to provide an output event, depending on the module settings. For high-speed processing, one can set this number to be around ten events or less. In general, more relevant pixels in the sensor have stronger signals and send out their events sooner or more frequently. Consequently, more relevant events will be processed first by later convolution chips. This way, in an object recognition hierarchical ConvNet, recognition can be achieved as soon as the sensor provides enough

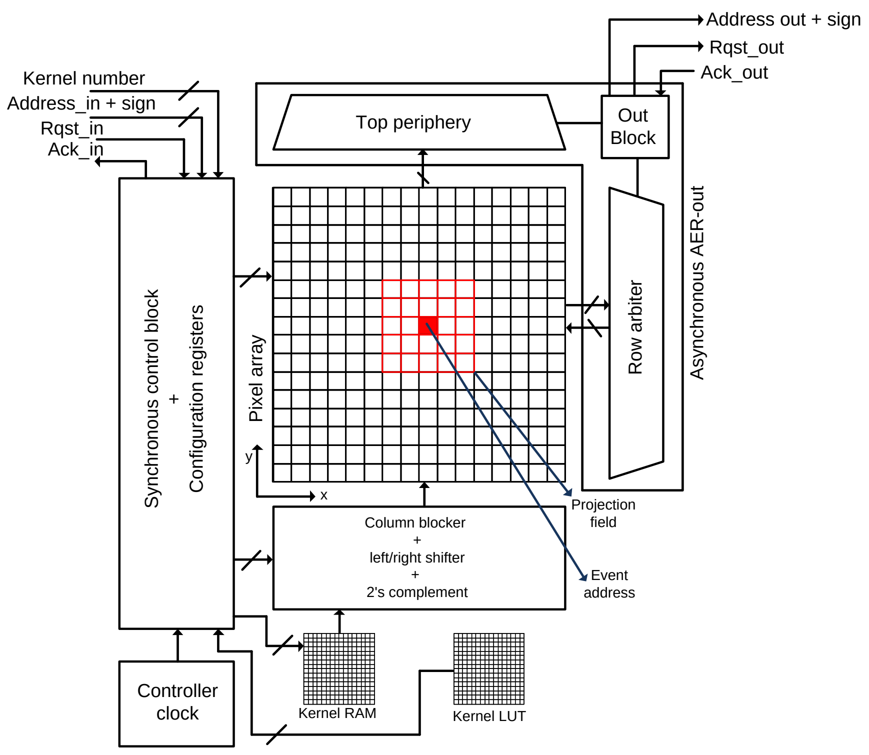


FIGURE 2.10: Event-based convolutional module architecture.

significant space-time correlated events. We refer to this as the "*pseudo-simultaneity*" property of event-driven convolution processing.

The second interesting property of implementing event-driven convolutions (or other operators, in general) is its modular scalability. Since event flows are asynchronous, each AER link between two convolutional modules is independent and needs no global system synchronization. Event-driven architectures are greatly simplified by this fact because the need of time-stamping events is avoided and there is no global clock. For this reason, the "*pseudo-simultaneity*" property of individual convolution modules also applies to large scale multilayer and multimodule systems.

The third interesting property of spike based hardware, in general, is that since processing is per-event, power consumption is, in principle, also per-event. Since events usually carry relevant information, power is consumed when relevant information is sensed, transmitted and processed. This is not completely true in convolution chips due to the leak mechanism which remains active, even when there are no input events. That leads to a fixed amount of background event-independent power consumption, apart from a variable term which depends on the chip input activity.

Besides the neuFlow platform, some researchers have reported other GPU or FPGA-based hardware realizations of sophisticated frame-based ConvNets for recognition type of applications. They showed extraordinary performance figures, for both recognition

Table 2.2: Frame-based ConvNets for face detection

	Nasse [66]	NEC [67]	Yale/NYU [41, 42]
Hardware	GPU Nvidia GeForce 8800GT	FPGA Virtex-5	FPGA Virtex-6
Input image size	640x480	540x480	512x512
total neurons	4.9 10^6	4.4 10^6	2.4 10^6
total synapses	930 10^6	530 10^6	115 10^6
peak MAC/s	N/A	3.37 10^9	160 10^9
delay	209ms	160ms	6ms

rates and processing times. Table 2.2 summarizes recognition delays of three example systems which perform face recognition with ConvNets, implemented either with GPUs [66] or FPGA's [67], [41, 42], when using VGA-like size input images. Nasse's implementation on an Nvidia GeForce 8800GT GPU needs 209ms per input frame. NEC's system implemented on a Virtex-5 FPGA requires 160ms, while Yale/NYU's *Neuflow* system in a recent Virtex-6 FPGA can do a similar task in 6ms.

The main advantage of frame-based realizations is that the hardware can be time multiplexed by fetching intermediate data between the processor and the external memory, at the cost of slowing down speed performance. Time-multiplexing is not possible with event-driven hardware as each neuron holds its state at each instant. On the other hand, one main advantage of event-driven hardware is that, because of the "*pseudo-simultaneity*" property, processing delay is kept approximately constant as hardware scales up. Another advantage is that up-scaling is simple by simply assembling more modules through asynchronous interconnect AER buses. However, large scale event-driven ConvNets are still under development. In this Dissertation, a scalable implementation of these networks will be proposed to overcome this limitation. The infrastructure can be used as hardware platform to map any large scale event-driven convolution-based architecture.

2.9 Conclusion

In this Chapter we have presented the state of art large scale spiking neural network implementations by reviewing some architectures proposed in the literature from a system level perspective. Several inter-module communication schemes suitable to transmit spikes between neurons have been studied and compared in order to identify the design challenges which arise from the large scale implementation. On the other hand, we have discussed the characteristics of ConvNets as a scalable architecture for the hardware realization of neuromorphic vision systems. Frame and spike based ConvNets approaches have been described to show the potential of this kind of neural networks and highlight the differences between them.

The next Chapter will be devoted to the design of a spike driven network suitable for the inter-module communication of AER modules. A 2D mesh assembly of AER chips will be analyzed as a suitable method for the inter-module interconnection. This module assembly provides the designers of neuromorphic systems enough degree of programmability to fit random connectivity patterns into a fixed hardware infrastructure. At the same time, the system design must be scalable and expandable to consider a broad range of application scenarios. Moreover, a lightweight implementation of networking circuits is mandatory to reduce the area and power headroom introduced by the system connectivity.

3 Spiking Neural Networks Hardware Implementation

3.1 Introduction

In previous Chapters, the need to build large scale neuromorphic systems which can perform brain-like signal processing tasks in a much more efficient way than with traditional computers was justified. The state-of-the-art systems which have been proposed throughout the last years were presented, emphasizing on the design constraints that the large scale of integration imposes. The ConvNet paradigm was also introduced as a suitable framework to build scalable and efficient neural vision processing hardware architectures. Frame-based and spike-based convolution modules were described and compared in terms of their computational properties. From this point on, we will focus on spike-based implementation of large scale ConvNets by developing a hardware infrastructure where software networks can be directly mapped and take advantage of the spike-driven computation.

As computational power of neural networks increases when the number of neurons and synapses rises, an scalable and expandable solution would be desirable. At the same time, configuration and testability also become important issues to obtain a really useful hardware which can be used by a wide community of researchers. This Chapter will be devoted to define the system level design of large scale event-driven neural units composed by a huge amount of neurons which communicate spikes using AER events. Prior to explain our solution, we will further analyze the network architectures proposed by other groups, although their basic properties were exposed in Chapter 2.

In general, AER processing modules require at least one AER input port and one

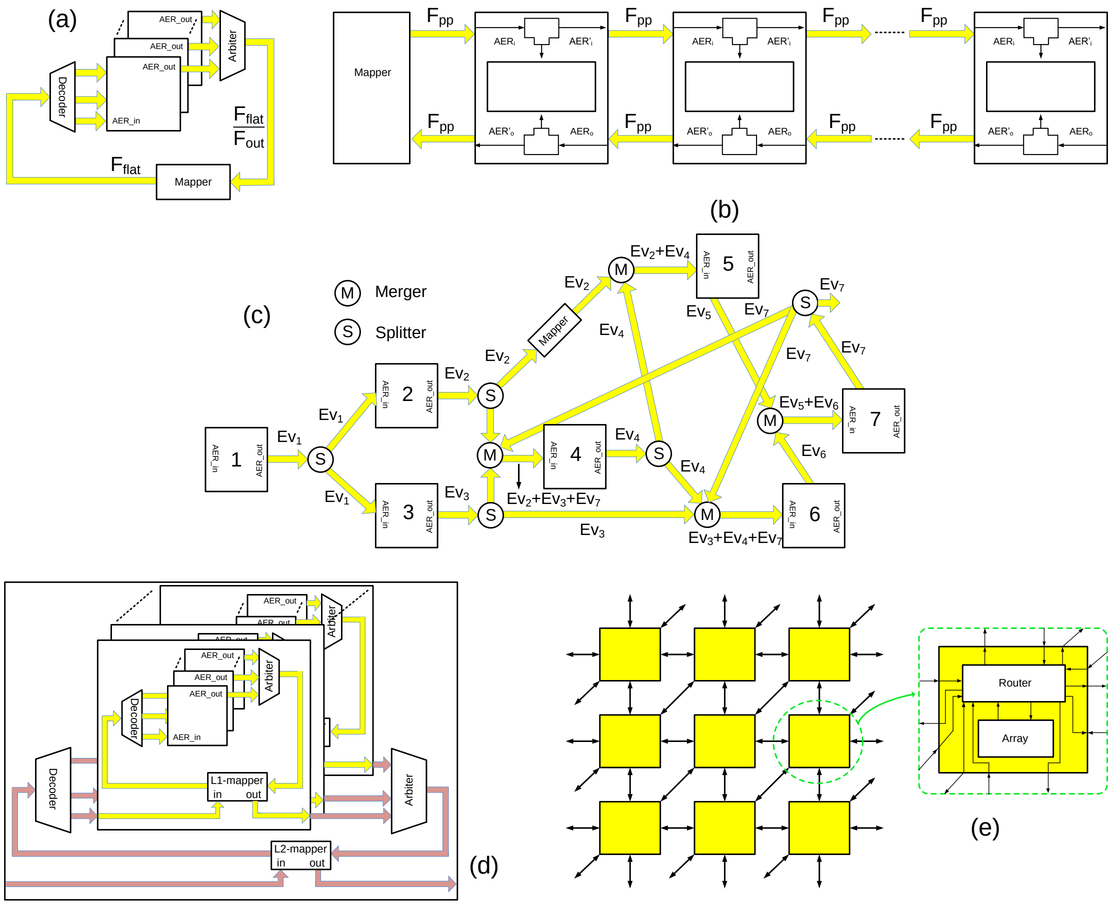


Figure 3.1: Illustration of different multi-module AER assembly options: (a) Flat-AER, (b) Router-Grid-AER, (c) Broadcast-Grid-AER, (d) Hierarchical-Fractal-AER, (e) Pre-Wired Structured-AER.

AER output port. As neuromorphic systems scale up in size, complexity, and functionality, researchers have been developing more complex and smarter AER "variations" to maintain efficiency, reconfigurability and reliability of the ever growing target systems they want to build. We can distinguish several solutions, summarized in Table 3.1 from a system level design point of view.

The simplest form of a generic AER concept, **"Flat-AER"**, for use in a large scale multi-module spiking neuromorphic system is illustrated in Fig. 3.1(a). Each module can contain, for example, an array of neurons. Each neuron is assigned a unique global address, which identifies the module it belongs to and its position inside the module. This way, the address space of all modules input and output AER ports is the same. All modules share a single external AER bus [15, 47, 73, 74]. The connectivity among neurons is configurable and set by a look-up-table in the external programmable "Mapper". Multi fan-out can be programmed in the mapper by repeating multiple destination addresses for each incoming address. Similarly, synaptic

Table 3.1: Multi-Module AER Addressing Schemes

		Flat AER [47]	Broadcast Grid AER [68–70]	Pre-wired Structured AER [52]	Hierarchical Fractal AER [48]	Router Grid AER [71]	Cross-Point Interc. [72]	Struc. Grid AER
Addressing	Address space	flat	flat	local	local	flat	local	flat
	Broadcast to all modules	Yes	Yes	No	No	No	No	No
	Local Routing Tables to define global network	No	No	No	Yes	Yes	Yes	Yes
	Single global mapper	Yes	Yes	No	No	No	No	No
Module Props.	isolated neurons	Yes	No	No	Yes	No	No	No
	neurons with synapses	No	Yes	Yes	No	Yes	Yes	Yes
	events with synaptic weighting	Yes	No	No	Yes	No	No	No
	projection fields with synaptic weighting	No	Yes	Yes	No	Yes	Yes	Yes
	physical synapses	No	Yes	No	Yes	No	Yes	No

weighting can also be implemented by programming destination address repetitions. However, event repetition (for either fan-out or weighting) severely penalizes the AER bus communication bandwidth. To overcome this, some reported neuron chips allow for an additional synaptic weight parameter together with the event address [47, 75], which could be programmed into the mapper as well.

Alternatively, some other reported neuron chips include a built-in mechanism to perform a given fan-out and synaptic weighting from a single input event (such as pre-wired diffusive networks [17, 18] or more elaborated computational hardware [20–22]). In this case, the mapper would only need to repeat an event if it is destined to neurons belonging to different modules. Normally, event addresses represent neurons. However, in some reported neuron chips which include a number of physical synapses per neuron [15], the input address represents one specific synaptic input. In this case, the address spaces at the mapper input and output would be different, as they represent different elements.

Flat-AER is simple and easy to build, configure, and use. It requires a mapper memory with as many positions as neurons in the system. However, the main limitation of flat-AER is communication bandwidth. Since every single event produced by any neuron has to travel through the single AER bus and Mapper, the maximum system total event traffic is limited by the bus bandwidth. If N_{tot} is the total number of neurons, f_n is the mean spike rate per neuron and F_{out} the average fan-out per neuron (spike repetitions introduced by the mapper to emulate the projection fields and/or synaptic weighting), the event arrival rate λ at the Mapper output channel is [76] $\lambda = N_{tot} f_n F_{out}$. If F_{flat} is the physical channel bandwidth, then the channel service rate is $\mu = F_{flat}$ and the average time an event waits to be serviced is (assuming an M/M/1 queue model [77]) $\bar{t_q} = 1/(\mu - \lambda)$. Consequently, the absolute maximum number of events per unit time Ev this approach can handle is obtained for $\lambda = F_{flat}$ and is

$$Ev_{max} = N_{tot} f_n \Big|_{max} = \frac{F_{flat}}{F_{out}} \tag{3.1}$$

or the maximum allowable number of neurons is

$$N_{tot\_max} = \frac{F_{flat}}{f_n F_{out}} \tag{3.2}$$

Note that, in general, $F_{out} = n_{RF} \times n_W$ (n_{RF} is the receptive field size and n_W is the synaptic weight dynamic range) can become significantly large. For example, if the receptive field has size of $n_{RF} = 11 \times 11$ neurons and weights can have integer values ranging from $n_W = 32$ to 1, then $F_{out} = 3872$.

Reported AER-bus bandwidths are presently below $100 Meps$ (mega events per second) for point-to-point links, although there are reported cases using high density channels and multiplexing techniques which achieve higher event rates [46, 73]. Therefore, *flat-AER* allows for a total communication bandwidth between about 10^8 eps and 10^4 eps, depending on fan-out.

Having several modules sharing the same physical lines degrades the speed proportionally to the number of modules [69]. This can be overcome by using *Broadcast-Grid-AER*.

"Broadcast-Grid-AER" uses multiple point-to-point AER buses [68–70] and Fig. 3.1(b) shows its corresponding 1-D version. Each neuron in each module has also a unique global flat address. Each module has an AER input event path and an AER output event path, each with an AER input port and an AER output port. AER input events received at the input **AERi** port are sent to the module neuron array but are also passed through to the next module, via output port **AERi'**. This way, input events "hop" from module to module through independent AER point-to-point links. Consequently, the speed at each AER link is optimum and events are copied more efficiently in a pipeline fashion. Output events generated in each module also "hop" from module to module until reaching the Mapper, through **AERo** and **AERo'** ports. In this scheme all input events coming from the Mapper are broadcast to all modules, and each module checks if it is destined to the local neural array [68, 69]. However, the overall network connectivity information is contained in the global mapper, and can be totally reconfigured by reprogramming the mapper (as in the *flat-AER* approach).

One main claim in this approach is that channel bandwidth of each point-to-point link F_{pp} is improved proportionally to the number of chips N_{ch} in the network (for the 1D case), with respect to the *flat-AER* case where N_{ch} chips share the same AER bus[1], as in Fig. 3.1(a). More precisely, for typical PCBs, the bandwidth of a point-to-point link is

$$F_{pp} \approx 2\,(N_{ch} - 1)\,F_{flat} \tag{3.3}$$

The network has now N_{ch} point-to-point links, which allows for a total communication bandwidth of $F_{pp} \times N_{ch}$. However, each event has to be copied to each link, so that maximum event rate is

$$Ev_{max} = N_{tot} f_n \Big|_{max} = \frac{F_{pp} N_{ch}}{F_{out} N_{ch}} = \frac{F_{pp}}{F_{out}} \tag{3.4}$$

Thus, the bandwidth improvement comes from improving F_{pp} with respect to F_{flat}, which is proportional to the number of chips [69].

Both *Flat-AER* and *Broadcast-AER-Grid* use a common global flat address space and allow, in principle, for any arbitrary interconnect topology. However, practical neuromorphic systems have a pre-established hierarchical structure, depending on the functionality they implement. This fact has been exploited by other researchers to assemble scalable multi-module systems through independent AER-links, where each link is a physical plugged-in point-to-point bus-wire [52]. This is illustrated in Fig. 3.1(c), where AER splitters (blocks labeled "S" in Fig. 3.1(c)) and mergers (blocks labeled "M" in Fig. 3.1(c)) are also used for branching or de-branching links. A splitter block receives one input AER channel and replicates the traffic for n different output channels, while a merger block multiplexes n input AER channels into a single output channel.

[1]For a more detailed explanation see [69]

In this **"Pre-Structured AER"** approach the address space is local to the neurons writing to or reading from an AER link. Optionally, local mappers can be inserted in a link to adapt address spaces from an output to an input (for example, to perform subsampling, address rotations, bit reallocations, etc.). In this approach no global Mapper is required, as the connectivity is pre-wired, and events do not need to travel over all links. The number of links scales with the number of modules, so that communication bandwidth saturation is much less likely to occur as systems scale up. However, system reconfiguration is painful as it has to be done manually by re-plugging bus-wires, splitters, mergers, mappers, and processing modules.

In this case each point-to-point channels receives events from only a small fraction of modules/chips. In general, we can define an effective number of independent channels M_{eff} as a fraction of the total number of chip modules $M_{eff} = \alpha N_{ch}$. Then the network maximum communication bandwidth would be

$$Ev_{max} = \alpha N_{ch} F_{pp} \tag{3.5}$$

Parameter F_{out} does not appear anywhere, since now we assume that projection fields and synaptic weighting are implemented inside each module. As an example, Fig. 3.1(c) has 7 modules, each generating an output event rate Ev_i. The distribution of splitters and mergers determine the potential bottlenecks as

$$\begin{gathered} Ev_{1max} \leq F_{pp} \ , \ (Ev_2 + Ev_4)_{max} \leq F_{pp} \\ (Ev_2 + Ev_3 + Ev_7)_{max} \leq F_{pp} \ , \ (Ev_5 + Ev_6)_{max} \leq F_{pp} \\ (Ev_3 + Ev_4 + Ev_7)_{max} \leq F_{pp} \end{gathered} \tag{3.6}$$

under these constraints, the absolute maximum capacity of this particular network is

$$\begin{gathered} Ev_{max} = \left(\sum Ev_i \right)_{max} = \\ = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} \right) F_{pp} = 4F_{pp} \end{gathered} \tag{3.7}$$

Thus, in this case $\alpha = M_{eff}/N_{ch} = 0.57$. Therefore, in this approach the optimum point-to-point bandwidth F_{pp} is further improved, proportionally to the number of chips (by a factor αN_{ch}).

Joshi et al. suggested recently a **"Hierarchical-Fractal-AER"** approach [48], illustrated in Fig. 3.1(d), which extends the basic Flat-AER concept of Fig. 3.1(a) in a hierarchical fashion. It exploits the assumption that nearby neurons are more heavily interconnected than more distant ones. Address space is expanded as events need to climb up in the hierarchy. This way, more intense local traffic is performed very fast in parallel at the numerous lower level modules, while longer range but sparser traffic needs to traverse levels of hierarchy and is slower. Neglecting the traffic at the higher hierarchies and if M_{L1} is the number of lowest level L_1 parallel sections and N_{L1} the number of chips per L_1 section, then the absolute maximum network bandwidth would be

$$Ev_{max} = M_{L1} \frac{F_{hier}}{F_{out}} \tag{3.8}$$

with $F_{pp} \approx 2\left(\left(N_{ch}/M_{L1}\right) - 1\right) F_{hier}$. This way, maximum network bandwidth can be expressed as

$$Ev_{max} = \frac{1}{2} \frac{M_{L1}}{N_{L1}} \frac{F_{pp}}{F_{out}} \tag{3.9}$$

Depending on the ratio M_{L1}/N_{L1} an important improvement with respect to *Flat-AER* can be achieved.

Another approach, illustrated in Fig. 3.1(e), is what we call here **"Router-Mesh-AER"** [71]. Here again neurons have a global flat address which identifies their module and their address within the module, but there is no external Mapper through which all events have to pass. Instead, the mapping table is contained within a "*Router*" in each module. The router also decides the ports through which an event is sent to reach its destination module. Therefore, events are not broadcast to all modules, but optimum "hop" paths are established in a multi-cast fashion. The main problem is that a very large mapping table needs to be programmed in each module. However, to simplify these tables, optimizations can be computed for each module router depending on the system topology, and default routing paths can be established for event addresses not listed in the tables [71]. This mesh approach has been traditionally used in NoC (Network on Chip) [78] topologies to assemble high performance multi-core processor systems for high performance computing applications [79–81].

Another approach being developed presently, but at wafer scale [72], exploits massive programmable cross-point interconnects to reconfigure the network topology. We have included this method also in Table 3.1 for comparison, although we are focusing more on multi-chip systems. Nonetheless, this wafer scale approach also includes off-wafer re-routing (and event re-timing) means for longer range interconnects [46].

In all of the previously listed methods which use a global flat address space, the event includes a module ID corresponding to the module where the event was generated. Let us call this "*Source-driven*" coding. However, it is also possible to label the event with the ID of the destination module instead. Let us call this "*Destination-driven*" coding. In this Chapter we will consider a type of Pre-Wired Structured-AER approach, but instead of having manually pluggable links, modules are arranged in a 2D-Grid while inter-module links are configured through in-module routers. We call it **"Structured-Grid-AER"**. The approach is similar to the "*Router-Grid-AER*" except that the module router tables only contain information of the module-to-module links, instead of the full inter-neuron connectivity. We will analyze both '*source-driven*' and '*destination-driven*' codings and will show experimental results on example vision processing systems implemented on Virtex-6 FPGA prototyping boards, based on Convolutional Neural Networks (ConvNets).

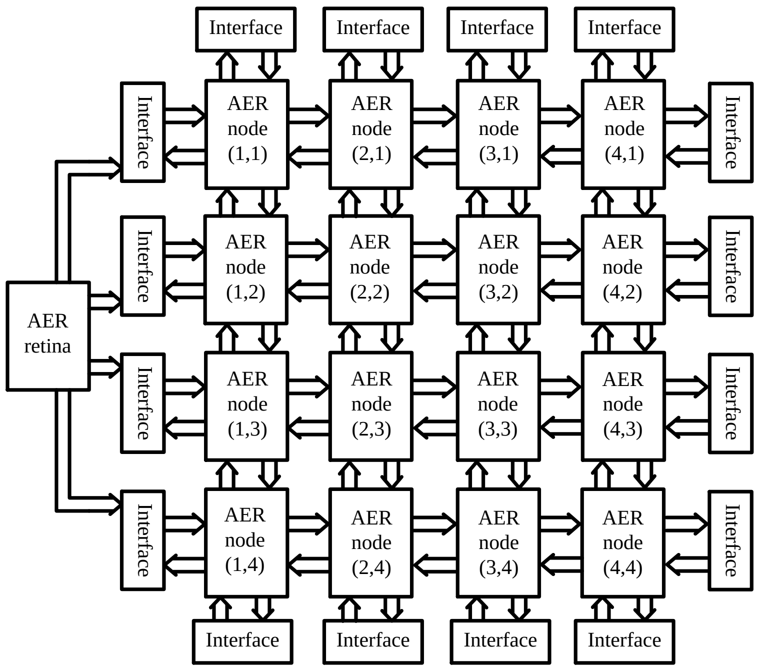


FIGURE 3.2: 2D Network Topology for Structured-Grid-AER. Each node is identified by the address field (*xNODE*, *yNODE*).

3.2 Routing in Structured-Grid-AER

Fig. 3.2 shows the 2D network topology we use for *Pre-Structured-Mesh-AER*. Modules communicate bidirectionally and orthogonally with their neighbors through point-to-point links. If the mesh has $M_1 \times M_2$ chip modules, the total number of inter-module links is $N_l = 2[(M_1 - 1)M_2 + (M_2 - 1)M_1] = 4M_1M_2 - 2(M_1 + M_2)$. Each event needs to hop through a number of links to travel from its source module to its destination module. Let's call n_h the average number of hops per travelling event. This average number n_h is application specific, but would usually be in the order of a fraction of M_1 and M_2. The network absolute maximum bandwidth is[2]

$$Ev_{max} = \frac{N_l}{n_h} \frac{F_{pp}}{F_{Mout}} \tag{3.10}$$

where F_{Mout} is the module fan-out (number of destination modules an event has to reach).

Each module is identified by a 2D index (*xNODE*, *yNODE*). From now on let us call each module in this 2D grid an *AER-node*. The internal structure of an *AER-node* is shown in Fig. 3.3. It contains a *Router*, a local *Event Processor* (or neuron/synapse

[2]By approximating $N_l \approx 4N_{ch}$ and $n_h \approx (2/3)\sqrt{N_{ch}}$, we can get a feeling on how maximum network bandwidth scales with the number of chip modules, since then $Ev_{max} \approx 6\sqrt{N_{ch}}F_{pp}/F_{Mout}$.

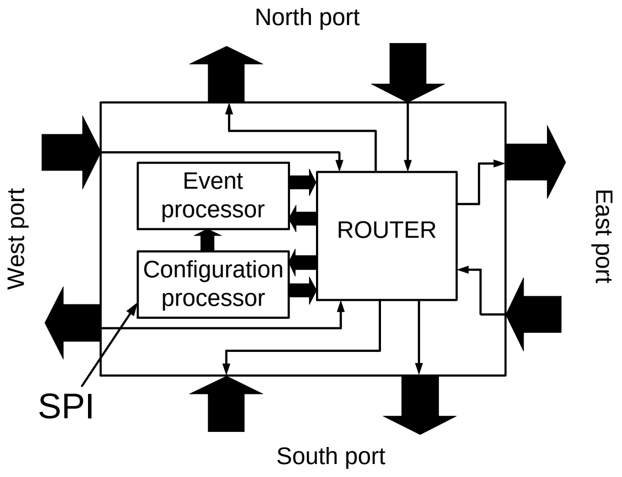


Figure 3.3: Example of AER node for a multi-node multi-link AER system.

array), and a *Configuration Processor* (to set configurable parameters in the *Event Processor* or *Router*). The *Router* receives external events from the four neighbors and based on its programmed routing tables decides whether to send them to the local processors or send them to other neighbors. For events generated by the internal *Event Processor*, the *Router* adds the corresponding node index and sends them through the programmed ports.

Thus, the *Router* introduces a network layer between the processing units logic layer and the physical layer implementation. A heading bit distinguishes between *configuration commands* to be handled by the *Configuration Processor*, and *data events* to be handled by the *Event Processor*. The *Configuration Processor* can also receive commands through an SPI (Serial Peripheral Interface) connection.

Another heading information identifies the node 2D index ($xNODE$, $yNODE$) coded in the event. This index identifies either the source node sending the event to the grid (in case of a *Source-Driven* addressing scheme), or the destination node to which send the event in the grid (in case of a *Destination-Driven* addressing scheme). Each addressing mode has pros and cons which are analyzed throughout this Chapter. For both cases, Fig. 3.4 shows the proposed 32-bit event format containing two fields:

- **Routing header:** the most significant bit is used to distinguish between data event (first bit is '0') or configuration command (first bit is '1'). The next 8 bits are used to code the destination or source node ID. Coordinates $xADD$ and $yADD$ are represented using 4 bits each one.
- **Upper layer data:** the remaining 23 bits contain the event/command data. If it is a configuration command, it contains a command description, for example, a checksum, a command identifier and command parameters.

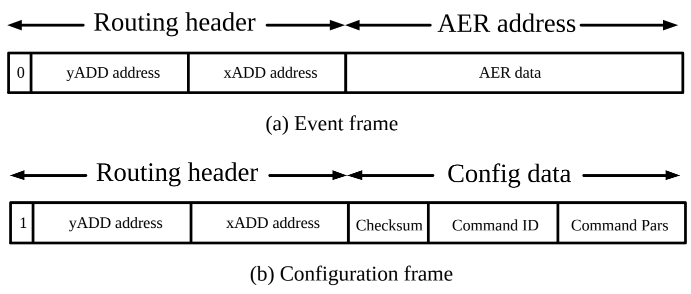


Figure 3.4: Events format with headers added for routing purposes.

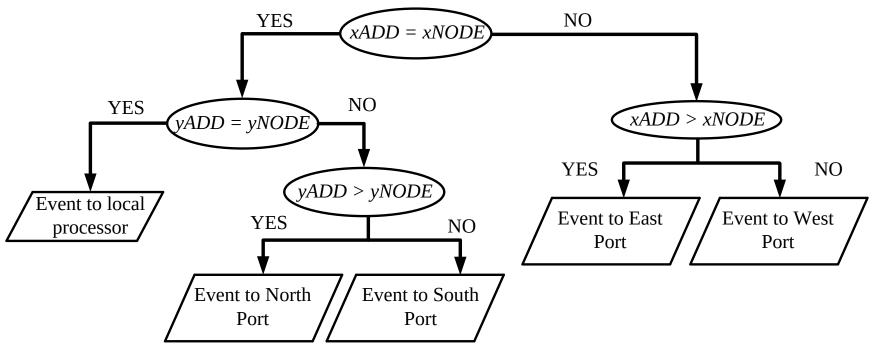


Figure 3.5: Destination-driven routing algorithm for handling incoming events. ($xNODE$, $yNODE$) is the local node address and ($xADD$, $yADD$) is the received destination address.

3.2.1 Destination-driven Routing Algorithm

In this algorithm, the destination node address is written in the routing header. When the event arrives to a network node, the router analyzes the addressing header and decides the output port to which forward the event. If the destination address corresponds to the node address, the event is sent to the local processor. If this is not the case, the event is routed based on the algorithm represented in Fig. 3.5.

The algorithm compares the event destination address $xADD$ and $yADD$ with the present node address $xNODE$ and $yNODE$ to decide the output port to which forward the event. Using the geographical information contained in the destination address, the event is routed to the neighbor node whose path to the destination is shorter in terms of the number of hops. As the router only requires a comparison between two 4-bit digital words, the hardware required can be very simple and the routing operation can be performed on the fly. The algorithm in Fig. 3.5 gives priority to $xADD$ and tends to concentrate the traffic in horizontal rows. To avoid this, routers which give priority to $yADD$ can be alternated with those priming $xADD$, balancing the situation..

Besides managing the traffic coming from the chip neighborhood, the router also

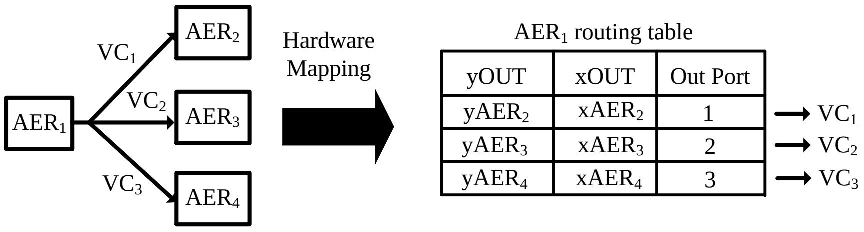


Figure 3.6: Routing table for cloning output events in the destination driven algorithm. Left: Example logic diagram (schematics) indicating the logic (virtual) connections between a source module AER_1 and three destination nodes AER_{2-4}. Right: Routing table that indicates the output event header to add ($xOUT$,$yOUT$) and the port through which to send it. VC_i (i=1,2,3) represents a virtual connection between the source node AER_1 and the destination nodes AER_{i-1}. ($xAER_i$,$yAER_i$) (i=2,3,4) is the network address of node AER_i.

inserts the headers for the new events created in the local processor. For each newly created event, the local router clones it as many times as destination nodes (or virtual connections VC_i) there are for this event. For each clone, or virtual connection VC_i, the router adds the destination node address and sends it to one of the local output ports. This is organized in a routing table which is consulted for every new event. Every entry of this table has an output destination address and an output port that must transmit the event. The routing table organization is illustrated in Fig. 3.6, for the case of one node AER_1 having three virtual connections $VC_1 - VC_3$ to three other nodes AER_2-AER_4. Fig. 3.7 shows the flow diagram of the output event management in the destination-driven routing algorithm.

3.2.2 Source-driven Routing Algorithm

In the source-driven option, the router receives information about the source address which generated the event. Hence, all nodes must store information about all the possible source addresses that they can receive. When a new event is received, the router searches for the source address in a local user configurable connection memory. This memory codes all the operations which should be performed when a source address is received: forwarding the event to one or more output ports, routing it to the local processor, or both at the same time. The input routing algorithm for the source-driven solution is shown in Fig. 3.8.

The user can program connectivity maps by setting the elements of the connection memory. Each node stores locally its connection memory with its own routing actions for all the possible source addresses. Each position of this memory corresponds with each of the possible 8 bit source addresses and stores a 5-bit digital word. When one of those bits is at high level, it indicates that the event must be transmitted through

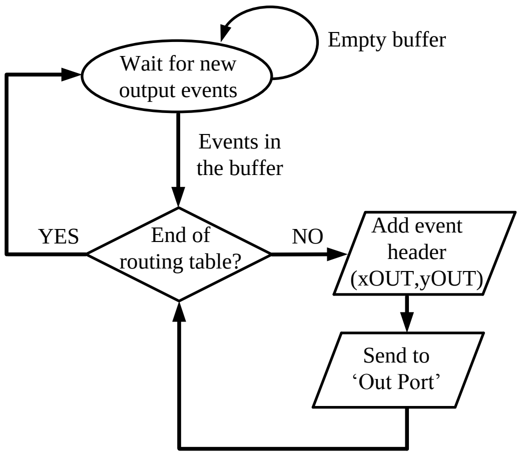


Figure 3.7: Output event management in the destination-driven routing algorithm. Values for “xOUT”, “yOUT” and “Out Port” are read from the routing table, as in the Fig. 3.6 example.

the interface associated with the bit position.

This feature allows replication of events at network nodes when it is necessary. This is done by activating several bits in the connection memory position of the received source address. The event will be transmitted through the selected output interfaces. The programmed tasks can be performed in a parallel way because they do not require any shared resource. All the output virtual connections coming out from the same chip can share the same route along the network and the event cloning implemented in the destination-driven solution is not needed.

The local processor output event stream management is greatly simplified in the source-driven solution. The source address is added to all events which must be transmitted through the network. The only configuration parameter is the output port or ports that transmit this information to get their final destination. Sending the same event through different output ports can be used by the user to balance the network traffic load and improves the overall latency by reducing the event rate on critical physical links which otherwise can get saturated.

3.2.3 Comparison between both algorithms

Any node interconnection map can be implemented using any of the proposed routing algorithms. However, each solution offers some advantages with respect to the other one when performing some kind of connectivity features. We will focus the attention over the impact of both algorithms on parameters, such as latency, network event traffic (number of events transmitted through the network) and hardware resources needed for the router implementation.

The main difference between both solutions arises when the logical network topology

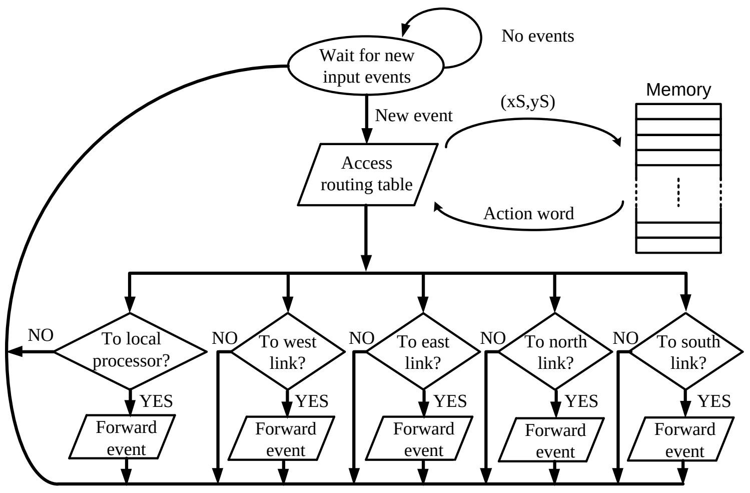


Figure 3.8: Source-driven routing algorithm.

requires that a node has to be connected with several destination nodes. This situation is illustrated in Fig. 3.6 where the output traffic generated by node AER1 must be received by several target nodes. The destination-driven solution replicates the output event for each output virtual connection. However, the source-driven solution permits to send a single event and replicate it in the network nodes. For this reason, this last solution makes a more efficient use of the network bandwidth resources. Moreover, it offers to the system designer more flexibility when implementing traffic balancing strategies.

In terms of hardware complexity, the source-driven implementation needs a more sophisticated routing algorithm. The increased complexity leads to longer delays in the router event processing, increasing the latency associated with the event transmission. The destination-driven router takes this decision on the fly only taking into account the node address and the information contained in the incoming event. The latency penalty caused by the source-driven router is strongly dependent on the shared connection memory implementation and its arbitration mechanism. This memory block is large and results in an important area overhead.

The source driven algorithm provides to the system designer more freedom to balance event traffic and design routes through the networks. For any source-to-destination route, the designer can insert detours, dedbranchings, and local event clonings at any intermediate node of route to balance and optimize overall traffic. On the other hand, the destination driven algorithm creates pre-determined routes along the network, and the designer can only change the output ports of the source module of a route. Also, for the destination-driven case, the events that have to reach several modules have to be cloned at the source module necessarily. However, in the source-driven case, multiple module destination events can be cloned at intermediate route points. This alleviates overall traffic and makes the average effective module fanout (F_{Mout} in eq. 3.10) smaller.

3.3 Router Design Details

The routing algorithms described above should be implemented efficiently and with a minimum hardware cost. This Section describes hardware implementations focusing on design issues which must be faced to reduce routing processing times, while keeping lightweight implementations.

3.3.1 Destination-Driven Router

Fig. 3.9 shows a block diagram of the destination-driven router circuit. The traffic through any of the input channels is processed by a *ROUTERIN* block which implements the lightweight destination-driven routing algorithm. A highly parallel hardware architecture has been chosen to reduce routing processing times. The goal is to separate the data paths of event streams which do not need to use the same channel to be forwarded. For example, a stream transferred from the west to the east port is never interfered by another one being transmitted from the north to the south port. This is only possible if shared routing resources are reduced to a minimum by replicating them in the architecture.

Routing is defined by user provided parameters. The first parameter is the node address which is used to identify the node in the network topology. The second parameter is the routing table containing the information to communicate with the node's target destinations. Every entry in this routing table is a 10 bit word, whose 8 less significant bits are used to code the destination address and the 2 most significant bits are used to specify the corresponding output ports (as in Fig. 3.6).

The basic building blocks of the destination-driven router architecture are:

- **ROUTERIN:** this block receives the input stream coming out from the input channel and implements the routing algorithm described in Fig. 3.5. It decides the output interface to which forward the event, by comparing the input event address (*xADD*, *yADD*) with the user-specified local node address (*xNODE*, *yNODE*). The AER handshaking protocol is also used to internally transfer the events between different blocks. Handshaking is used here for flow control purposes as the individual processor operation is stopped when a network communication link cannot transmit events. In these overflow situations, the router does not send the acknowledge back until there are free hardware resources to process the event. Each *ROUTERIN* block has four independent output interfaces connected to the output channel access arbiters or with the local processor arbiter.
- **ARBITER:** this block manages the access to the output channel for events coming from the *ROUTERIN* blocks or from the local processor. The *ARBITER* scans its four input AER interfaces to detect any new event. If an event is detected, it takes control of the output channel. If another event from any other

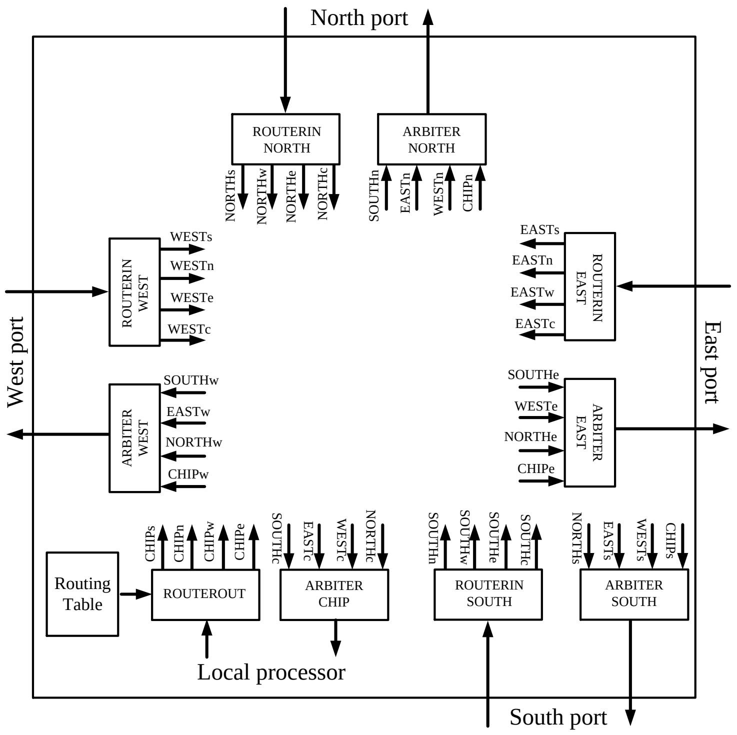


FIGURE 3.9: Destination-driven router block diagram.

interface arrives while the output channel is busy, the resource is not assigned until the current event releases the shared resource. If several input interfaces want to take control of the shared resource at the same time and the block is busy transmitting an event, the *ARBITER* gives lower priority to the last interface attended. Therefore, the arbitration mechanism avoids that a fast input interface takes full control of the shared resource. Note that all arbiters are synchronous circuits driven by the common FPGA clock.

- **ROUTEROUT:** this block implements the algorithm described in Fig. 3.7 to manage the events coming from the local processor. This block reads the routing table row by row to add the proper header to the event, forwarding it through the specified output interface. For this purpose, the *ROUTEROUT* block has four output interfaces connected to the north, south, east and west arbiters. To improve the router parallelism, the routing table is divided for every output port. This way, if an event has to be transmitted through different output ports, this can be performed in a parallel way.

3.3.2 Source-Driven Router

Fig. 3.10 shows the block diagram of the source-driven router. The basic building blocks are very similar to those described for the destination-driver router. However,

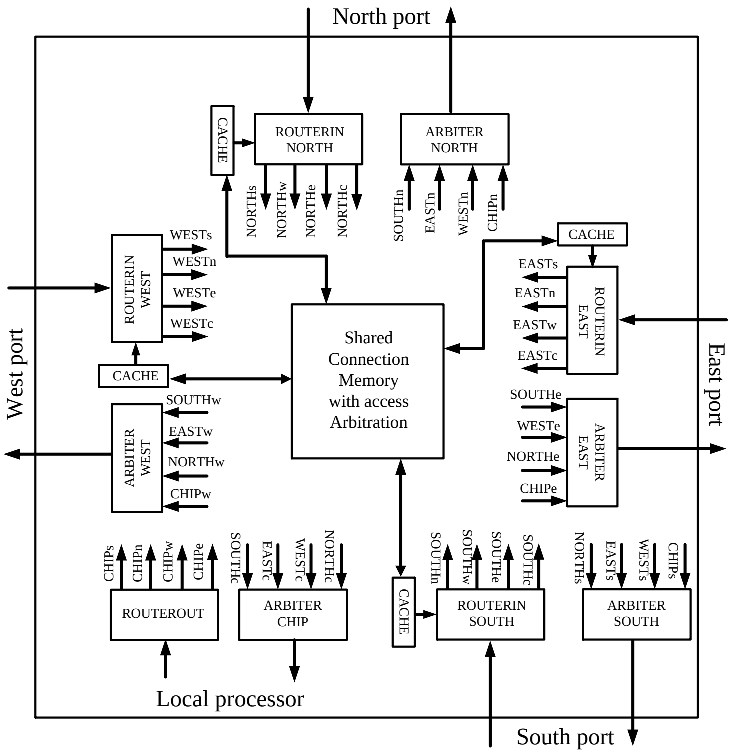


Figure 3.10: Source-driven router block diagram.

the *ROUTERIN* and *ROUTEROUT* blocks have very different internal structure due to the differences in the routing algorithm. In the source-driven solution, all blocks have to read a shared connection memory containing the routing information. Implementing an efficient shared access scheme to this memory can dramatically reduce the delay time associated with the routing event processing. The router contains the following blocks:

- **ROUTERIN**: this block analyzes the events coming from the input channel to extract the source address. A shared RAM connection memory of 256 positions containing the routing algorithm information is accessed using this address. The word read from the connection memory codes the output port(s) that the event must be routed to. When this task is done, the behavior of this block is exactly the same than in the destination-driven router.
- **Local Cache:** every event which is routed in the source-driven solution needs a memory access and all *ROUTERIN* blocks can read the memory at the same time. However, each input block will process a limited number of flows in a real network. Hence, the addresses that every *ROUTERIN* block is going to read will be repeated very often and it will be a reduced group of the complete space of addresses. To speed up this process, the *ROUTERIN* block reads first a dedicated cache memory which stores the most common accessed addresses content. When the routing algorithm requires to access a shared memory position, the

ROUTERIN block checks the cache memory and searches for this word. If it is found in the cache, the event is routed and there is no need to access the shared RAM. If the word is not found, the block reads the shared RAM and stores the word on its own cache.

- **ROUTEROUT** is greatly simplified because an event does not have to be replicated nor the same header has to be added to all events. In this case, every time an event coming from the local processor is detected, the router node address is added and the event sent to the corresponding output port(s).
- **Shared Connection Memory:** this block stores the configuration words which code the routing actions to be taken for each possible source address. All *ROUTERIN* blocks need to read this memory when they receive a source address that they have not stored previously in their local cache. An arbitration mechanism is needed to avoid conflicts when several *ROUTERIN* blocks need to access the shared memory. This mechanism ensures that only one *ROUTERIN* block has control over the address bus of the shared memory.

3.3.3 FPGA Implementations Comparison

In order to compare both router implementations in a multi-module system, we have analyzed the impact of realizing different size networks on a Virtex-6 FPGA prototyping system. We have used as unit-module a VHDL description of an event-driven programmable-kernel 2D AER-Convolution-processor for vision applications, capable of handling programmable kernels of size up to 11×11 on pixel arrays of size 64×64 [82]. The VHDL code of this convolution module is listed in Appendix I. The Virtex-6 could hold up to 64 of these Convolution modules, programmed with any arbitrary interconnection map. Fig. 3.11 illustrates the case of a 3×3 Convolution network. Inputs to the network are provided through 3 input ports, connected to an AER splitter which receives a unique external input AER flow and replicates it over the three inputs. The VHDL code generation to build the system in Fig. 3.11 has been automated with a set of Matlab functions which describe the system features depending on the network size which wants to be implemented. The Matlab code is shown in Appendix II.

Every network node includes one of the previously described routers, the convolution block, and a dedicated configuration processor with an SPI. This interface is fed by a global configuration controller that receives all the configuration data (like router tables, and convolution processor parameters) from a host computer. The rest of peripheral modules can connect one of their AER outputs to a multiplexer block that connects to the FPGA outside. This way, such peripheral modules can send its output to any of the multiplexer inputs to allow AER flow monitoring from the outside. Fig. 3.12 shows the FPGA occupation ratios for different network sizes (where the number of nodes is $N_n \times N_n$) in terms of occupied slices and memory resources. The destination-driven routing solution needs less hardware resources in terms of memory

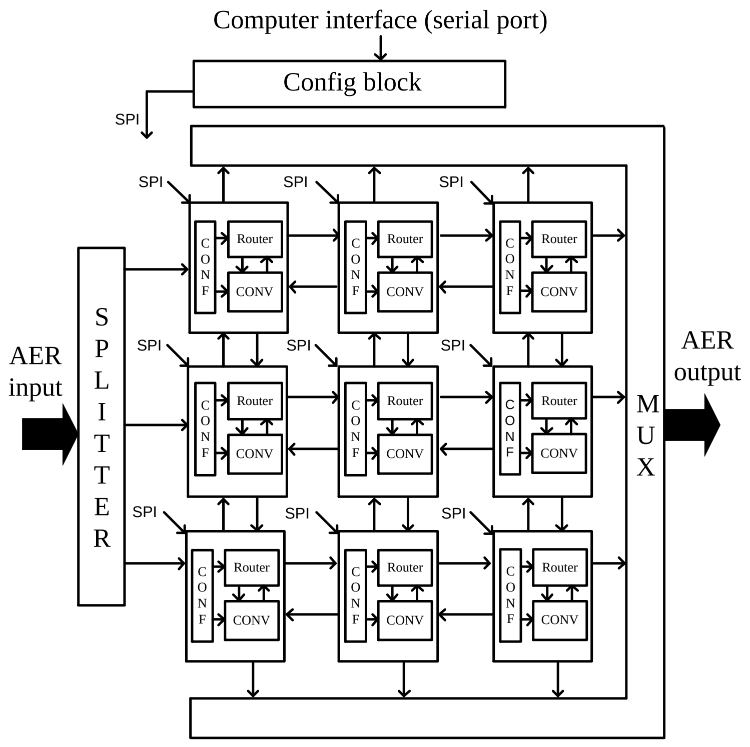


Figure 3.11: Block diagram for the network on chip implementation.

and slices than the source-driven implementation. Note that a 7×7 grid of these Convolution modules implements a neural network with $7 \times 7 \times 64 = 196k$ neurons [3] and an equivalent number of $196k \times 11 \times 11 = 24.3$ million synapses. Since each convolution module only has to store one 11×11 kernel of 8-bit words, the total physical RAM memory to store 7×7 kernels is 5929 bytes. The RAM required to hold all 8-bit neural states is 196KB. Consequently, what limits in this particular case is the number of slices available in the FPGA and not its memory.

3.4 Network Extension to Multiple FPGA

Examples illustrated in the previous Section were synthesized on a single Virtex-6 FPGA. To allow modular scalability to arbitrary size multi-module networks, provisions for multi-FPGAs (or multi-chips, in case of ASICs) need to be made. AER links inside the FPGA have been made using parallel AER buses. However, this is not realistic for a multi-FPGA realization because of the excessive number of resulting pins. Fortunately, high-end FPGAs include state-of-the-art serial links, like the Rocket I/O. In this Section we describe a way of extending each asynchronous bidirectional

[3] where k is 1024

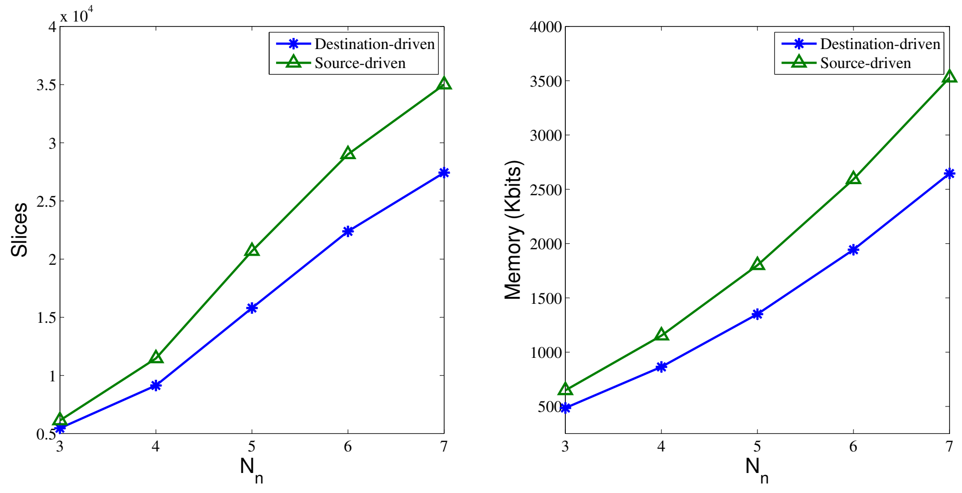


Figure 3.12: Slice and memory occupation in an $N_n \times N_n$ node ConvNet implementation. The LX240 Virtex-6 FPGA used in the experiments have 37680 slices and 14976 Kb of block RAM.

AER link to use the available Rocket I/O serial interfaces, without using dedicated handshaking lines (such as *Ack* and *Rqst*), nor an extra LVDS pair [83]. When the event rate transmitted in one direction exceeds the processing speed of the receiving module, a stop command is transmitted in the opposite direction. This way, flow control is implemented for both directions by just using the two required LVDS cable pairs for bidirectional serial transmission. For this purpose, we use the 8b/10b encoding scheme, as this allows for 12 special characters, commonly called K-characters. We use these characters to implement idles commas (to keep the link synchronized during absence of address events) as well as flow control commands.

Fig. 3.13 shows the full duplex serial Rocket I/O AER link with flow control capability. The Xilinx CORE Generator tool provides a wrapper to interface with the FPGA dedicated hardware. It defines the signals that the user must generate in order to send and receive data over this link. The data width interface, N_D in Fig. 3.13, is configurable by the user in 8, 16 or 32 bits. *TXp-TXn* and *RXp-RXn* represent the serial output interface and *REFCLKp* and *REFCLKn* the reference clock used by the Rocket I/O circuitry to generate the transmission frequency.

The *FRAME_GEN* block handshakes the input parallel AER stream and sends $8N_D$ bits at every rising edge of the master clock CLK provided by the Rocket I/O circuitry. If there are no user data that must be transmitted, the interface transmits a comma character represented by a K-character of the 8b/10b code. *FRAME_CAPT* receives the continuous data stream coming out from the channel to analyze it, discards the commas and frames the parallel AER events, implementing the handshaking with the next processing block. Signals *Ktx* and *Krx* are activated when a K-character is transmitted or received, respectively. The receiver also uses the information provided by signals *DISPrx* and *NTrx* to detect possible transmission errors. *DISPrx* indicates

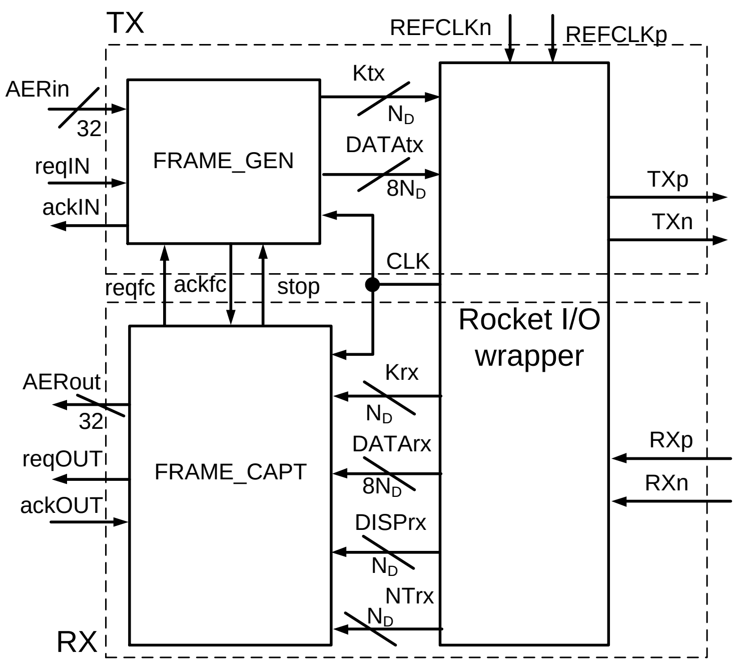


FIGURE 3.13: Full-duplex Rocket I/O based parallel-serial AER link with flow control capability.

a disparity error in the 8b/10b words received and *NTrx* is active when the received word is not a valid code character.

Fig. 3.14 illustrates the full-duplex flow control mechanism implemented through signals *reqfc*, *ackfc* and *stop* in Fig. 3.13. The figure illustrates the case of ROUTER1 sending events to ROUTER2. The ROUTER2 RX2 *FRAME_CAPT* block (see Fig. 3.13) writes each incoming event in a FIFO memory, which is read by the router when there are new events to be processed. If the ROUTER1 TX1 transmission event rate is faster than the ROUTER2 RX2 handling capabilities, the number of elements stored in the FIFO will increase. If ROUTER1 TX1 keeps sending new events, the FIFO would overflow and information would be lost. The *FRAME_CAPT* block detects when the number of elements is greater than a user-defined threshold N_{max} and sends a flow control message using a K-character using the ROUTER2 transmitting channel TX2. This message request is done by activating *reqfc* and it will be processed with the highest priority by the ROUTER2 TX2 *FRAME_GEN* block. When the flow control message is sent, the ROUTER2 TX2 *FRAME_GEN* block acknowledges the transmission using *ackfc*.

When the ROUTER1 RX1 *FRAME_CAPT* block detects the flow control special K-character, it automatically stops the event transmission asserting signal *stop*. In an overflow situation, the AER acknowledge signal *ackIN* is not activated even when the request signal is asserted and the AER data flow is stopped. The ROUTER2 RX2

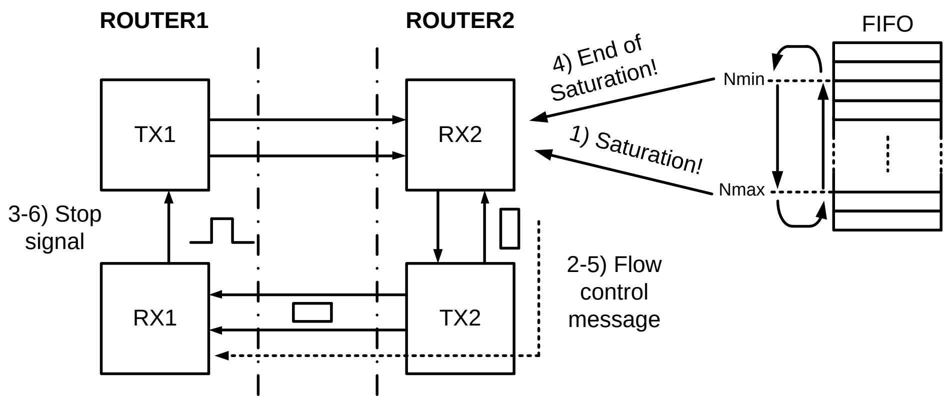


Figure 3.14: Flow control mechanism in full-duplex link.

FRAME_CAPT block monitors the receiving FIFO until the number of elements is below a second user-defined threshold N_{min}. From this point on, the overflow situation is considered to be finished and the flow control message is sent to ROUTER1 RX1. When it is received, *stop* signal is deasserted and the transmission flow starts over.

The ROUTER1 sender keeps transmitting events while the flow control mechanism is being carried out. To ensure that no events are lost during traffic peaks, the time needed to stop the transmitter when an overflow situation is detected must fulfill the next inequality:

$$T_{DET1} + T_{PROP} + T_{STOP} \leq (N_F - N_{max})\, T_{TX,EV} \tag{3.11}$$

where T_{DET1} is the time needed to detect the overflow situation and generate the flow control message, T_{PROP} is the channel propagation time and T_{STOP} is the time required to stop the transmitter. The maximum number of FIFO elements is represented by N_F and $T_{TX,EV}$ is the transmission event period that is causing the overflow.

It would be desirable that the transmission begins after a flow control pause as soon as possible. For this purpose, the flow control mechanism has to ensure that there will be stored events in the FIFO waiting to be processed by the router after the recovery. To maximize the receiver event rate, we can impose the following restriction on the N_{min} value:

$$T_{DET2} + T_{PROP} + T_{START} \leq N_{min} T_{RX,EV} \tag{3.12}$$

where T_{DET2} is the time needed to detect the end of an overflow situation, T_{START} is the time required to start the transmitter again and $T_{RX,EV}$ corresponds with the receiver event processing time (or inverse of event rate).

3.5 System level design considerations

Several analysis and optimization methodologies for 2D mesh connected networks have been proposed in the literature [84, 85]. The 2D communication layer is analyzed from a traffic management perspective using queuing theory to find network parameters such as latency, queue delays or queue occupation rates. All these parameters are strongly influenced by the application, because they depend on the network topology (physical and logical) and the traffic rates generated by the processors. On the other hand, optimization procedures for massively parallel architectures [86] have been successfully applied to neuromorphic systems which integrate millions of neurons [74]. Here we will rely on analysis techniques for 2D mesh networks, and use the results to suggest ways to optimize the implementations. The study will be centered in our convolution units network implementation, but it can be easily extended to other neuron arrays approaches. We will analyze two point of views: hardware resources requirements and even traffic.

3.5.1 Hardware resources requirements

The AER convolution modules and network circuits take a certain amount of resources (logic area and memory) which must fit within the selected implementation platform. This resource consumption will be related to the number of AER modules and the neuron array sizes. In general, an N_{units} AER system requires an area of

$$A_{total} = N_{units}\left(A_{conv} + A_{router}\right) + A_{prog} \tag{3.13}$$

Let every convolution module have a maximum kernel size of $N_{ker} \times N_{ker}$ and every weight be coded with W bytes. If convolution modules integrate an array of $N_{arr} \times N_{arr}$ neurons whose state is represented by S bytes, the area taken by a ConvModule is given by

$$A_{conv} = N_{arr}^2 S + N_{ker}^2 W + A_{logic} \tag{3.14}$$

where A_{logic} is the area taken by the logic of the convolution module implementation, which depends on N_{arr}, W and S.

For the routers, as we discussed previously, the simplicity of the destination driven algorithm needs less logic resources $A_{logic,dest}$ than the source driven $A_{logic,sour}$. On the other hand, the source driven approach needs extra memory to store the routing actions for all possible source addresses in the network (coded in 5 bits). If N_{add} bits are used to code the network addresses, the router areas can be expressed as

$$A_{router,dest} = A_{rlogic,dest} \tag{3.15}$$

$$A_{router,sour} = A_{rlogic,sour} + \frac{5 \times 2^{N_{add}}}{8} \tag{3.16}$$

The total number of neurons which can be integrated in the system is $N_{units}N_{arr}^2$ and the maximum number of kernel weights is $N_{units}N_{ker}^2$. Taking into consideration

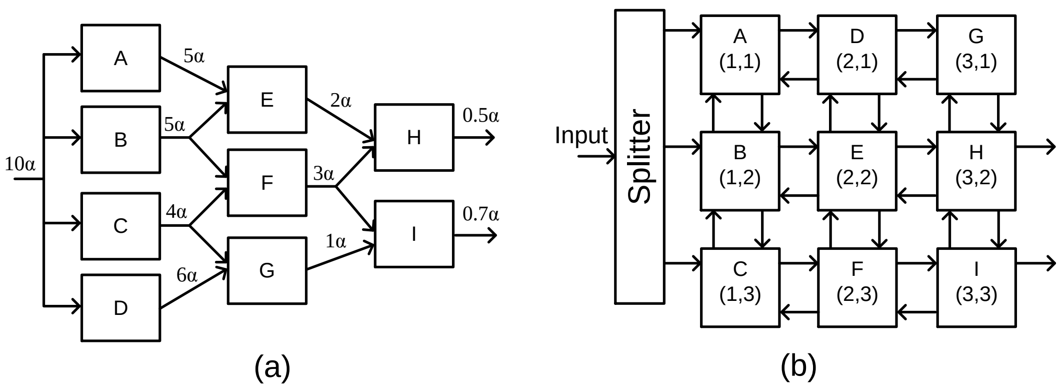


Figure 3.15: Example AER system to study the system level characterization methodology. (a) Logical description of the network and event rates for each logical (virtual) channel. (b) Physical implementation on the structured-grid-AER infrastructure. Parameter α is a scaling factor to study the network using different traffic loads. Mapping of the logical modules (A to I) on the physical 2D grid.

the previous resource analysis, the total RAM memory M needed for both routing algorithms can be written as

$$M_{dest} = N_{units}\left(N_{ker}^2 W + N_{array}^2 S\right) \tag{3.17}$$

$$M_{sour} = N_{units}\left(N_{ker}^2 W + N_{array}^2 S + \frac{5 \times 2^{N_{add}}}{8}\right) \tag{3.18}$$

3.5.2 Event traffic estimation

One of the most important parameters of any AER communication scheme is the event transmission latency between processing blocks. The AER grid architecture used in this Chapter can be studied using an analytical model for NoC performance analysis [84], where routers are modeled as a collection of FIFO buffers with five input/output channels (north, south, east, west and local interfaces). This model computes the network queues occupation at every interface of all the routers assuming that event rates for every network channel are known parameters. These rates are strongly dependent on the specific application and can be easily estimated through a behavioral level simulation.

For example, Fig. 3.15(a) shows the logic network topology of a specific pre-structured AER system. This network can be simulated behaviorally to obtain the average event rates at each connection (virtual channel). To map the logic network onto the physical 2D mesh of nodes, the first step is the “placement” of modules, which is illustrated in Fig. 3.15(b). The second step is to assign a route (or list of nodes) for events going from a source node s to a destination node d. Let's call this list of route nodes Π_{sd}. Each route corresponds to a virtual connection in he logic network. Once the module placement and route lists Π_{sd} are established we know the event rates at the input and output router channels.

Let l_{ijr} be the event rate at input channel i routed to output channel j in router r. For every router, we can define a 5×5 forwarding probability matrix where element f_{ijr} corresponds to the probability that an event which arrives at interface i leaves the router through interface j. These probabilities can be computed for every router in the network as

$$f_{ijr} = \frac{l_{ijr}}{\lambda_{rj}} \qquad i,j \in [1,5]\,, \quad r \in [1, N_{unit}] \tag{3.19}$$

$$\lambda_{rj} = \sum_{k=1}^{5} l_{ikr} \tag{3.20}$$

In the network the events from different flows have to share common resources to get their final destination. If two events want to use the same resource, arbiters grant the access and make some events wait in their queues while the resource becomes available. Using the forwarding matrix, we can compute the contention probabilities c_{ijr} for every router, i.e., the probability that channels i and j compete for the same output, as:

$$c_{ijr} = \sum_{k=1}^{5} f_{ikr} f_{jkr} \quad \forall i \neq j \qquad c_{ij} = 1 \quad \forall i = j \tag{3.21}$$

The router forwarding matrix $F_r = [f_{rij}]_{5\times5}$ and the contention matrix $C_r = [c_{ijr}]_{5\times5}$ describe the routers traffic management. It can be demonstrated [84] that the average number of events per queue $N_r = [N_{rj}]_{5\times1}$ at every router can be computed as:

$$N_r = \left(I - t_r \Lambda_r C_r\right)^{-1} \Lambda_r \bar{R}_r \tag{3.22}$$

where scalar t_r is the mean event processing time in the router and $\Lambda_r = [\lambda_{rj}]_{j=r}$ is a diagonal matrix whose elements are the total event rates through the 5 input interfaces. $[\bar{R}_r]_{5\times1}$ is the residual time matrix, which represents the amount of time that a new event has to wait in queue until the event which occupied the shared resource at the moment of the new event arrival finishes its processing. Solving eq. (3.22) and applying Little theorem [87], we can compute the mean waiting time in channel j of router r as $W_{rj} = N_{rj}/\lambda_{rj}$. This way, the total latency of one node-to-node hop is $N_{rj} + t_r + t_{tx}$, where $t_t x$ is the transmission time through the inter-node physical channel. The total latency of an event traveling from source node s to destination node d is therefore,

$$L_{sd} = \sum_{(r,j)\in\Pi_{sd}} \left(W_{rj} + t_r + t_{tx}\right) \tag{3.23}$$

This analysis methodology will be applied in a subsequent Section to an example system where the network traffic will be estimated for a source driven and a destination driven solution. Moreover, we will discuss how to use this analysis procedure to improve the network performance by varying some of the implementation parameters. Note that, given a logical network together with virtual connections event rates, one only needs to establish a node "placement" and the route lists $\{\Pi_{sd}\}$. The rest of computations (from eqs. (3.21) to (3.23) follow in a straight forward way, given parameters t_r, Λ_r, $\bar{R}_r$ and t_{tx}. The result is the route delays $\{L_{sd}\}$ from which one can identify the main timing bottleneck as its maximum:

$$L_{max} = max\,\{L_{sd}\} \tag{3.24}$$

The designer must now adapt the node "placement" and route lists $\{\Pi_{sd}\}$ to minimize L_{max}.

3.5.3 Example of use

Once we have theoretically described the network traffic analysis methodology, we will briefly illustrate the system level analysis methodology presented in Section 3.5 with the simple example shown in Fig. 3.15. This example is not optimized to achieve high performance, but it only tries to show how we can analyze the network using queuing theory and how this can help us in taking design decisions. Fig. 3.15(a) shows the logical connections (virtual channels) between blocks (nodes) and the event rate at each channel. Channel event rate has been scaled by term α to study traffic under different load conditions. Traffic information can be obtained from behavioral simulations. The logical topology can be mapped into the physical system as depicted in Fig. 3.15(b). The routing algorithm and tables determine the event routes through the network and we can use this information to estimate the total event rate through each physical channel. Using this information, the forwarding F_r and contention C_r matrices can be computed for each router using eqs. (3.21) and (3.21). The model is fed with all these matrices and with some implementation parameters, such as the routers service time t_r or the physical channel transmission times t_{tx}. Knowing all these parameters, we can solve the traffic equation (eq. (3.22)) to estimate the mean number of packets at every network queue N_r. This number provides information about the network traffic distribution and allows to compute the queues mean waiting time W_{rj}. By applying eq. (3.23) to every route we obtain the latency associated with every route L_{sd}, as well as the mean and worst case latency for the whole network.

Fig. 3.16(a) shows the resulting mean network latency versus scaling factor α for the example in Fig. 3.15. For low event rates, routers are fast enough to process events on the fly and queues are empty most of the time. As a result, the mean latency is constant and depends only on the mean number of hops and routing times. In this situation, and for this particular example, the destination driven algorithm presents lower mean latency than the source driven one because the routing algorithm is faster. The mean latency starts to increase exponentially with acceleration factors $\alpha \approx 8 \times 10^5$ because queues must handle more events and their waiting times rise.

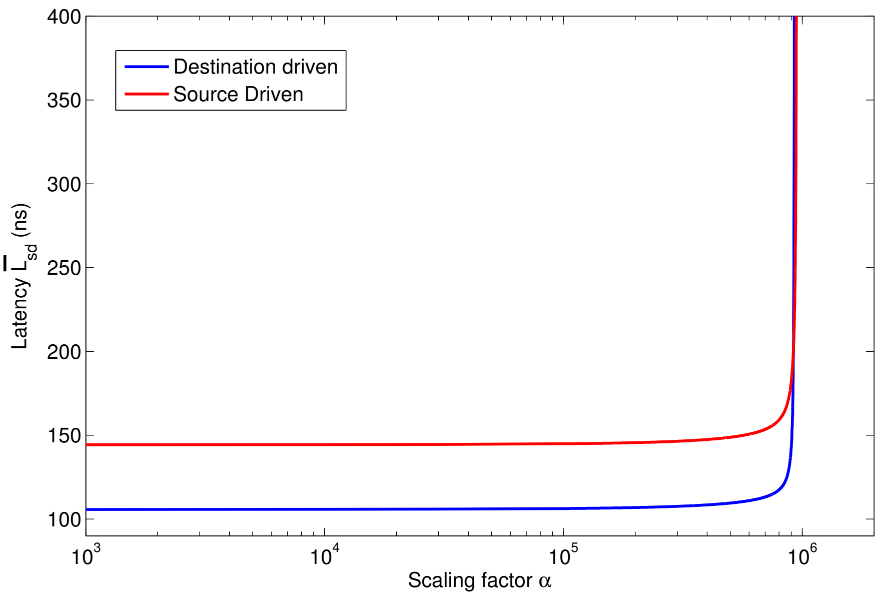
For this particular system, the saturation point is almost the same for both routing algorithms. However, Fig. 3.16(b) shows the same simulation for a 3×3 array of Gabor filters where the input flow must be forwarded to all the nodes in the network. In this case, the network saturates at $\alpha \approx 5.8 \times 10^5$ for the destination driven algorithm, while the source driven one saturates at $\alpha \approx 7.4 \times 10^5$. In this case, the source driven routing is more efficient for very high traffic. This is because in the destination driven case each input event to the network has to be replicated once per destination module, while in the source driven case each node can do local replication. In the source driven case the number of actual events traveling over the physical links is therefore much less for this particular application.

For the example in Fig. 3.15 the number of queued events in all routers when the network saturates at $\alpha = 9 \times 10^5$ can be obtained by solving eq. (3.22). Fig. 3.17 represents, for each router, the 5 input interface queues. This allows to find the network bottlenecks for every case. For the destination driven case, the west interface of router A (RA in Fig. 3.17(a)) is the most loaded queue. As input events have to be replicated to reach nodes A and D, the input event rate at RA west input interface is artificially increased, overloading this channel. For the source driven algorithm represented in Fig. 3.17(b), the west interface of routers RA and RD represent the system bottleneck. Again, these two channels are the most overloaded due to the multiplexing of several AER streams.

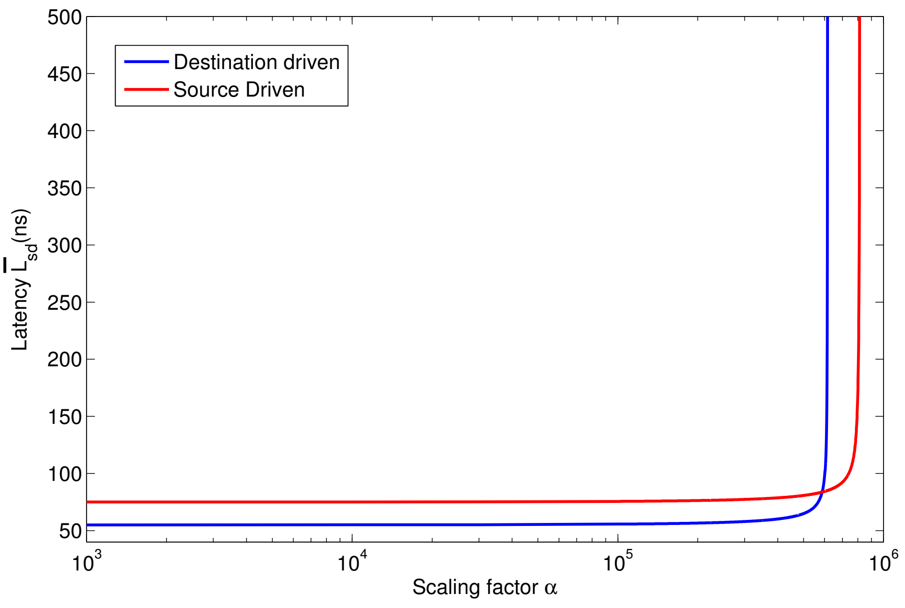
Fig. 3.18-(a) is an example of how the traffic analysis methodology can be used to explore the network parameters design space. We have repeated the traffic simulations varying the routing time t_r for the destination driven case (same conclusions apply to the source driven case). For longer service times, the mean latency increases because each hop takes a longer time to route events. Moreover, the network becomes saturated for lower event rates if the routers service time t_{tx} is increased reducing the maximum achievable event rate. Fig. 3.18-(b) performs the same simulation varying now the transmission time through the channel. In this case, the saturation point remains at the same point in all simulations, but the overall latency increases for low event rates.

3.6 Experimental Results

In this Section we provide experimental results by implementing the previous concepts on Virtex-6 hardware using Xilinx ML-605 development boards. First we show characterization results of the Full-Duplex Rocket-I/O-Based AER parallel-serial interface described in the previous section. Afterwards, an example multi-module AER processing system is described, implemented on a single FPGA, consisting of an array of Gabor filters. Then, a second system, implementing a multi-layer feed-forward Convolutional Neural Network on a single FPGA, is described. Afterwards, we check the maximum capacity of a single FPGA, and to finalize we provide results on a multi-layer feed-forward ConvNet for character recognition.



(a)



(b)

Figure 3.16: Mean network latency for different event rate scaling factors for the destination and source-driven routing algorithms (a) for the example system in Fig. 3.15 and (b) for a 3×3 array of Gabor filters.

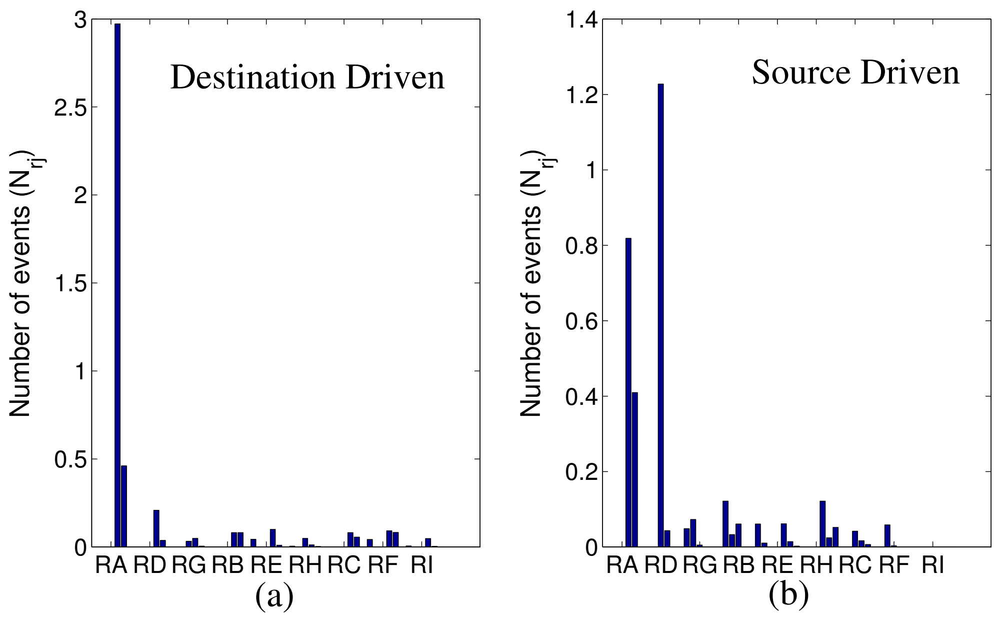
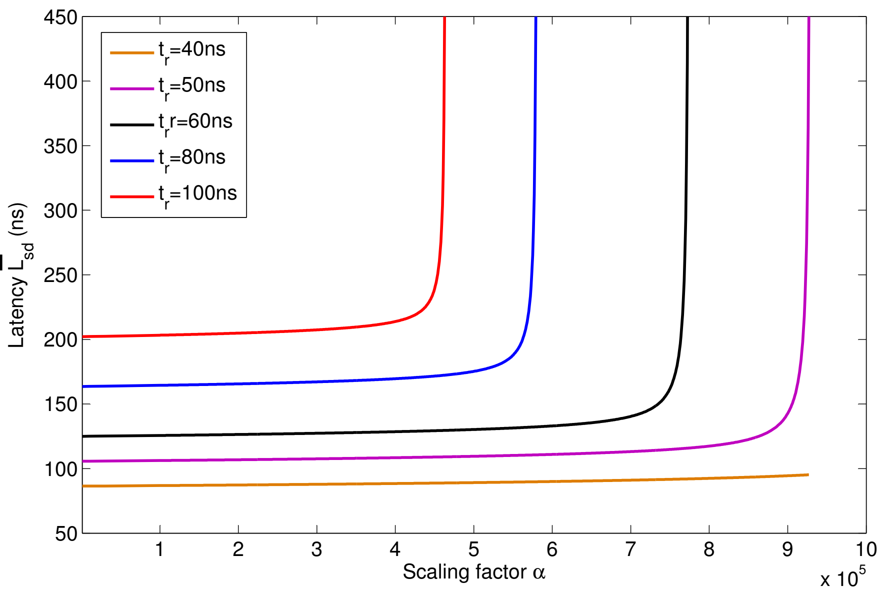


Figure 3.17: Events in router queues for an acceleration factor $\alpha = 9 \times 10^5$. Vertical bars indicate number of events in the order north-east-south-west-local interfaces for every router R_i.

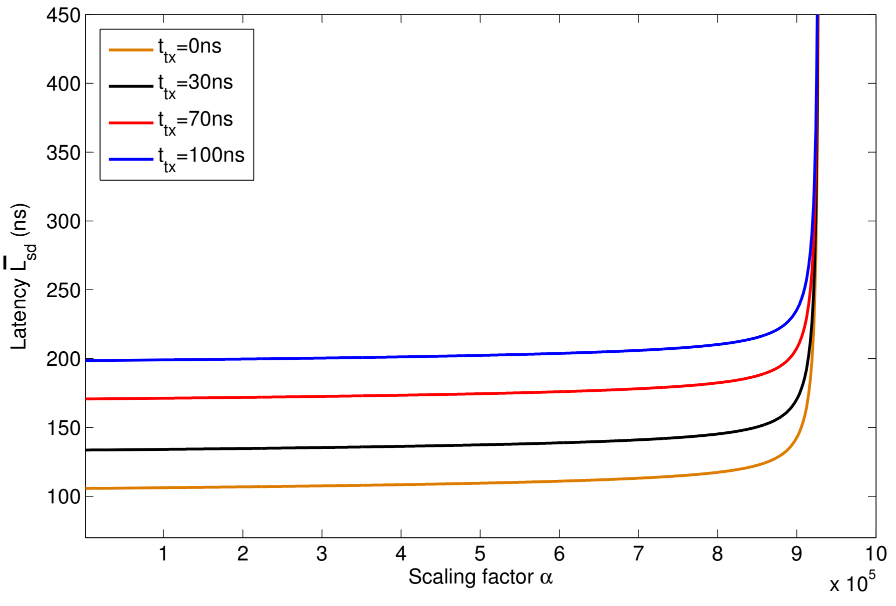
3.6.1 Full-Duplex Rocket-I/O-Based Parallel-Serial AER Interface

The ML-605 development board provides twenty independent full-duplex Rocket I/O serial ports, eight of which are available through an 8x PCIe connector. We used a dedicated board to adapt this connector to 16 independent SMA pairs, thus allowing not only to independently test and characterize several transmitters and receivers, but also to interconnect them. Some extra test circuits were added inside the FPGA, such as an event generator and an event consumer/analyzer, both with independent programmable event rate. This allows to force overflow situations and test the flow control dynamics, while detecting errors between the sent and received events.

The timing characteristics of the serial link are given by its latency and maximum event rate. To characterize latency, two independent Full-Duplex AER serial links are interconnected (their two LVDS differential pairs), and the delay between '*reqIN*' (see Fig. 3.13) of the first one to '*reqOUT*' of the second one is measured. This latency includes the delay introduced by the 8b/10b encoders and decoders, phase alignment buffers, comma detection circuits, ... To measure this latency, a very low event rate is programmed, so that consecutive events are sufficiently spaced in time. The measured latency was 232ns for a 2.5Gps bit rate, as shown in Fig. 3.19. The maximum event rate supported by the Rocket I/O can be characterized by analyzing the input AER handshaking (*reqIN*, *ackIN*) cycle duration which has been highlighted in Fig. 3.19 as 20ns. This results in 50Meps (mega events per second) maximum event rate for 32-bit events.



(a)



(b)

Figure 3.18: (a) Latency curves for the destination-driven solution for different routers service times. (b) Latency curves for the destination-driven solution for different transmission times Ttx.

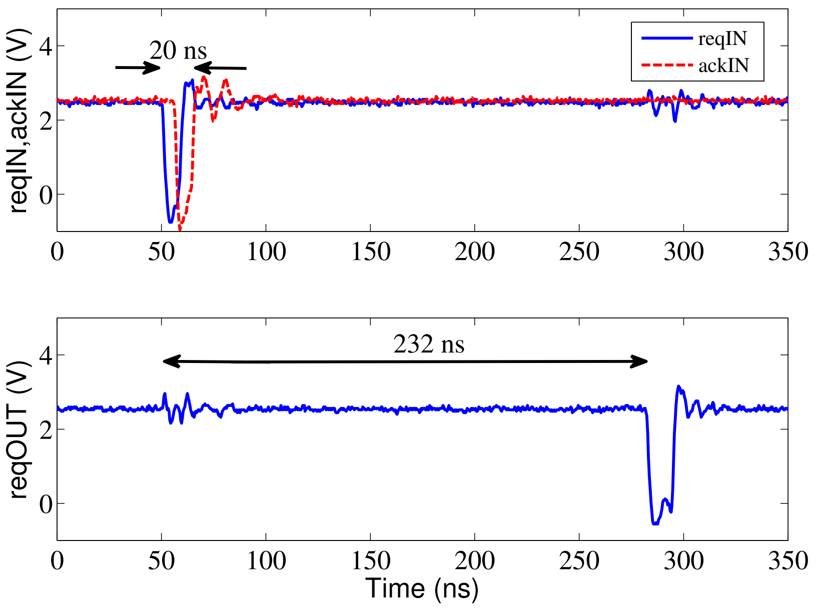
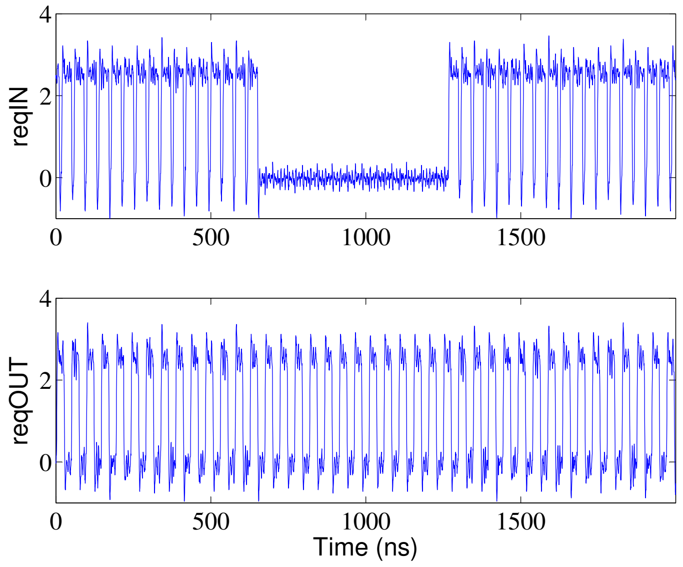


FIGURE 3.19: Interface input (reqIN,ackIN) and output request (reqOUT) when the link operates with very sparse events to measure the input to output event latency.

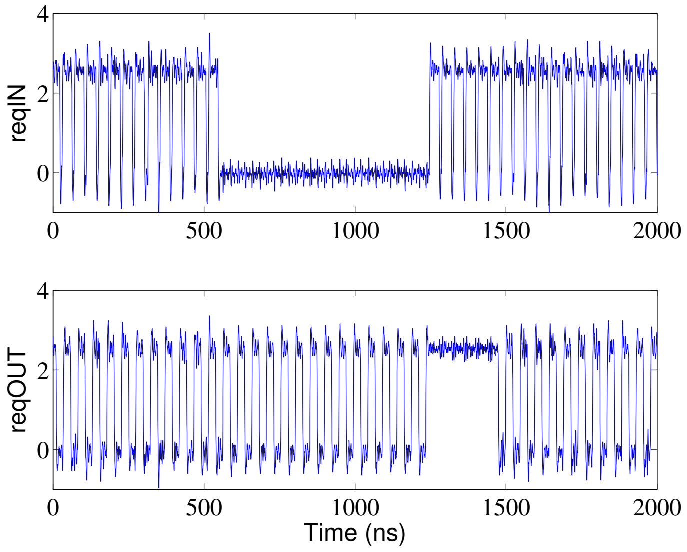
Fig. 3.20 illustrates the flow control operation. The event generator in the transmitter event rate is set intentionally to 26Meps, while the event consumer in the receiver is set to have an event consumption rate of 20Meps. This leads to an overflow in the receiving FIFO. It can be seen how the transmitter is stopped when the flow control message is received and started over when the overflow situation is overcome. Two different overflow behaviors are possible depending on the chosen N_{max} or N_{min} values. If they are optimally programmed, there is no stop in the output event flow, as shown in Fig. 3.20(a). On the other hand, pauses can appear if the receiver processes all events stored in the FIFO before the flow control message arrives to the transmitter. This situation is illustrated in Fig. 3.20(b).

3.6.2 Routers with Parallel-Serial Interfaces

The routers were complemented with four Full-Duplex Rocket-I/O-based AER parallel-serial interfaces (see Fig. 3.13), to test the performance for multi-FPGA event routing. Table 3.2 shows the Virtex-6 occupation statistics associated with both implementations. The total number of slices, memory blocks and Rocket I/O transceivers for the Virtex-6 FPGA has also been included to give an idea of the occupation ratio taken by the routers. For the destination-driven router, clock frequency could be set up to 250MHz resulting in 2.5Gbps serial bit rate. However, for the source-driven router, clock frequency could only be set up to 200MHz because of the higher complexity, resulting in a 2Gbps bit rate. The latency introduced by the routers can be measured by looking at the delay between the *reqOUT* signal of the input full-duplex Rocket-I/O serial-to-parallel interface to the *reqIN* signal of the output full-duplex Rocket-I/O parallel-to-serial interface. This latency was 12.5ns for the destination-driven router and 20ns for the source-driven router, in case of non-overflow situations. The source-driven case corresponds also to the case where the received address is in local cache. Otherwise, the latency becomes 30ns.



(a)



(b)

Figure 3.20: Interface input (reqIN) and output request (reqOUT) in overflow with (a) optimum N_{max}-N_{min} election and (b) non-optimum N_{max}-N_{min} election that leads to pauses in the output flow

TABLE 3.2: Router implementation statistics for the DD (destination-driven) and the SD (source-driven) routers.

Resources	DD router	SD router	Total FPGA
Occupied slices	1121	1400	37680
Occupied RAMB18E1 blocks	0	1	832
Rocket I/O transceivers	4	4	20

The destination-driven router can handle a maximum 32-bit event rate of 27Meps, which corresponds to 37ns for handshaking cycle completion. On the other hand, the source-driven router can handle up to 17.5Meps maximum event rate, or 57ns handshaking cycle. Although event transmission in destination-driven routing is faster, it is also true that events have to be transmitted multiple times when destinations are multiple. In this case, if the multiple events are routed through the same port, there will be an important delay penalty as shown in Fig. 3.21(a). However, it is also possible most of the times to replicate the events through (up to four) different ports, as shown in Fig. 3.21(b), to avoid this penalty.

3.6.3 Single-FPGA Implementation of Gabor Filter Array

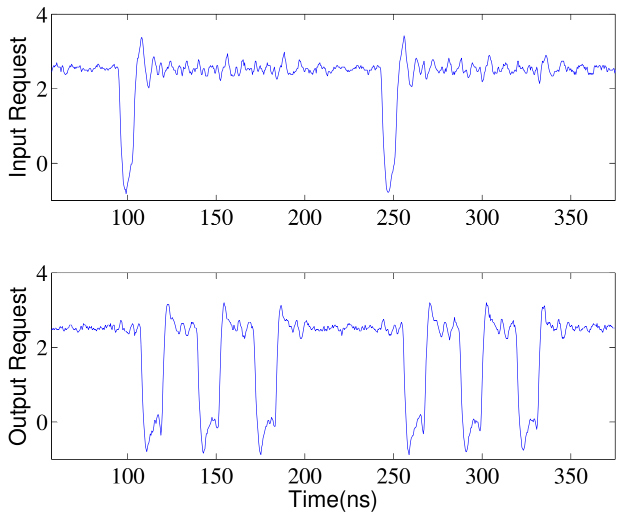
As a first illustration of multi-module operation we have implemented on a single Virtex-6 a 3×3 array of orientation extraction 2D-Gabor filters of different scales and angles, whose kernels are shown in Fig. 3.22. The implemented structure follows the diagram in Fig. 3.11 and receives sensory input from an AER DVS retina [11, 88]. All filter outputs are routed to one of the multiplexer inputs and captured off-chip with an AER data logger [89]. Fig. 3.23 shows the sensor and the nine filter output events collected during the same 160ms, when the retina is observing two walking people.

Event-driven convolution processors present the "*pseudo-simultaneity*" property [22]: the input flow of events (representing the input scene) is simultaneous to the output flow of events (representing the filtered input scene). This is because events are processed as they arrive with delays shorter than the average inter-event time. For the convolution modules we are using [82], event processing time for 11×11 kernels is about $3\mu s$. The input event flow provided by a 128×128 pixel DVS retina when observing people walking is in the range of 10-50keps (kilo events per second). Consequently, a Gabor filter output event representing an angle at a given scale will be available as soon as sufficient input events representing this feature are received plus the extra $3\mu s$ for processing the last one. This is illustrated in Fig. 3.24, where stars represent input events and circles output events. As can be seen, the output flow overlaps in time with the input flow.

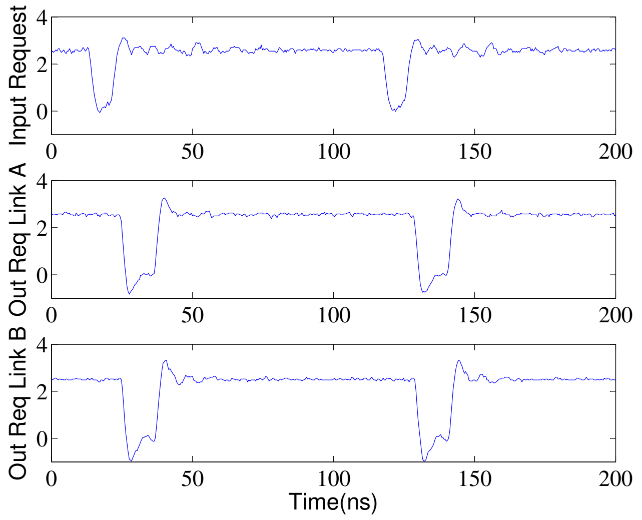
3.6.4 Multi-FPGA Implementation of Gabor Filter Array

In order to illustrate the use and operation of the Full-Duplex Rocket-I/O-Based Parallel-Serial Interfaces described in Section V.A, we implemented a 3×6 array of Gabor filters in 2 FPGAs. The corresponding diagram is shown in Fig. 3.25.

The retina events are fed through port 'AER input' and an in-FPGA splitter replicates them on three rows. Node routers are programmed so that these retina events



(a)



(b)

FIGURE 3.21: (a) Output event replica when the same event must be transmitted to different destinations using the same output port. (b) Parallel transmission of events coming from the local processor that must be transmitted through different output ports A and B.

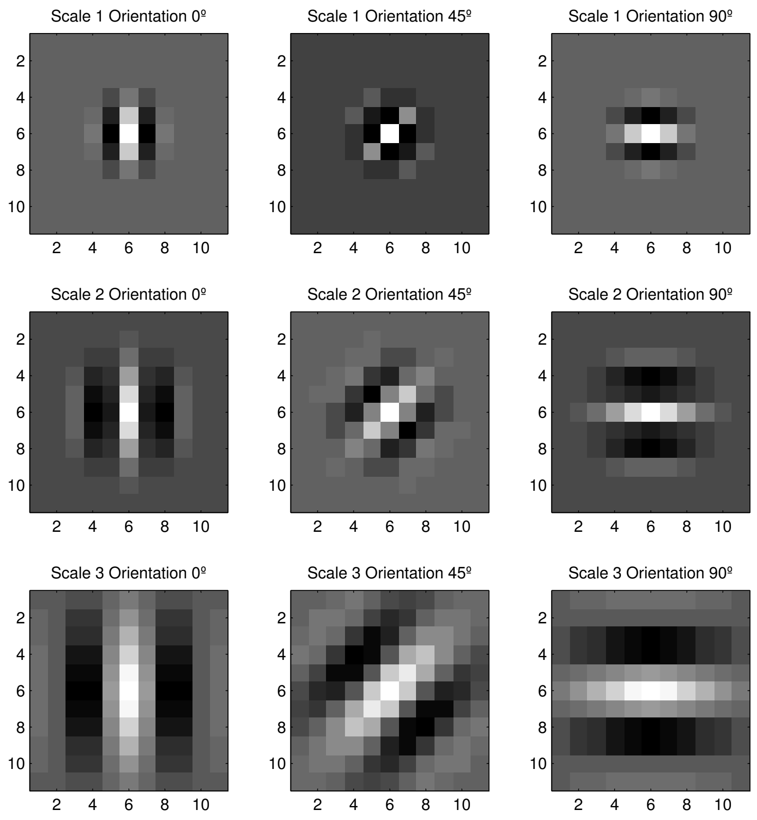
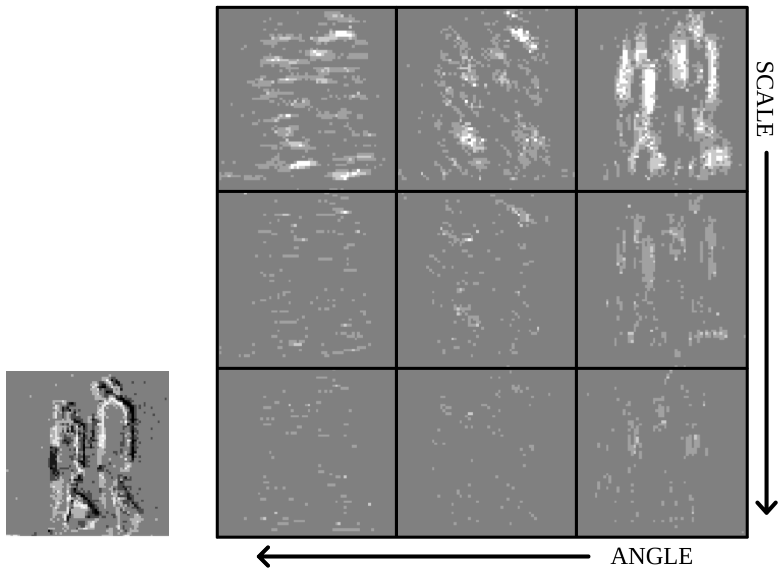


Figure 3.22: Kernels in the bank of Gabor filters for the three chosen orientations and scales.

are copied horizontally from node to node inside each FPGA and also from FPGA1 to FPGA2. To transfer the output events produced by each node (or Gabor filter), the routers are also programmed to copy all output events horizontally from node to node and from FPGA1 to FPGA2. At the right end of all three rows there is an in-FPGA merger block that merges the three flows into a single port connected to the outside, where an AER data logger board [89] is used to capture and timestamp events. The flow between the two FPGAs is fed through three Full-Duplex Rocket-I/O-Based Parallel-Serial Interfaces.

In order to analyze the impact of event hopping from node to node (either intra-FPGA or inter-FPGA) we programmed the same Gabor filter into all nodes in the first row. The Gabor filter output flow is routed to west and north channels at each node to observe the output and measure the latencies between the first output node $(1, 1)$ and the other nodes $(j, 1)$ with $j = 2...6$. These latencies can be measured by observing the delays between request signals req_j $(j = 2...6)$ and req_1.

The delays of req_j with respect to req_1 are shown in Fig. 3.26, for the destination and source-driven algorithms. In this particular example, lower latencies are obtained for the source-driven algorithm, although there is a higher number of events in the



(a) (b)

Figure 3.23: Event-driven Gabor filtering illustration. Background gray represents zero activity pixels, brighter pixels are active pixels sending positively signed events, darker pixels are active pixels sending negatively signed events. (a) Input scene captured by the DVS retina, with pixel activity of both signs (b) Results of the 3 × 3 bank of Gabor filters and sign rectification, so that only positive events result.

network. Every in-FPGA network hop adds a latency of 150ns for the destination-driven algorithm and 70ns for the destination-driven one. For the inter-FPGA hops an additional 350ns is added to the routing delay, for both routing algorithms.

3.6.5 Testing Single-FPGA Maximum Capacity

So far, we did program in each FPGA nine 64 × 64 pixel ConvModules to verify the operation of routers and interfaces. In order to test the maximum capacity of one Virtex-6 FPGA, we checked the maximum of Gabor filters it could hold, together with their routers, peripheral interfaces, SPI configuration circuitry, and input splitter. We used the destination-driven routers, as they are more efficient in terms of FPGA resources. We were able to have the FPGA hold a total of 64 Gabor filters, each with 64 × 64 pixels and kernels of size 11 × 11. The ConvModule array, together with the 1x8 input splitter circuit, the configuration infrastructure through the serial port and the output channels read-out occupy 32720 slices which represents 86% of the Virtex-6 FPGA capacity. Internal memory occupation is 15% for the 36K RAM blocks and 18% for the 18K RAM blocks. We programmed a Gabor filter array by sweeping four scales and 16 angles. Fig. 3.27 shows the collected output events for the 64 filters during the same 160ms time window, while the input DVS retina is observing the

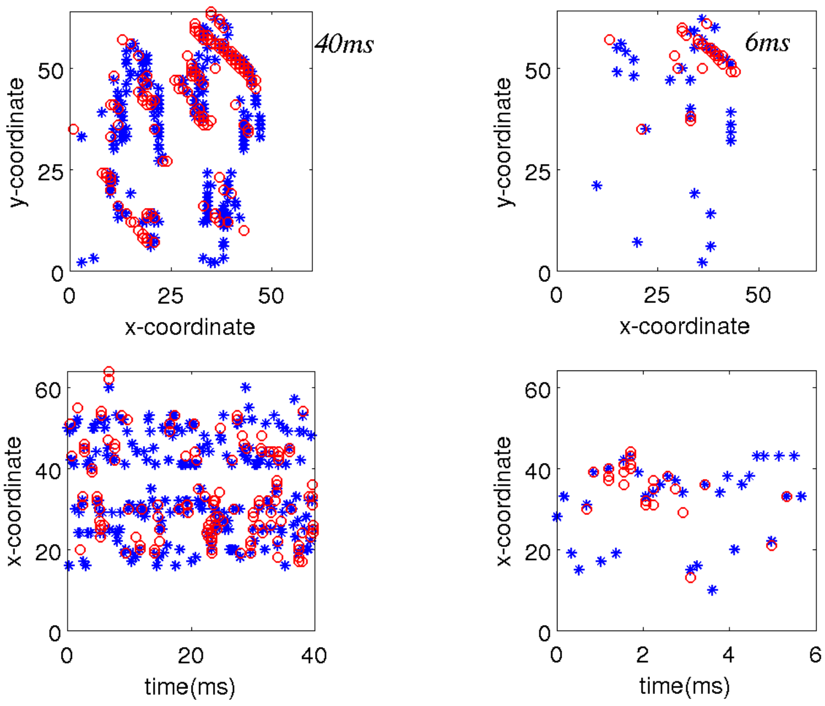


FIGURE 3.24: Illustration of pseudo-simultaneity of event-driven convolutional filtering with a Gabor filter detection of -45^{o} oriented edges. The two top subfigures represent the x/y projection of events captured during 40ms and 6ms, respectively, while the bottom subfigures show the x/time projection of the same events. Convolution module input events are presented with stars and output events with circles.

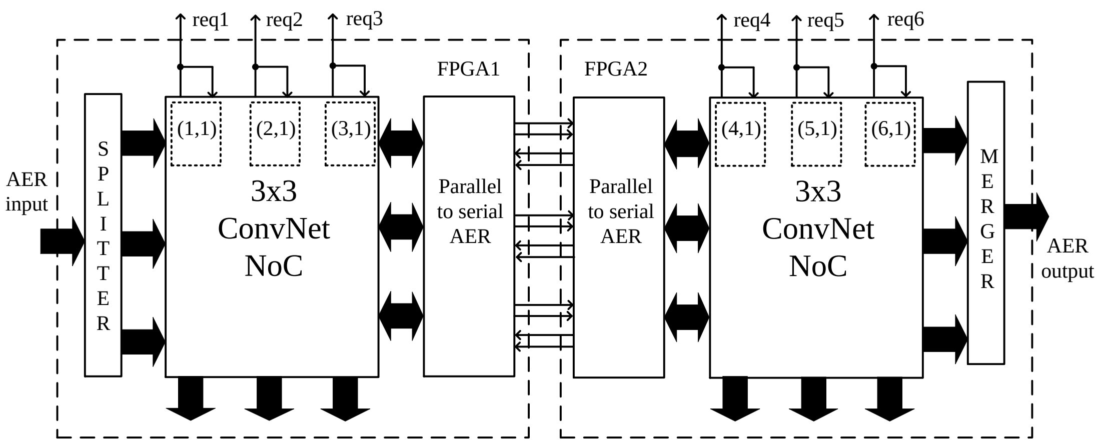


FIGURE 3.25: Diagram of 3x6 Gabor Filter Array Implementation in two FPGAs

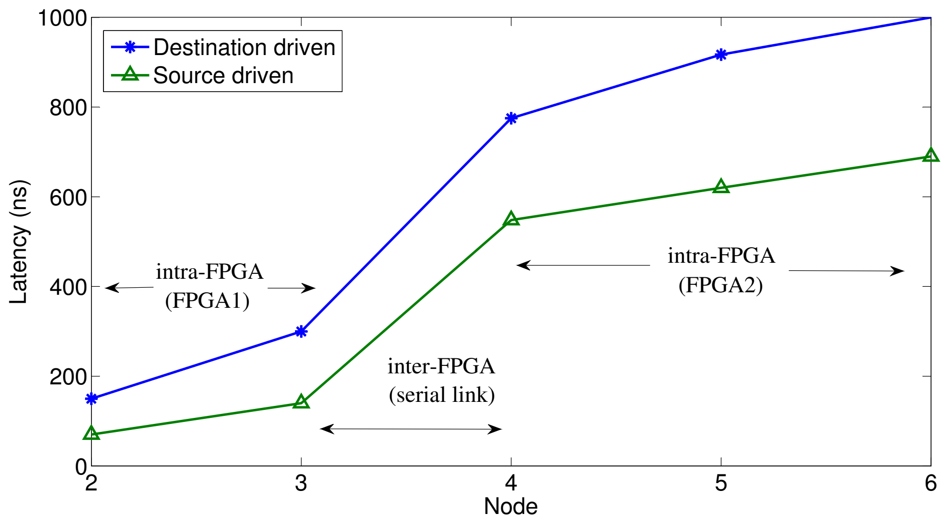


Figure 3.26: Latency between units for the experiment described in Fig. 3.25 for the destination and source-driven routing algorithms.

same two persons walking than in Fig. 3.23(a). Note that, in this case, one single FPGA is emulating a system with $N_{neurons} = 64 \times 64 \times 64 = 2.62 \times 10^5$ neurons and $N_{synpases} = N_{neurons} \times 11 \times 11 = 3.17 \times 10^7$ synapses.

3.6.6 Multi-Module Multi-Layer Convolutional Neural Network Recognition Example

The previously described arrays of Gabor filters represent a one-layer neural system, where all modules (filters) receive the same replica from the input sensor. The example illustrated in this Section is a multi-layer Convolutional Neural Network that performs a character recognition task which has been reported previously and verified using an AER module simulator [90]. It is loosely based on Fukushima's neocognitron [58] or Serre's hierarchical network [2]. We have used the same 64×64 convolution module than above [82] to assemble the 36-node Convolution Neural Network shown in Fig. 3.28. For this a 2D-array of 6×6 AER-nodes was synthesized in a single FPGA. The kernels programmed [90] are illustrated in Fig. 3.29. Kernels *k1* to *k13* perform feature extraction for the 1st layer. Kernels *Ker1* to *Ker6* are used for the 15 filters in the second layer. Convolution outputs are always half-wave rectified (events are assigned a positive sign). The layer 2 output virtual channels (labelled 19 to 41 in Fig. 3.28) are fed to four modules labelled $AGGR_i$ in Fig. 3.28. These are not ConvModules, but plain arrays of integrate-and-fire neurons. Each $AGGR_i$ module includes an AER-merger at its input to merge the traffic from several virtual channels, while forcing their sign bit. For example, module $AGGR_i$ merges events from virtual channels $\{19, 20, 21, 23, 33\}$ and $\{25, 27, 32, 38, 40, 41\}$, while for a positive sign bit for the first set and negative one for the second set. Finally, the 4th layer performs 4

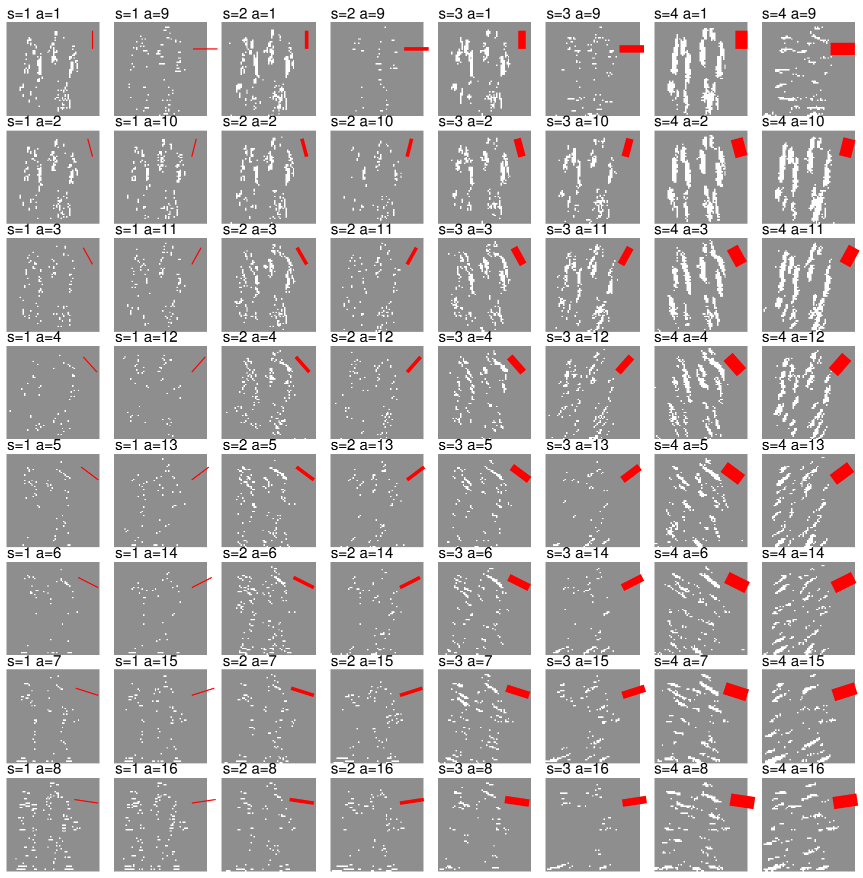


FIGURE 3.27: Output captured from 64 Gabor filter array. Four scales ($s = 1, ...4$) and 16 angles ($a = 1, ...16$) were swept. Gray scale represents number of events integrated in every address position in a 160ms temporal window. Background gray is zero output, while bright pixels represent active pixels. The rotated red bar in each subfigure indicates angle and scale (thickness) of the corresponding Gabor filter.

convolutions in parallel, all with the same kernel $KerC$ in Fig. 3.29.

The system was stimulated with bursts of events representing three different versions of letters A, C, H, and M. Bursts have between 200 to 400 events and last from about 0.5 to 1ms. Fig. 3.30 indicates the main timing properties in this set-up. An input stimulus burst lasts for a time T_{burst}. At one of the four output recognition channels (nodes '46' to '49' in Fig. 3.28) the first output events appears at time T_{first} and the output burst lasts until time T_{last}. Table 3.3 summarizes the measured timing results (T_{burst}, T_{first}, T_{last}) as well as the number of events per output burst for each letter representation. On average, correct recognition output spikes T_{first} appear at about half the input stimulus burst $0.5 \times T_{burst}$ and last for shortly after the input stimulus burst has finished.

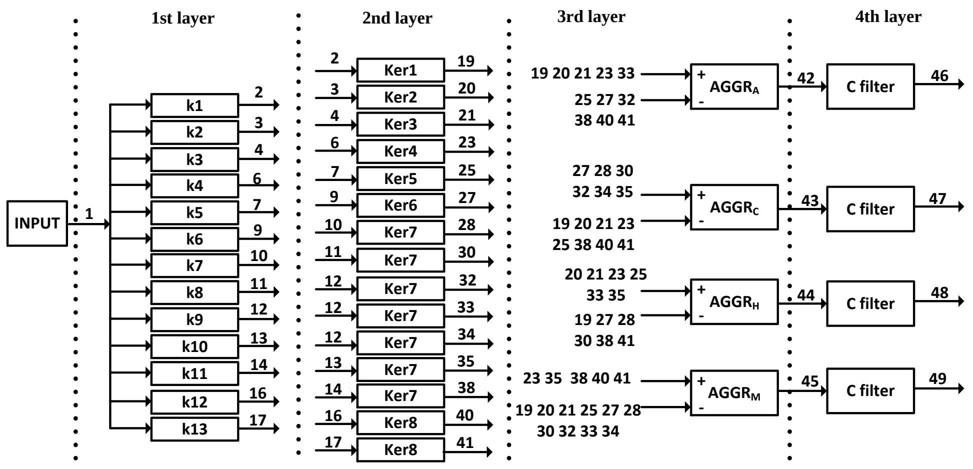


Figure 3.28: Logical network topology for the four letters recognition system. Filters k_i, Ker_i and $KerC$ are kernels programmed in the event-driven convolution modules of layers 1,2 and 4, respectively. Modules $AGGR_i$ are simple integrate-and-fire neurons which count events produced at every address for any of their input channels, producing output events when the count threshold is reached. Node numbers in the figure represent virtual AER channels. Modules $AGGR_i$ include AER-mergers at their inputs, which force the sign bit of incoming events.

Table 3.3: Test results for the letter recognition system.

Input Letter	Burst Time (μs)	Time first output (μs)	Time last output (μs)	# of events A	# of events C	# of events H	# of events M
A1	897	366	941	38	2	0	0
A2	777	589	812	17	0	0	0
A3	867	367	899	39	1	0	0
C1	596	334	619	4	65	0	0
C2	656	373	690	4	87	0	0
C3	476	289	495	5	26	0	0
H1	897	331	928	5	0	68	8
H2	777	460	776	1	0	41	1
H3	746	424	778	1	0	44	1
M1	927	281	885	6	0	5	47
M2	897	584	892	0	0	8	21
M3	837	400	786	0	0	11	28

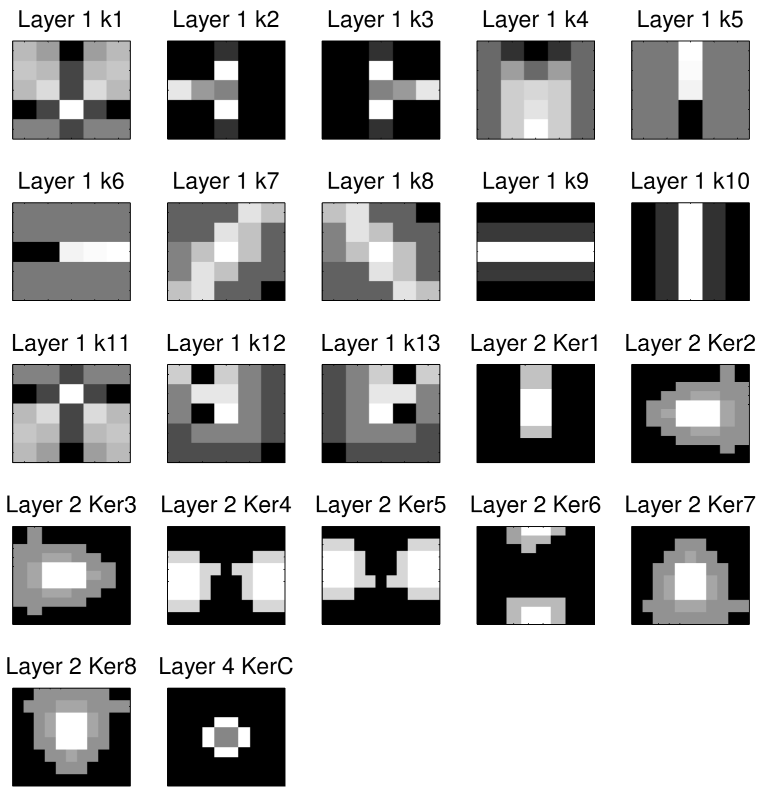


Figure 3.29: Kernels used in the letter recognition system through all the network layers.

3.7 Conclusion

Throughout this Chapter, a scalable method for assembling arbitrary modular AER neuromorphic systems by arranging modules in a 2D grid has been presented. Address events include a module label, and modules include a simple router which routes the events depending on the module labels. The approach is generic for ASIC and FPGA based hardware implementations, but has been tested on single and multiple Virtex-6 FPGAs. Extensive experimental results are provided for AER-based vision processing applications, such as multiple Gabor filtering and character recognition based on convolutional type neural networks.

Experiments have been carried out in a generic development platform (ML605) for Virtex-6 which provides us easy access to the FPGA resources through standard connectors and interfaces. However, the general purpose nature of the prototyping board leads to over engineered solutions, as shown in Fig. 3.31. The board includes DDR, flash and EEPROM memory blocks, connectors for several standards (Ethernet, USB, VGA, PCIe, . . .), an LCD display, expansion connectors, button, switches, LED's,. . . For the experiments described in this Chapter, only the PCIe expansion connector, the configuration resources and the expansion connectors have been used. Due to the large amount of resources contained in the ML605 board, the experimental setups were bulky. This limits the number of FPGAs which can be integrated in the same

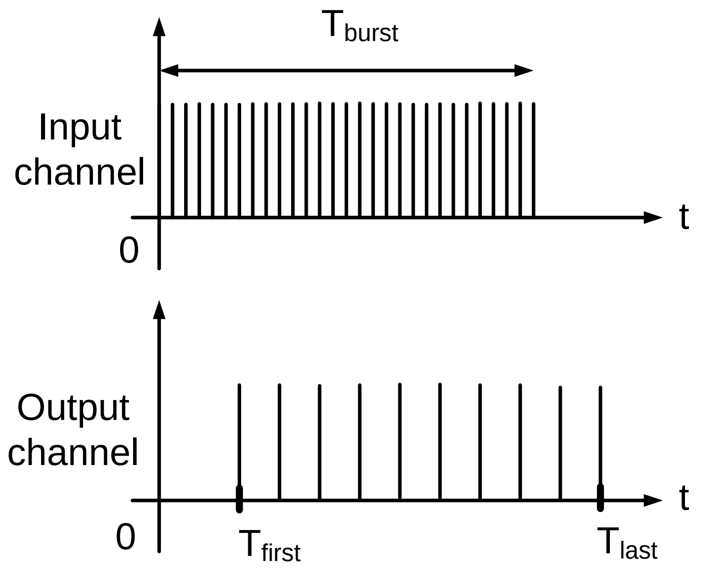


FIGURE 3.30: Timing diagram of the letter recognition process. T_{burst} is the total duration of the input stream which represents the input letter. The first output event in the recognition channel appears at instant T_{first}, while the last one is generated at T_{last}.

system.

However, the ultimate research goal to exploit the advantages of spiking neural networks is migrating this research to VLSI systems, which present higher power, integration density and processing efficiencies. In the FPGA implementation described in this Chapter, we have used conventional high speed serial links offered by Xilinx to implement the off-chip event transmission. This solution provides minimum hardware resources and high bandwidth, but it still suffers from long latencies and high power consumption. The problem is more severe if these links want to be used in large networks where hundred of chips have to be integrated along the same hardware infrastructure. The Rocket I/O circuitry is optimized for applications where the data flow between transmitter and receiver is continuous and a large transmission bandwidth is required. As idle commas are sent when there are no user data to transmit, the high speed circuits keep consuming a large amount of power and the link energy efficiency is reduced.

If the number of links scales up to hundred or thousand, the problem becomes critical if we want to ensure system reliability and an affordable power budget. Taking advantage of the asynchronous nature of AER data, the individual links power consumption can be scaled down with the instantaneous transmitted event rate. This must be done without introducing long event transmission latencies and using low complexity circuits. The next Chapter will address the problem of serializing the parallel AER

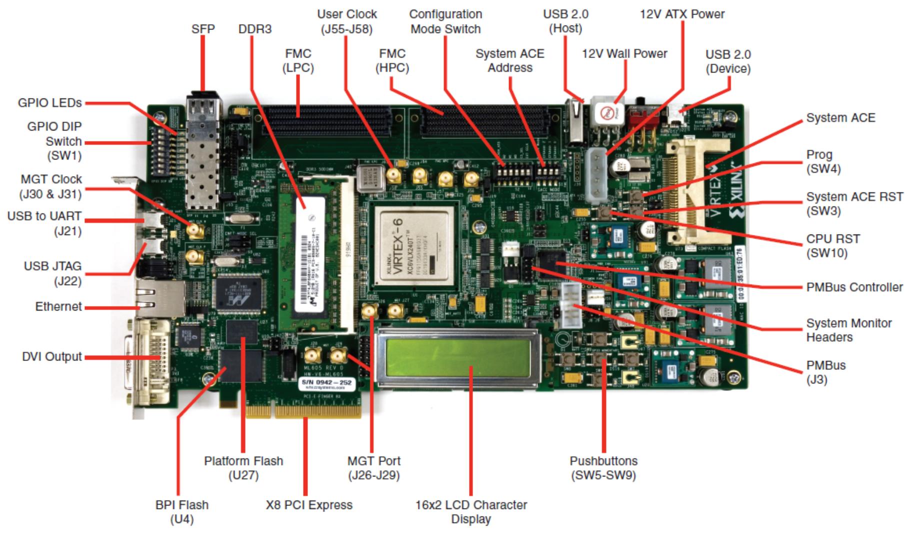


Figure 3.31: ML605 general purpose Virtex-6 development board.

bus by proposing a novel event-driven serialization/deserialization (SerDes) architecture. The design goals for the SerDes circuitry are: low event transmission latency, scaling the power consumption with event rate and robustness.

4 The Event-Driven Bit-Serial Inter-Chip AER link

4.1 Introduction

A system level design for large scale hardware spiking neural networks has been described in Chapter 3. This infrastructure allows to map any arbitrary AER-based neural network logical connectivity by properly configuring the modules which perform the computation. An FPGA-based platform was used for the system level design prototyping, focusing the attention on the hardware implementation of spike-based ConvNets. However, this is a power hungry and inefficient solution to perform the event-based computation as we are not taking advantage of the higher speed and lower power of VLSI AER chips, such as the spike-based convolvers [22, 23]. Moreover, the Rocket I/O resources provided by the FPGA are not optimized to transmit asynchronous nature AER traffic. The idea is to migrate the ideas exposed in Chapter 3 to a VLSI implementation where full custom AER chips can be merged with router engines to assemble large scale multi-chip networks. However, further improvements on the communication layer are needed to improve the scalability and reliability of the structured AER grid.

Size and power consumption become important issues which must be carefully addressed in multichip environments. Serializing the AER parallel bus is mandatory to improve the system scalability and reduce its hardware complexity. As only a few output pins and PCB traces will be required to connect each chip with its neighborhood with the serial solution, hundreds of these chips could be located within the same printed circuit board. Keeping a low power consumption is also essential for a reliable assembly of all hardware components. An important amount of the consumed power will be due to the inter-chip communication layer which transfers events from one chip to another. A power efficient design of the circuits in charge of serializing, transmitting, receiving and deserializing the serial AER flow becomes indispensable.

In this Chapter, we will describe a specialized scheme for Serializing/Deserializing the AER parallel bus which overcomes the limitations of solutions which have been traditionally used for the industry. For this purpose, we first review the clock-data-recovery (CDR) schemes reported in the literature to analyze their efficiency to transmit AER information. After checking the limitations of traditional high speed serial links when transmitting asynchronous data, the serial AER link architecture will be described. A more detailed description of transmitter and receiver blocks will be also provided along with the circuit level design techniques used for the physical implementation. Finally, experimental results in a $0.35\mu m$ CMOS prototype will be presented to validate the robustness and feasibility of the approach.

4.2 Overview of clock-data-recovery (CDR) schemes

In general, commercial high speed serial links require a continuous data flow between transmitter and receiver in order to keep the link synchronized. When there are no user data to be transmitted, an idle comma character is sent and the receiver uses it to identify a pause and discard the received data. In this case, architectures use a PLL or a DLL [91] to lock in frequency and phase with the transmission clock. If no data edges are received, the PLL cannot lock with the transmission clock, and data synchronization is lost. As a PLL or DLL is bandwidth limited for stability or jitter issues, a lost of synchronization leads to hundreds of clock cycles of recovery [92]. This extra latency introduced by the PLL re-locking is not tolerable in event-driven asynchronous AER systems, or systems which require to transmit data in bursts and stop the transmission during data pauses.

There are several known solutions for burst-mode (or event-driven) CDR implementations [93]. They have a data-independent loop which tunes the transmission frequency and keeps it tuned also in absence of data. The phase information is extracted from the data stream, but since there is no feedback with the frequency, proper phase alignment is achieved after a few clock cycles. There are three main types of burst-mode CDRs:

1. *Oversampling CDRs* [97, 98, 100, 101]: a PLL generates several phases of the transmission clock and the input stream is sampled at these instants through a bank of multi-phase samplers. A high speed digital circuit chooses the received bit value analyzing all samples. The major drawbacks of this architecture are that it requires many parallel blocks, each being very high frequency (faster than the transmission frequency), and the jitter budget of the link is very stringent for keeping all phases properly tuned.
2. *Gated-oscillator CDRs* [94, 95]: the synchronous clock is derived from a gated oscillator triggered by pulses generated from the data stream edges. These pulses cause the VCO to start its oscillation with an initial phase given by the data, not requiring any loop to tune the delay.

Table 4.1: Low Acquisition Time VLSI CDR Circuits

	Gated Oscillator [94]	**Injection Locking** [95]	**Broadband PLL** [96]	**Blind Oversampling** [97]	**Semi-blind Oversampling** [98]	**Present Work** [99]
Technology	250nm SiGe BiCMOS	90nm CMOS	180nm CMOS	90nm CMOS	110nm CMOS	350nm CMOS
Technology f_t	200 GHz	180 GHz	120 GHz	180 GHz	150 GHz	15 GHz
Bit Rate (absolute)	10.3 Gbps	5-10 Gbps	2.488 Gbps	2-3.5 Gbps	3.2 Gbps	0.67 Gbps
Bit Rate (relative to f_t)	0.052 f_t	0.028-0.055 f_t	0.021 f_t	0.011-0.019 f_t	0.023 f_t	0.045 f_t
Area (in mm^2)	3×3	0.55×0.55	1×1	0.13×0.13	0.44×0.34	0.9×0.38
Area (in λ^2)	24000×24000	12172×12172	11111×11111	2940×2949	8000×6182	5142×2171
Supply Voltage	3.3V (bipolar)/ 1.8V (CMOS)	1.2V	1.8V	1.2V	1.2V	3.3V
Power consumption	544mW	70mW	54mW	5mW@2.5Gbps	115mW	46mW@10Meps
Acquisition time	1 bit	8-15 bits	100 bits	1 bit	N/A	0 bit
CID tolerance	160 bits	N/A	infinite	infinite	N/A	infinite
Jitter Tolerance	0.27UIpp	N/A	N/A	93UI @ LF 0.68UI @ HF	200UI @ LF 0.4UI @ HF	0.4UI @ worst case
Jitter reported	14.7ps_{pp}/ 2.4ps_{rms} @ recov. clk	15.5ps_{pp}/ 2.2ps_{rms} @ recov. clk	-79dBc/Hz @ 1MHz	142.2ps_{pp}/ 25.94ps_{rms} @ recov. clk	N/A	Not affected by recov. clk jitter
Needs frequency tuning loop	Yes	No	Yes	Yes	No	No
Power scales with packet rate	No	No	No	No	No	Yes
Wide input frequency tolerance	No	No	No	No	Yes	Yes

3. *High-Q Bandpass Filter CDRs* [102, 103]: the gated oscillator can be substituted by a high quality factor bandpass filter that recovers the clock from the edges detected in the data stream.

In both solutions (2-3), there is no phase tracking between the receiving and transmitting clocks, requiring the use of low jitter circuits to generate both frequencies with a very low offset between them. This phase shift also limits the maximum number of consecutive ones or zeros which can be transmitted without any data edge. Moreover, in (2) a replica of the VCO for the frequency tuning is needed. This results in high sensitivity to process, temperature, voltage variations and mismatch. The edge pulse generation and the gated-oscillator are also very demanding because they must be much faster than the transmission frequency. In (3) a monolithic integration is not possible due to limitations of CMOS technologies implementing high quality factor resonators.

Here we seek a low cost and simple solution for burst mode serial interfaces. The solution should overcome the limitations mentioned for existing burst-mode CDR architectures without increasing the system complexity or power consumption. The design goals for our target serial interface are:

- **Low complexity**. External components or huge on-chip devices such as inductors are not desirable, as we want to integrate several of them per chip.
- **Low latency**. After a data pause, the link must recover the transmission clock on-the-fly when a new event is received.
- **Jitter robustness**. The link will be part of noisy environments where a low jitter clock and CDR implementations could be very demanding. If the solution is very robust to timing variations caused by jitter, the clock generation complexity, the CDR, and the system level design can be made more simple and with lower power budget.
- **Low power**. Reducing the power consumption, specially in pauses, of the Serializer/Deserializer circuits is essential to integrate many of them in the same AER system.
- **Arbitrarily long pauses capability**. AER data rates can vary several orders of magnitude. The link should be able to keep the system synchronized in all cases, allowing arbitrarily long pauses in the data stream.

Table 4.1 compares the performance figures of several reported VLSI CDR circuits which have low acquisition times, making them potential candidates for event-driven AER systems. All of them are burst-mode CDRs, except the one with a broadband PLL [96], that we have included for comparison. Usually PLL based CDRs are slow and require long bit streams for locking. This particular design uses a broadband PLL making it faster, although it still requires 100 bits of acquisition time. The fastest ones present one bit equivalent delay of acquisition time, which means the first bit might be lost. On the contrary, the architecture presented in this Chapter does not loose the first bit, thus presenting an effective acquisition time of 0 bits.

Transmission data rate is expressed in absolute values as well as normalized to the process cut-off frequency[1] f_t, for comparison. Similarly, circuit area is also given in

[1]None of the references in Table 4.1 gives the f_t of the technology used. This number has been estimated from other similar technologies.

absolute number and relative to λ (half of minimum feature size). Acquisition times and CID (consecutive identical digits) are expressed in equivalent bit times. As can be seen, for none of the reported circuits power scales down with data packet rate, and all of them have strong jitter requirements.

As we will see later in experimental results, the proposed Serializer/Deserializer pair consumes about 73mW at 10Meps, which would scale down to 0.73mW at an average event rate of 100keps. Consequently, for a large scale scalable neuromorphic system with 120×4 Ser/Des pairs, power consumption would be about 350mW @ 100keps average link rate. However, for any of the solutions in Table 4.1, power consumption of just 120×4 CDRs would range from 2.4W to 261W.

The jitter performance of the compared CDR implementations is shown in Table 4.1. The jitter analysis has been divided in two parts. The first one (Jitter Tolerance) is related to the CDR immunity against the data pattern jitter. The second part (Jitter reported) is related to the recovered clock jitter obtained for a safe data recovery. Our solution is immune to the recovered clock jitter because the clock is generated through edges present in the Manchester-encoded bit stream. Hence, data will be always sampled at proper instants. The CDR design allows for a 0.4UI in terms of input clock tolerance. Only oversampling CDRs present larger input clock immunity at the cost of having a lot of low-jitter sampling phases for the high speed data.

Other researchers have reported serial LVDS links developed specially for AER systems, but using either commercial Serializer/Deserializer circuits or FPGA built-in ones. Table 4.2 compares some of them against the VLSI Ser/Des pair reported in this Chapter. Serial bit rate ranges from 0.7 to 3.125 Gbps, with event sizes of 16, 24, or 32 bits. Table 4.2 also includes the "latency overhead" from the input of the Serializer until the output of the Deserializer, expressed in serial bit speed. Note that the total latency includes two components. The "data latency" to transmit the events bits serially at the available bit rate, plus the extra "overhead latency" (which includes, for example, error correction codes, framing and de-framing, physical circuits delay, etc.).

In general, commercial chips and FPGA built-in Ser/Des pairs add important overheads, resulting in significant extra latencies. Also, all of them require continuous uninterrupted serial transmission. This means they cannot be turned off and on quickly. Only one of them [83] allows for flow control, although it is implemented through an extra LVDS return path. All of them use embedded clock, except one [46] which requires an additional LVDS link for transmitting the clock (which is shared by 16 LVDS data links). This allows for burst mode operation and DDR (double data rate) achieving 1Gbps bit rate with a 500MHz clock. They also report an AER link between Virtex5 FPGAs using Rocket I/O, which is similar to the one by Berge [105] in Table 4.2.

In summary, none of the reported LVDS links (either VLSI or non-VLSI) fulfills all of the requirements we need for a multi-chip AER system with large number of chips. In the rest of this Chapter we describe a VLSI Ser/Des pair which does satisfy the requirements we seek.

TABLE 4.2: Serial Non-VLSI AER Realizations

	Indiveri et al. [83]	**Miró et al. [104]**	**Hartmann et al. [46]**	**Berge et al. [105]**	**Present Work [99]**
Chips	TI TLK3101	MAX9205/6 Spartan3	Virtex5 LX110T	Rocket I/O Virtex2 Pro	VLSI 350nm CMOS
Bit Rate (Gbps)	3.125	0.792	1	2.5	0.67
Event Size (bits)	32	10	24	16	32
Event Rate (Meps)	78	66	31.25	41.66	15
Encoding	8B/10B	none	9 bit CRC	8B/10B	Manchester
Latency overhead (bits)	113 (w/o FC) 398 (with FC)	106	N/A	304	8
Flow Control	Through extra LVDS return path	No	No	No	Yes
Burst-Mode	No	No	Yes	No	Yes
Extra clock path	No	No	Yes	No	No

4.3 High Speed Serial AER link

Fig. 4.1 shows the block diagram of the proposed serial AER sender and receiver. Parallel 32 bit input events *AERin* are managed by the "*Parallel to Serial AER Sender*" system. This circuit receives the input request *reqIN*, generates the signals to latch the event data, and begins the serialization process. When the event data is captured, an acknowledge signal *ackIN* is sent back to the parallel AER sender. AER information is coded with the transmission clock using Manchester encoding [106]. This introduces extra edges at middle bit times. Manchester encoding reduces bit rate to half compared to non-return to zero encoding. However, it allows for very low cost, low power, jitter tolerant, and efficient CDR, thus being very suitable for asynchronous AER transmissions. For the "Physical Driver" and the "Physical Receiver" pads in Fig. 4.1 we use conventional LVDS [107] designs available for our technology [108]. This signal format reduces the power consumption and filters the common mode noise coupled into the channel [109]. The standard specifies a $1.2V$ common mode level and $350mV$ of differential amplitude for the LVDS signals [107–109].

The "*Serial to Parallel AER Receiver*" system in Fig. 4.1 manages the serial stream of Manchester encoded data and converts it into the parallel output *AERout* with its corresponding handshaking signals *reqOUT* and *ackOUT*. Together with the serial transmission, a pair of request *reqSER* and acknowledge *ackSER* signals are used for flow control purposes. When a new event is ready to be transmitted serially, a request pulse *reqSER* is sent to the receiver. If it is not busy processing a past event,

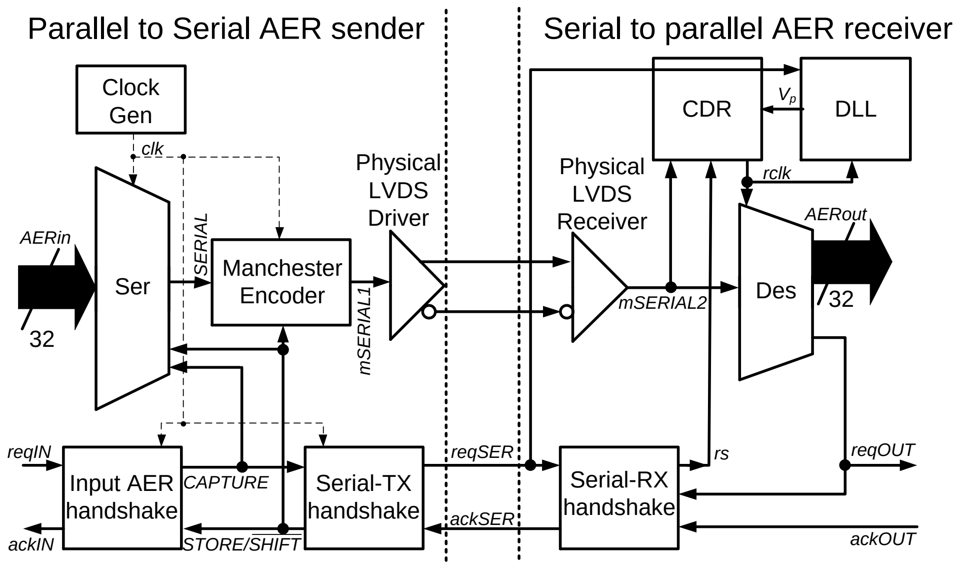


FIGURE 4.1: Serial AER interface system level design.

it accepts the event by activating $ackSER$. In case of not being able to receive a new event, the acknowledge activation is delayed and transmission is temporarily stopped. Signal $reqSER$ is also used by the receiver to initialize all the blocks for receiving a new serial event.

The transmission clock is recovered by a CDR block designed to extract it from the Manchester encoded data. This circuit is in charge of generating a half data rate clock, which is used to decode the serial stream. For proper clock extraction in a Manchester encoding scheme, the receiver tunes a delay [106]. This parameter is very sensitive to temperature, supply voltage and process variations. For this reason, a DLL analyzes the extracted clock generating an analog voltage which controls tunable delay elements. At steady state, this loop keeps the receiver synchronized with the transmission clock.

4.4 Transmitter design

4.4.1 Serializer

Parallel data are latched and serialized by the reconfigurable capture/store/shift register shown in Fig. 4.2. When a new parallel input event is received, data are latched in this register through the *Capture path*, by setting signal $CAPTURE$ high. After this, the serialization is activated during 34 clock cycles using a high speed clock (Clock-Gen). Two preamble bits are added for alignment purposes, letting 32 bits for the AER address. In a shift register based solution, the maximum speed of operation is given by the delay of the critical path. In this case, this path corresponds to the delay of a flip-flop, plus the shift register branch propagation time, plus the set-up time of

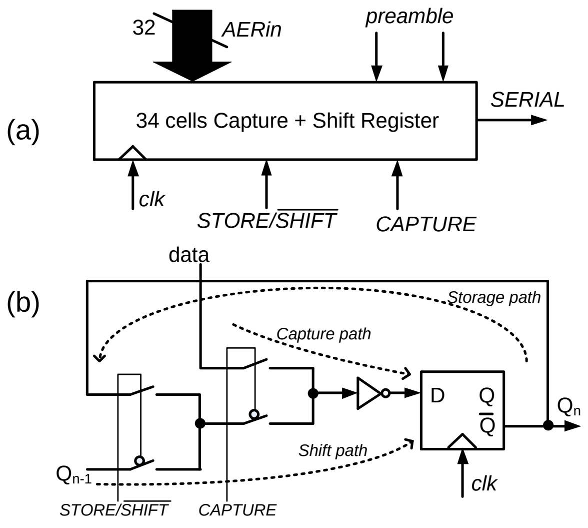


Figure 4.2: 'Ser' block. (a) Reconfigurable capture/store/shift register used to store and serialize parallel input data. (b) Details of one bit cell. If $CAPTURE$ is high, incoming parallel data bits are latched through the *Capture path*. If $CAPTURE$ is low, either data is stored for $STORE/\overline{SHIFT}$ high through the *Storage path*, or data is shifted for $STORE/\overline{SHIFT}$ low through the *Shift path*.

the next flip-flop. Therefore, this solution requires 34 high speed flip-flops, but it can work at higher frequencies than multiplexer based solutions. It also has a fixed delay per bit when increasing the number of bits.

Fig. 4.3(a) shows the details of the 'Input AER handshake' block. It uses a very simple Finite State Machine (genQ FSM), whose functional description is shown in Fig. 4.3(b). The FSM starts in the "IDLE" state, where output signal $CAPTURE$ is low. When $reqIN$ becomes active (low), the FSM switches to state "genQ", where $CAPTURE$ is set high. One clock cycle afterwards, the FSM switches to state "WAIT", setting $CAPTURE$ back low and waiting for the request to be deactivated. This way, an input request ($reqIN$) activates signal $CAPTURE$ during one clock cycle.

Fig. 4.5 shows the details of the 'SerialTX handshake' block, which handles the serial handshaking as well as the store/shift operation of the parallel input event data. The serialization process begins with the $CAPTURE$ pulse generated by the FSM. C-element C_0 sends a $reqSER$ pulse to the receiver indicating that there is a new event to be transmitted. This signal is kept at low level until the acknowledge $ackSER$ coming from the receiver is received. At the same time, two cascaded C-elements C_1 and C_2 are used to generate an asynchronous signal a_1, which is activated when the $CAPTURE$ pulse is detected and the receiver chip has activated $ackSer$, indicating that the event can be transmitted. Signal a_1 is synchronized with the transmission clock using a flip-flop for the $STORE/\overline{SHIFT}$ signal generation. This signal is low

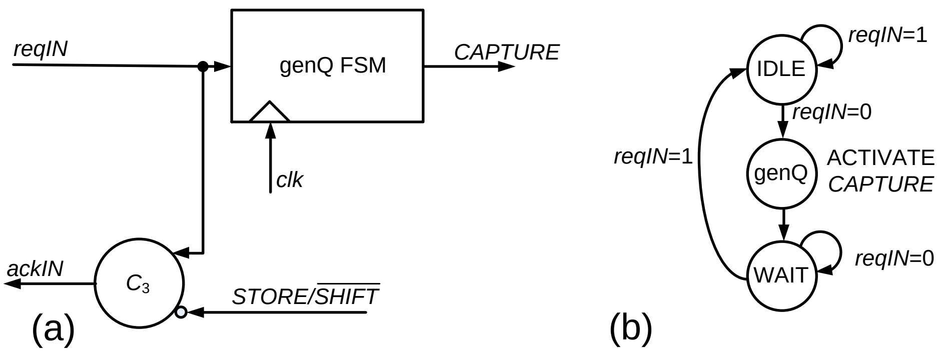
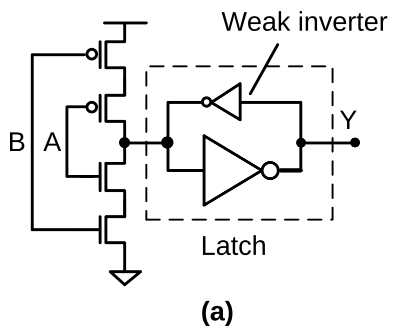


Figure 4.3: (a) Details of 'Input AER handshake' block, and (b) functional description of the genQ FSM.



A	B	Y
0	0	0
0	1	Y_{n-1}
1	0	Y_{n-1}
1	1	1

(b)

Figure 4.4: C-element description. (a) Example implementation using latch based on weak inverter. (b) Truth table, where Y_{n-1} denotes "no change" condition.

during the serialization process. C-element C_3 (in Fig. 4.3(a)) is used to handshake the communication with the transmitting parallel block.

C-elements (or Muller C-gates) [110] are commonly used in asynchronous logic circuits. Their output switches to '1' or '0' only when all inputs have switched to '1' or '0', respectively. Fig. 4.4(a) shows a possible circuit implementation of a 2-input C-element, and Fig. 4.4(b) its truth table description.

The 33-bit auxiliary shift register in Fig. 4.5 is used to calculate the $STORE/\overline{SHIFT}$ signal duration at low level. This register shifts a zero from the register input to the last position. Signal *endSER* is at high level while the register is shifting. After 33 clock cycles, the zero reaches the output and the serialization process is finished, setting *endSer* high. This implementation was preferred over a counter-based option (where the output signal is generated using the counter values) because the counter limits the serialization speed as the number of bits increases. In a shift register, there is no speed per bit penalty when increasing the number of bits and the number of bits per event is not limited by the physical implementation. Note that shift registers in Fig. 4.2 and

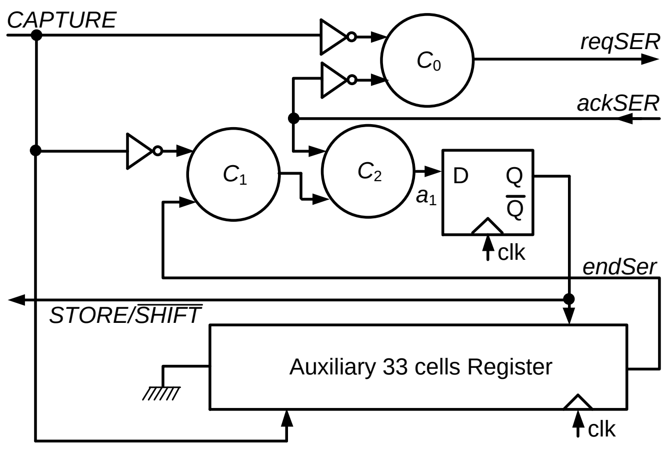


FIGURE 4.5: Details of block 'SerialTX handshake'

Fig. 4.5 can be made of programmable length, so that the user could freely configure the number of bits per event. However, in the presented prototype both registers are of fixed length of 34 and 33 bits, respectively.

4.4.2 High Speed Manchester Encoder

Fig. 4.6 presents the circuit which encodes the serializer output stream into a Manchester format. As the link operates at very high speed, an asynchronous solution is desirable in order to avoid the double rate clock generation. Such an implementation implies an XOR operation between the serial data flow and the transmission clock. A high speed implementation of this combinational circuit requires a careful delay compensation.

Two flip-flops are used to synchronize data $SERIAL$ and signal $STORE/\overline{SHIFT}$. This latter one is used to frame the AER event in the output bursts, because it is active during all the serialization process. A replica of the flip-flop direct path is included to compensate the delay between the output and the clock. A dummy XOR gate is used to equalize the delay between the $SERIAL$ and the $STORE/\overline{SHIFT}$ paths. Finally, a NAND gate stops the encoding when the transmitter is not enabled.

4.5 Receiver design

The deserializing scheme is shown in Fig. 4.7, where two shift registers are used: one triggered by the recovered clock *rclk* rising edges and the other by the falling edges. This way, a half rate clock can be used to decode the input data, reducing power consumption and system complexity. The incoming bits *mSERIAL2* are shifted

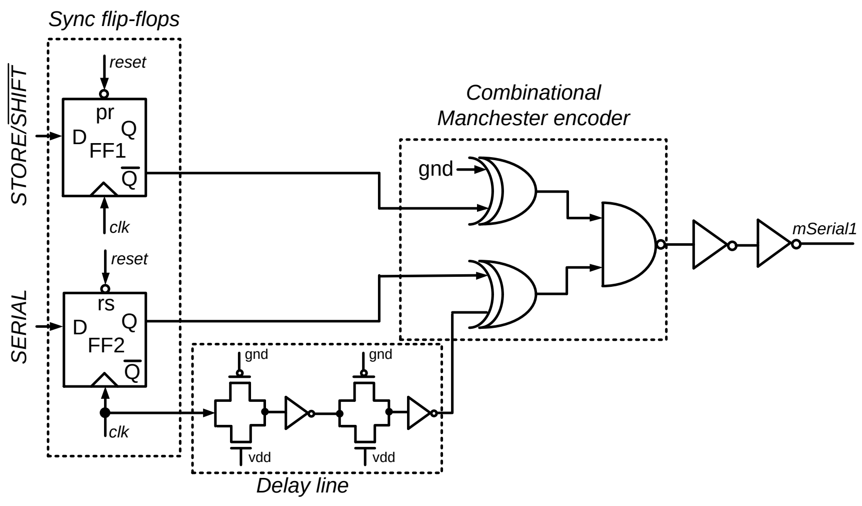


Figure 4.6: Implementation of the high speed Manchester encoder.

through the whole register until the two header bits reach a NAND gate. Then, signal *reqOUT* goes low, starting a new handshaking cycle at the parallel AER output port. This signal is also used to latch the data in a capture register, waiting for this port to read the data. When *ackOUT* is received, the registers are reset to zero and the parallel output request *reqOUT* is deactivated. During this process, the receiver does not send any *ackSER* pulse to the transmitter, suspending any new data transmission which could overwrite the current event.

4.5.1 Clock Extraction circuit

Manchester data include additional edges which are used by the receiver to extract the synchronization information. The CDR must generate a recovered clock through these extra edges, ignoring bit data dependent edges. The circuit in Fig. 4.8 is able to extract this information and to generate a half rate clock for data recovery [106]. The circuit consists of a Double Edge Triggered Flip-Flop (DETFF) configured as a frequency divider. The DETFF's clock input is directly connected to the serial input signal *mSERIAL2* provided by the "Physical LVDS Receiver" (see Fig. 4.1). As information edges of the Manchester signal *mSERIAL2* want to be filtered, a five inverter delay line is included between the DETFF input and output.

The delay value is critical for the CDR proper operation. Controlling the delay of an inverter in a CMOS technology is difficult due to process, temperature and supply voltage variations. For this reason, these delay elements have analog control voltages which tune their delay value. This delay can be controlled by adjusting the gate voltages of a CMOS switch at the inverter input. This implementation allows a very wide tuning range so that the CDR can extract the transmission clock in a large range

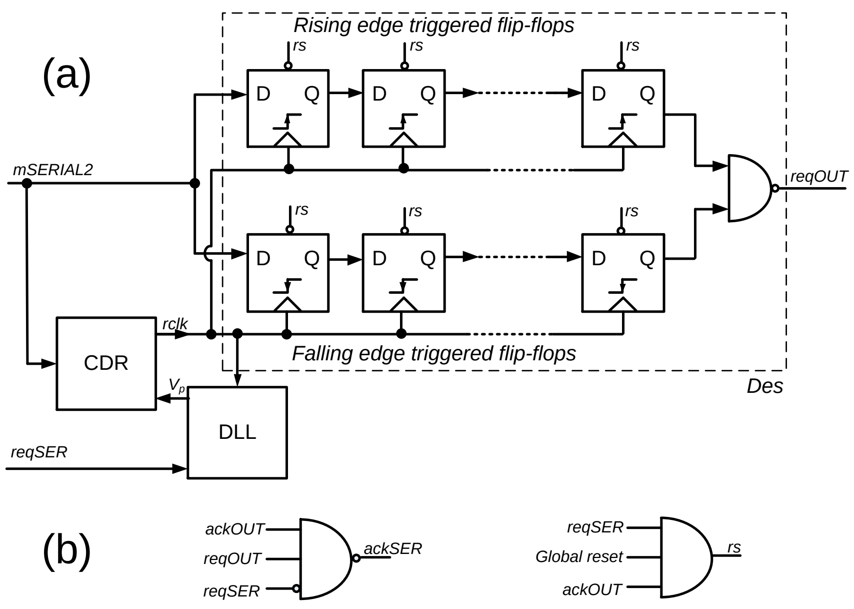


FIGURE 4.7: (a) Details of deserialization block '*Des*' in Fig. 4.1. (b) Details of block '*Serial-RX handshake*' in Fig. 4.1.

of frequencies.

If this delay is greater than $T_b/2$ and lower than T_b (T_b is the bit time), data bit dependent edges will be filtered and the DETFF will not switch at them. However, if the delay is not properly tuned, the DETFF would switch at any *mSERIAL2* edge. Fig. 4.9 shows the CDR circuit chronogram for two different conditions of the analog control voltage. In Fig. 4.9 (a) all delays are properly tuned and the CDR is extracting the recovered clock properly. In case (b) there is no lock in the delay tuned loop and extra edges of Manchester code are causing recovered clock triggering. If there are no incoming serial data, there are no clock edges, saving dynamic power during pauses. The value given for the jitter tolerance 0.4UI in Table 4.1 allows the input clock to change its frequency in the CDR locking range during each bit. If the frequency changes slowly, the DLL will track it in the loop and larger frequency changes can de adapted. This is because the Manchester code provides extra data edges for phase comparison at every bit time. Therefore, 0.4UI is a very pesimistic estimation for the tolerated input data jitter.

A DETFF [111]-[112] clocked by the data stream generates the recovered clock edges. Hence, there will be a clock edge after a data edge if delays are properly tuned. As recovered clock edges are generated directly through data edges, the receiver is very robust against jitter. Even when the transmitter has a very poor jitter performance, that will not impact on the bit error rate because of this feature. That opens the door for very low cost clock generation solutions which do not require very power hungry clock generation circuits.

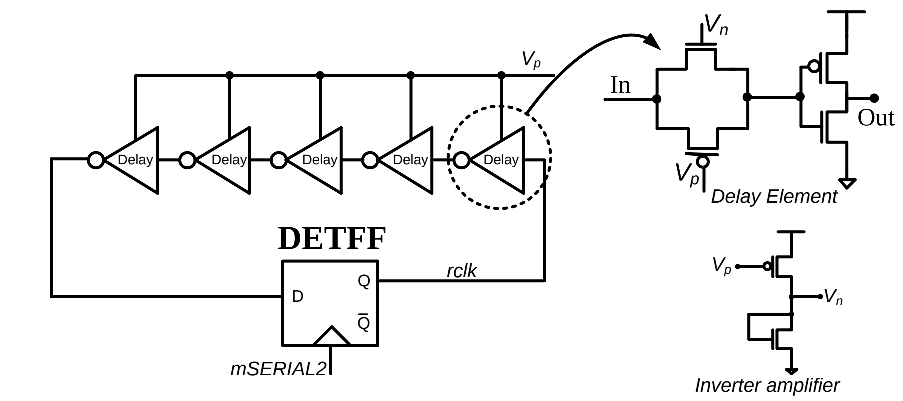


Figure 4.8: Details of block 'CDR' in Fig. 4.1 and Fig. 4.7.

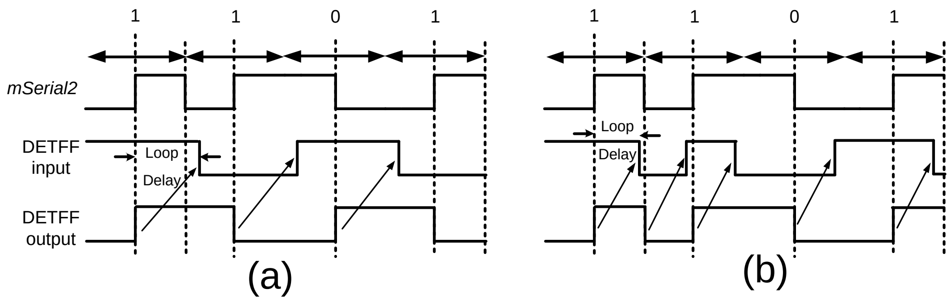


Figure 4.9: CDR circuit chronogram in which (a) delays are properly tuned or (b) are out of lock

The clock is recovered thanks to a DETFF which operates at very high speeds. The circuit in Fig. 4.10 is a high speed implementation of a DETFF used in the burst-mode CDR. The flip-flop is composed by two latches, corresponding to the parallel branches in Fig. 4.10. Each path is high and low level sensitive, respectively. The output is combined in a shared node. Both branches have a reset signal that puts the flip-flops in a known state before starting the deserialization.

4.5.2 Delay tuning circuit

Fig. 4.11 shows the DLL circuit used to tune the delay in the CDR. This circuit compares the recovered clock signal *rclk* with a one clock cycle delayed version, independently of the clock frequency, as long as it is within the delay elements tuning range. This delayed version is generated by 16 cascaded delay elements, identical to the five used in the CDR. A Phase and Frequency Detector (PFD) compares the extracted

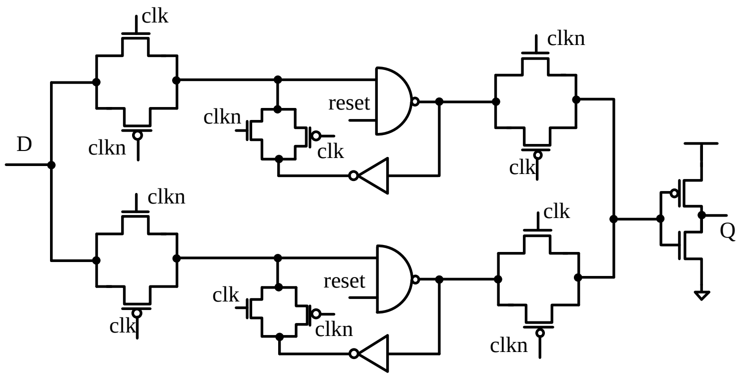


Figure 4.10: Double edge triggered flip flop for clock extraction.

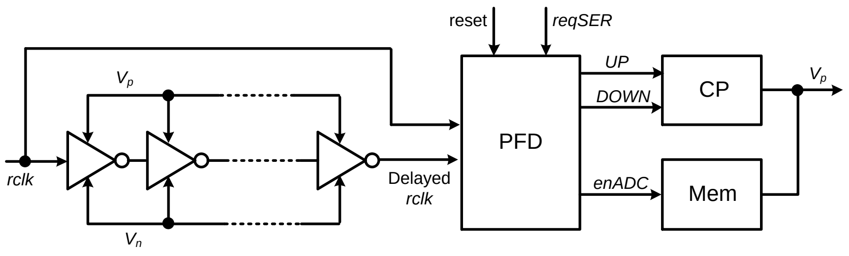


Figure 4.11: Details of the DLL (Delay Locked Loop) block in Fig. 4.1 and Fig. 4.7.

clock and the delayed clock phases and generates two correction signals for a Charge Pump (CP). Signals UP and $DOWN$ make the analog control voltage V_p evolve to correct the phase lag between the delayed and recovered clocks. The link can tune the CDR for any frequency which belongs to the delay element tuning range.

Fig. 4.12 shows the schematics of the PFD which compares the phase of the recovered and delayed clocks. Flip-flops FF1 and FF2 are used to detect the phase lag between both signals by triggering their outputs with every new edge. Rising edges cause that $DOWN$ and UP signals get activated and the combinational logic resets the flip-flops when both are at high level. The duration difference between both signals codes the existing phase shift. While the counter is counting, output signal $enADC$ is low. After counting 16 clock cycles $enADC$ is set high, activating the memorization circuit ('Mem' in Fig. 4.11) for control voltage V_p. When a new event is received ($reqSER$ goes low) the counter starts counting again, $enADC$ goes low again, the memorization circuit is deactivated, and the charge pump controls again voltage V_p.

If the DLL compares a 360° delayed signal with the reference clock, there is an initial clock edge that should not be compared. For this reason, the counter shown in Fig. 4.12 resets the PFD until the first recovered clock is produced. Then, the PFD

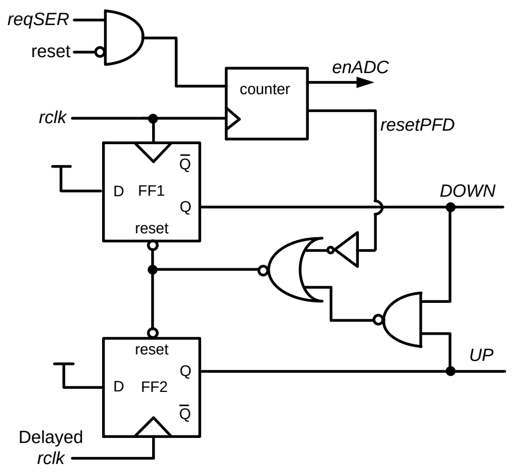


Figure 4.12: Details of the Phase and frequency detector (PFD) circuit used in Fig. 4.11.

is enabled to compare the phase for a number of cycles given by the number of bits used to implement the counter. In this design, a 4-bit counter was chosen and 8 phase comparisons are performed for every AER event.

Fig. 4.13 shows the schematics of the charge pump which pulls up and down the analog control voltage V_p, depending on the information provided by the PFD. The CP integrates a bias current I_b on the on-chip capacitor C_{CP} (of value $1.5pF$) during a time slot given by the difference in the duration of UP and $DOWN$ pulses. A current steering topology was chosen [113] to reduce spurious current injected in the output capacitor when switches controlled by $DOWN$ and $UPnot$ are turned on and off. These currents can cause a phase offset in the DLL loop which can lead to a significant error in the delay tuning precision. This is particularly important in high speed charge pumps, where very low phase shifts want to be tracked.

Bias current I_B is always flowing through current sources M_{BP} and M_{BN}, keeping these transistors saturated. This reduces the current peaks generated when current is switched from one branch to another. Capacitors were included at the current sources output nodes to help these bias currents keep their terminal voltages constant. An analog buffer was included to clamp both CP branches and maintain the same conditions in the two current paths. This way, both current paths are completely symmetric and current peaks are reduced to a minimum. The matching between NMOS and PMOS currents can also cause phase offset in the DLL loop. In this design, the current sources transistor lengths were carefully designed to achieve a good trade-off between speed and matching.

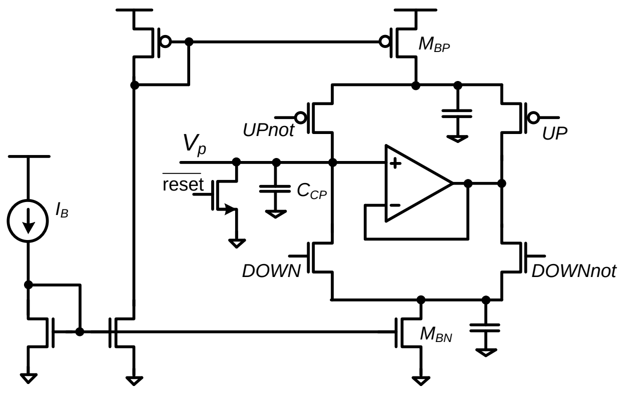


FIGURE 4.13: High speed charge pump design.

4.5.3 Control voltage memorization circuit

Arbitrarily long event pauses can make the CP capacitor C_{CP} discharge through leakage currents in the switches and elements connected to it. A memorization circuit is required to retain this voltage during event pauses. A digital storage element is mandatory in order to guarantee an arbitrarily long memorization time. The synchronization is controlled by analog voltage V_p which must be properly interfaced for digital conversion and storage using adequate precision.

The circuit in Fig. 4.15 is used to store the delay tuning voltage V_p. Intensive simulations showed that a 5 bit ADC (Analog to Digital Converter) architecture provides enough precision to keep the link synchronized between consecutive events. This can be understood with the help of Fig. 4.14, which shows the corner simulated delay in pico seconds of one “Delay Element” (see Fig. 4.8) as function of control voltage V_p. The tuning loop has to adjust the delay to 1/8 of bit time T_b [106] (i.e. $250ps$ for 500Mbps). The maximum V_p range is $[0V, 2.7V]$. Quantizing to 4-bit results in half an LSB of $84.4mV$, or $42.2mV$ for 5-bit, resulting in phase errors of $55ps$ and $20ps$, respectively, for the worst case scenario, e.g. around the $250ps$ delay. On the other hand, the CDR delay has to satisfy [106] $5T_{delay} > T_b/2$ and $5T_{delay} < T_b$. Consequently, for the nominal case $T_b = 2ns$ this is $200ps < T_{delay} < 400ps$. In the ideal situation the loop would lock to $T_{delay} = T_b/8 = 250ps$, so that a quantization error of up to $50ps$ could ideally be tolerated (i.e. 5 bits). Nonetheless, we performed extensive corner and mismatch simulations of the complete circuit including all circuit non-idealities to make sure no extra bit was necessary to guarantee safer margins.

A flash architecture is used because of its simplicity and an asynchronous design is implemented to reduce its inherently high power consumption. The “Bank of 31 comparators” is used to determine the resistor ladder level V_{rj} that better approximates the analog control voltage V_p. The “Asynchronous Digital Controller” takes this decision and controls the “Bank of switches”, connecting the chosen level with the output node. The “Asynchronous Digital Controller” is composed of 31 “selection elements” which

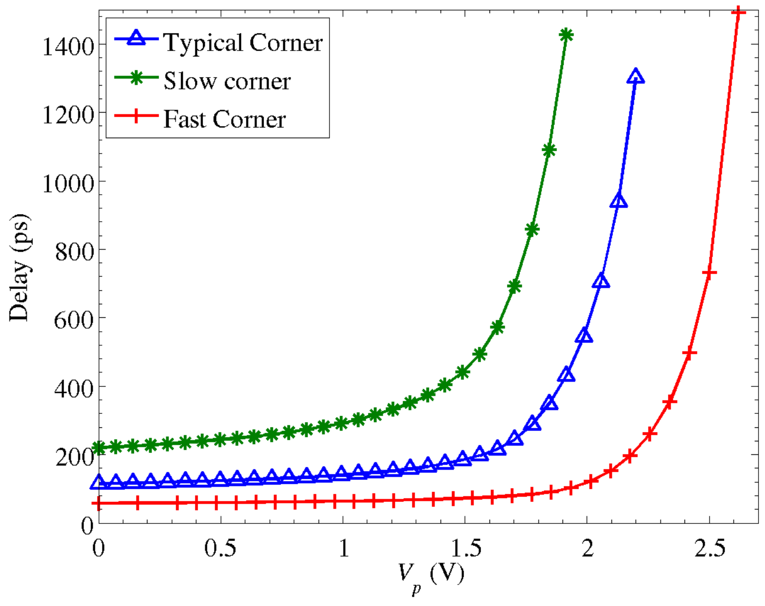


Figure 4.14: Corner simulations of CDR 'Delay-Element' delay as function of control voltage V_p.

generate the enable signals sel_j for the "Bank of switches". They perform a logical operation between the outputs of consecutive comparators to find out the DAC output which makes a comparator to switch from a positive to a negative comparison. Each comparator provides an additional output signal "$valid_j$" to notify that the comparison has settled. The operation of the asynchronous controller is enabled through signals "$valid_j$" of the comparators.

Fig. 4.16 shows the schematics of the comparator used in the ADC [114]. It was designed as a two stage comparator with a pre-amplifier, represented by a PMOS differential pair and a latch built with a two inverter loop, which stores the comparison result. Signal $COMP$ is used to reset the circuit before any comparison, pre-charging all the comparator nodes to known initial values. Moreover, the bias current is switched off to save power when no comparison is being carried out. When the comparison process finishes, an XOR gate sets signal "$valid_j$" high to notify the digital controller that the comparison value is ready to be read.

One of the main drawbacks of flash ADC architectures is their high power consumption, since all comparators operate in parallel when the device is converting. In our asynchronous implementation, the output bits are calculated sequentially and the comparators are switched off and on at every stage. Fig. 4.17 shows the tree-like connectivity of the comparators to implement this asynchronous conversion process. The tree has $log_2(n+1)$ levels, where $n = 31$ is the number of comparators. In our case there are 5 levels. Depending on the result at one level and the corresponding "$valid_j$"

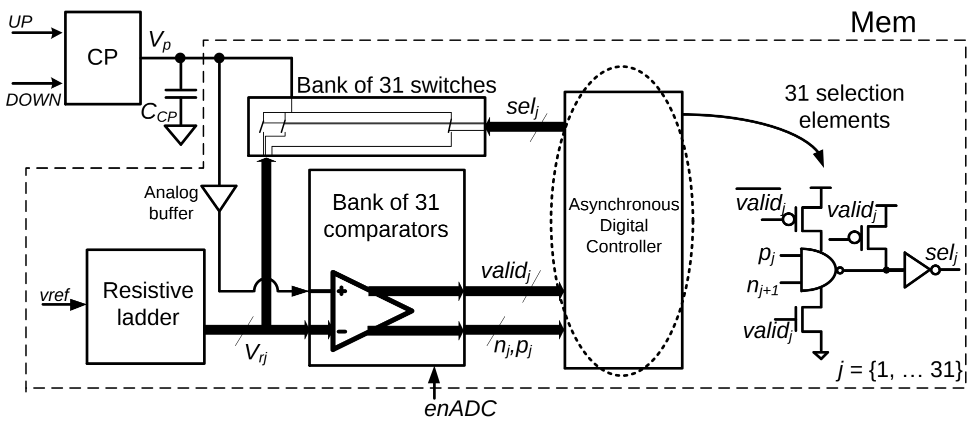


FIGURE 4.15: Analog control voltage digital storage system.

signals, only one comparator is activated at each level. Although this is a slow process, the conversion rate is not a major concern in this application because the CP voltage variation in pauses is caused by leakage currents with a time constant in the order of milliseconds. Hence, besides the asynchronous implementation, currents of $10\mu A$ are used to bias the comparators in order to further reduce the power consumption.

4.6 Experimental results

A proof of concept test prototype using the presented serial AER link has been implemented in a $0.35\mu m$ CMOS technology. Nominal data rate was chosen to be 500 Mbps using Manchester encoding and supply voltage was 3.3V. The transmitter requires an area of 350x375μm^2 and the receiver occupies 900x380μm^2, not including LVDS pads. Fig. 4.18 shows a micro photograph of the fabricated chip, highlighting the main components. The test channel used in the experiments consists of a pair of PCB traces forming a 100Ω differential micro strip line of $3cm$ length. The clock was generated through a simple VCO based on a ring oscillator that can be externally tuned by an analog voltage. No PLL or DLL based solution to generate a jitter clean master clock reference was integrated. This is the worst situation [115] for clock jitter performance. However, it is useful for showing the circuit robustness to high jitter and demonstrate that a very simple clock solution is enough for this architecture.

Fig. 4.19 shows the test set-up. Two 16-bit USB-AER boards [89] receive events from a PC connected to them through USB ports. These events are sent through a parallel connector to the test board using the AER protocol. Each USB-AER board provides a 16-bit AER bus, but a 32-bit version is needed for the serial transmitter. For this reason, streams coming from two different boards must be synchronized to form up a 32-bit AER bus. A CPLD implements a C-element for $req1$ and $req2$ to generate

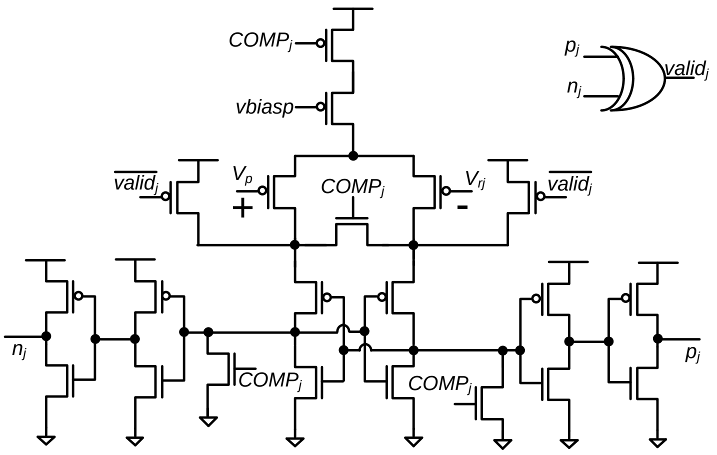


Figure 4.16: Circuit schematics of comparator for asynchronous flash architecture.

$reqIN$. This way, both 16-bit parallel input streams are merged into a single 32-bit one which is serially transmitted. At the receiver side, the output flow must be split into two AER streams, each of which can be captured by a single 16-bit parallel USB-AER board. The CPLD performs this task in the same way than in the transmitter side, managing $ack1$ and $ack2$. Each output USB-AER board is able to transmit captured events to the PC through its USB connection.

Fig. 4.20 shows the measured AER protocol signals (a) at the transmitter parallel input, (b) at the serial inter-chip link, and (c) at the receiver parallel output, as well as the serial data stream. The CPLD generates $reqIN$ when a new 32-bit event is ready to be transmitted. When this event data is latched by the receiver, $ackIN$ is activated during all the serial transmission process. The delay between the $reqIN$ and $ackIN$ activations in Fig. 4.20(a) is $10ns$ for a $550MHz$ transmission clock. The $ackIN$ pulse duration is $62ns$, corresponding to 34 clock cycles. The transmitter cannot send a new event until this acknowledge signal is deactivated.

The transmitter uses the serial AER signals to perform a flow control mechanism. The receiver can send a new event if the transmitter activates $ackSER$ after $reqSER$. The delay between the request $reqSER$ activation and the acknowledge $ackSER$ activation is $2ns$. In this case, the protocol is completely asynchronous and only depends on the logic and pads delays. Handshaking is also implemented at the receiver output. This way, the serial AER transceiver can communicate with any parallel AER chip without any protocol conversion. The $21ns$ delay between $reqOUT$ and $ackOUT$ is due to the USB-AER boards.

Fig. 4.21 shows the high-speed serial AER signal measured at the receiver input. This figure illustrates the asynchronous event driven nature of the AER data sent through the link. Transmission frequency in this case is 500Mbps and event rate is 4.8Meps. Maximum event rate through the interface is limited by the transmission

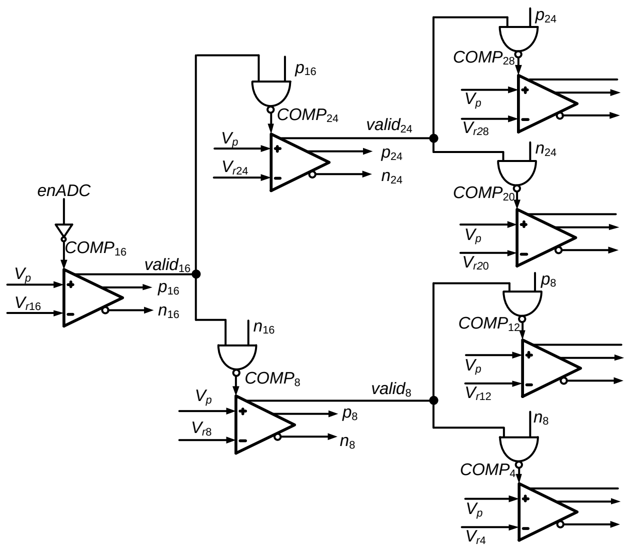


Figure 4.17: Asynchronous low power flash architecture in “Bank of 31 Comparators” in Fig. 4.15

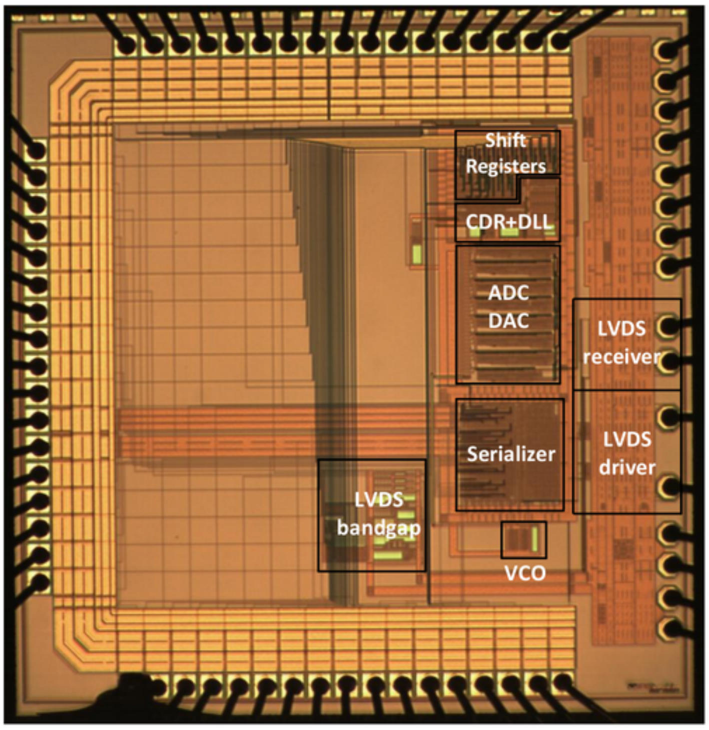


Figure 4.18: Micro photograph of fabricated test prototype

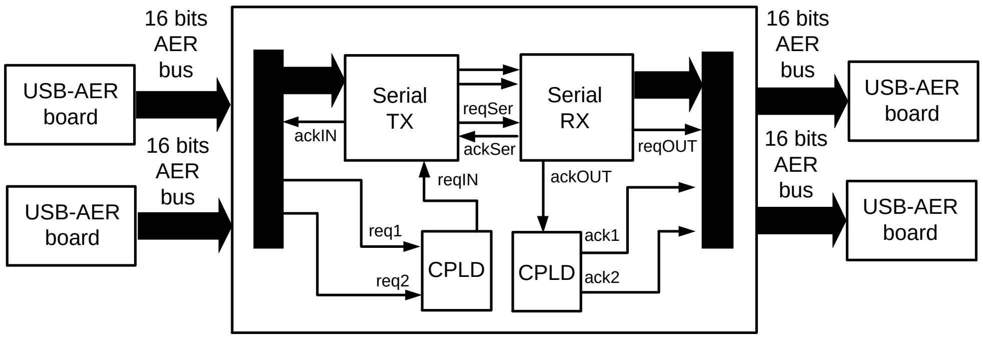


Figure 4.19: Test set-up to generate a 32 bit AER pattern.

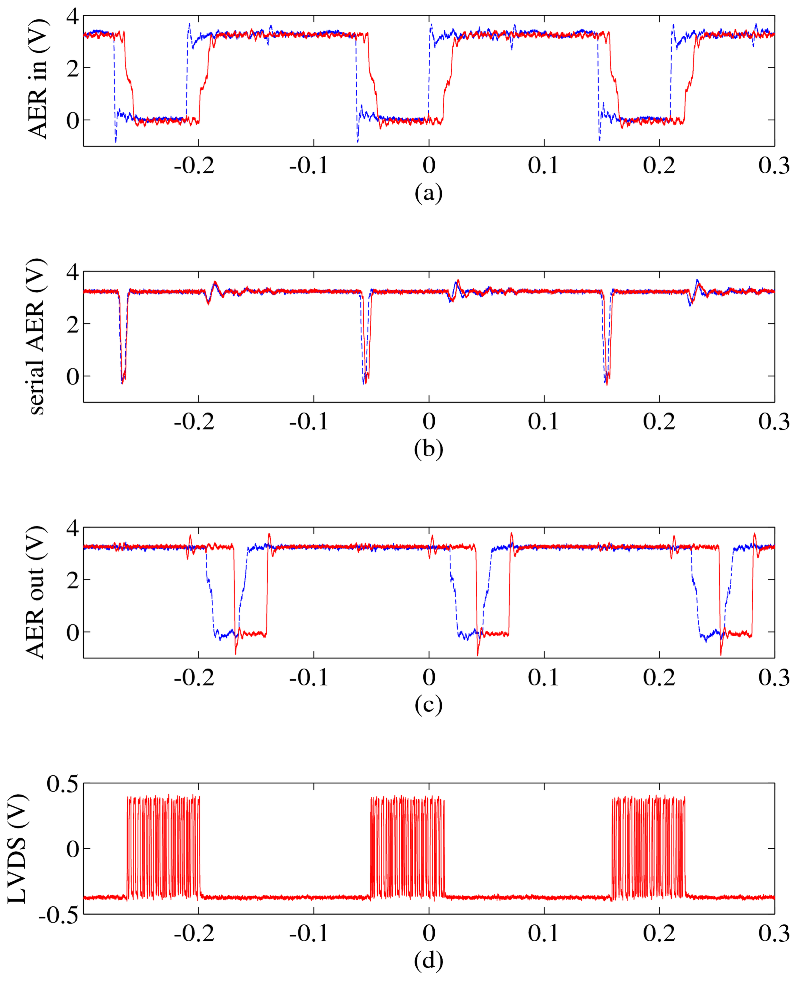


Figure 4.20: AER protocol management at (a) *reqIN* (dotted line) and *ackIN* (continuous line) signals (b) *reqSER* (dotted line) and *ackSER* (continuous line) signals (c) *reqOUT* (dotted line) and *ackOUT* (continuous line) signals (d) differential mode of the LVDS signal. Time scale is μs.

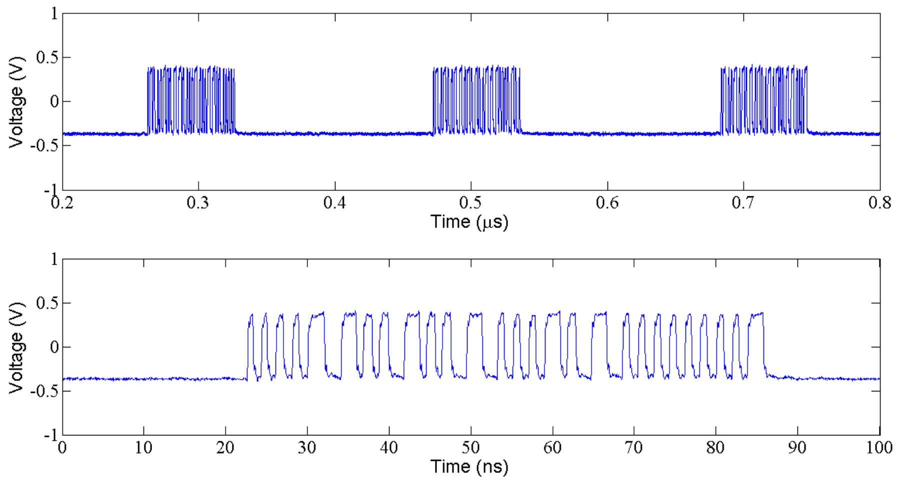


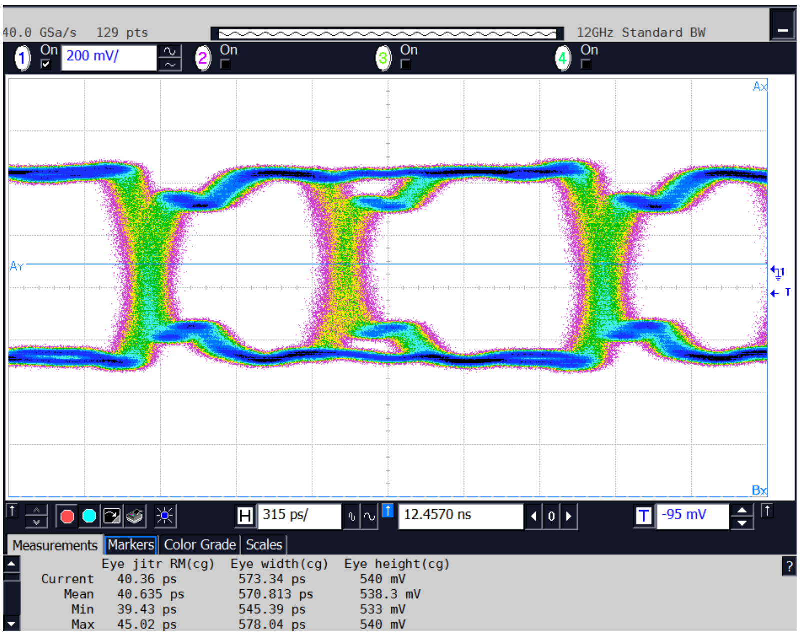
Figure 4.21: Measured LVDS signals at receiver input.

frequency and delays associated with the AER protocol signals. To measure this link event rate, $reqOUT$ and $ackOUT$ were shorted. Maximum event rate for the nominal 500Mbps bit rate was measured to be 12.5Meps, corresponding to an input-to-output request latency of 80ns. If a 670MHz transmission frequency is set, a 15Meps and 66ns latency can be achieved.

Fig. 4.22 shows the eye diagram measured for a 500Mbps bit rate using an Agilent DSO81304B Infinium 12 GHz bandwidth oscilloscope. A 40ps of rms jitter was measured and event error rate (EER) resulted to be below 3.3×10^{-10} (3.1×10^{9} events where analyzed without any error found). This demonstrates the robustness of the approach, capable of keeping very low EER without very stringent jitter requirements.

Fig. 4.23 shows the control voltage V_p evolution when the link is powered-up the first times. Voltage V_p is available to be measured through an on-chip buffer to not affect the CP integration capacitor. After reset, the control voltage is set to ground by a switch and evolves to its steady state value when events are received. The analog control voltage convergence process lasts 6μs and requires 6 events in this particular case. In general, this delay depends on the input event rate, the input patterns, and the bias settings. When the DLL stabilizes at the steady state, this control voltage remains constant. The ADC block keeps it memorized, tolerating arbitrarily long event pauses. This was measured in the laboratory by stopping the interface and measuring the control voltage with an oscilloscope. It was noticed that it does not change during hours of pause.

The link can tune the delays to receive serial streams with a wide range of transmitting frequencies. Maximum frequency is limited by parasitic effects, not only by the delays tuning range. As the oscillator frequency can be programmed by controlling a test board potentiometer, the transmission data rate was swept in the range of 175



```
          Eye jitr RM(cg)   Eye width(cg)   Eye height(cg)
Current     40.36 ps         573.34 ps        540 mV
   Mean     40.635 ps        570.813 ps       538.3 mV
    Min     39.43 ps         545.39 ps        533 mV
    Max     45.02 ps         578.04 ps        540 mV
```

Figure 4.22: Measured eye diagram at receiver input.

to 670 Mbps, in which the link operates correctly.

Transmitter and receiver power consumption depends on event rate. Fig. 4.24 shows the current consumption of the serializer and the deserializer for different event rates. As can be seen, for high event rates, current consumption grows proportionally to data event rate. For lower event rates, current consumption stabilizes to constant values of $3.1mA$ and $2.0mA$, respectively. Besides the Serializer and Deserializer circuits, other components consume a constant event-rate independent current. For example, the LVDS physical transmitter pad pair (driver) consumes $8.2mA$, while the LVDS physical receiver pad pair consumes between $8.4mA$ (for low event rate) and $9.8mA$ (for high event rate). The VCO used for the clock generator consumes $12.55mA$.

In this first proof-of-concept test prototype we were mainly concerned with a scheme for quickly turning on and off the Ser/Des pair. However, for a practical realization to be used in a large scale scalable neural system, all power hungry components need to be turned off and on quickly as well. For example, standby power consumption of the Serializer can be drastically reduced by simply adding a clock-gating mechanism during pauses. Fig. 4.24 shows the simulated power consumption of an improved Serializer with this simple addition. The fabricated Deserializer also has an important stand-by power consumption because of the following three components: (a) the small "inverter amplifier" in Fig. 4.8 which requires a high current to be fast, (b) the "analog buffer" in Fig. 4.15 to buffer voltage V_p and isolate it from switching noise coming from the comparators, and (c) the Charge Pump. Cases (b) and (c) are corrected easily by switching between ON and OFF bias currents. For case (a) the inverter current cannot be switched because voltage V_n needs to be kept stable during pauses.

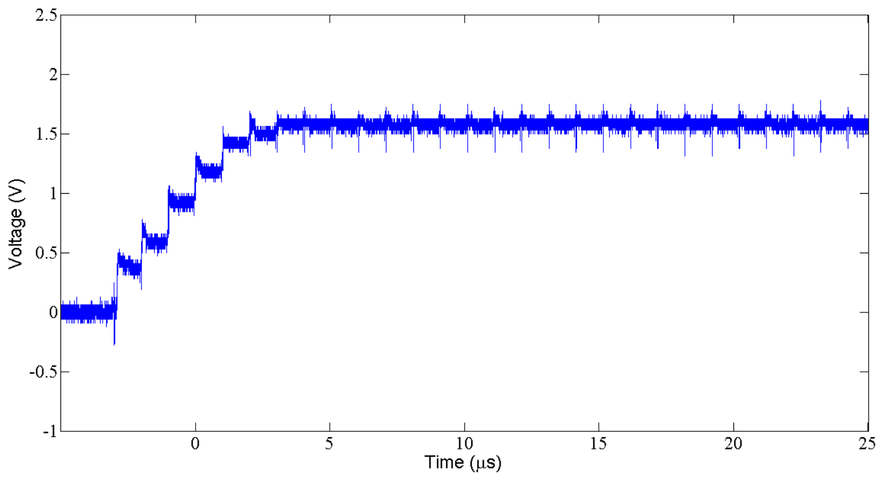


FIGURE 4.23: Delay control voltage V_p convergence at power-up.

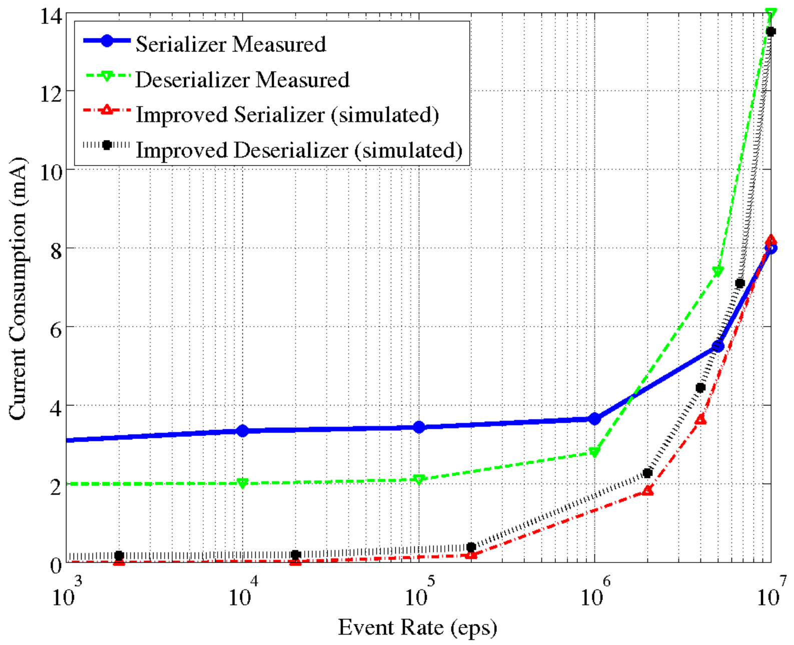


FIGURE 4.24: Serializer and Deserializer current consumption as a function of event rate.

However, this circuit can be resized for lower power while satisfying speed requirements. By introducing these modifications, we re-simulated the Deserializer and obtained the current consumption shown in Fig. 4.24, which shows a remaining stand-by current consumption of $150\mu A$.

Also, the flash ADC is an over engineered solution (although it simplified the overall design), resulting in an excessive area consumption (see Fig. 4.18). In a final implementation, since the ADC does not require high speed and it only has one comparator ON at a time, the flash ADC would be substituted by a successive approximation version with just one comparator.

4.7 Conclusion

Along this Chapter, a SerDes circuit specially designed for asynchronous and event-driven AER systems has been described. The design does not require to keep the link active during absence of user data. The receiver includes a means for memorizing the tuned state during data pauses. This way, when a new event is transmitted, the communication is reestablished without information lost. As a result, the power consumption of the SerDes circuit is proportional to data event rate. The proof of concept test prototype has been fabricated in $0.35\mu m$ CMOS technology and is capable of achieving a maximum event transmission rate of 15Meps with 32-bit events. The system is jitter tolerant and does not require a very low jitter clock. The proposed architecture uses simple components which do not require critical matching, jitter, nor supply voltages.

As it was sketched in the previous Section, the SerDes circuit allows further improvements in terms of power and area efficiency. However, the main limiting factor is the conventional LVDS driver/receiver circuitry used for the high speed differential signal transmission. The static current consumption of these circuits is around 17mA (driver+receiver) each one and they remain active in pauses. The event driven SerDes power consumption reduction vanishes when it is combined with such as circuits. For this reason, the next Chapter will be focused on developing power consumption reduction design techniques for current mode LVDS circuits exploiting the sparse nature of AER data. This feature must be introduced in the link without introducing extra latency in the event transmission or limiting the maximum event rate.

5

Current Mode Switchable I/O Circuitry for Low Power Serial Transmission of AER Streams

5.1 Introduction

The event-driven SerDes architecture proposed in Chapter 4 overcomes the power limitations of conventional high speed serial links approaches for sparse events transmission. Clock is embedded in the data flow but the CDR circuit does not need any comma transmission when there are no user data. Synchronization information is digitally stored at the receiver side and updated with every new event. Communication can be restarted immediately, avoiding the need for long preambles or loss of information. Only one driver/receiver pair is needed to implement the communication. A pair of digital request/acknowledge signals are sent along with the high speed AER data for flow control purposes, using a 4-phase handshaking protocol.

The asynchronous SerDes power consumption scales down with event rate, as it can be verified in the experimental results shown in Chapter 4. This is not the case with the high speed LVDS (Low Voltage Differential Signalling) [109] driver/receiver pair [108] used in this primary implementation. These circuits have a large static power consumption which remains even if there are no data being transmitted. As serial lines must operate at any known logic value during pauses, a mA range current flows through the termination resistor. Several authors have addressed the problem of reducing the LVDS driver power consumption [116]-[117]. The techniques proposed range from low voltage implementations of current mode drivers which reduce the number of stacked elements [116], to low input capacitance driver realizations to save power in the pre-driver stage [118] or even voltage mode circuits in BiCMOS [117]-[119] or CMOS [120] technologies. However, all these solutions are focused on dynamic power reduction and still suffer from high static power dissipation.

Introducing techniques to reduce the static power consumption in pauses is essential to increase the serial AER link power efficiency. Quickly turning off the LVDS driver/receiver bias currents which are responsible for static power consumption seems to be the easiest way to achieve this purpose. This way, the driver/receiver pair will be switched OFF in the presence of pauses and turned back ON quickly when new events are ready to be transmitted. A fast start-up link is mandatory because recovery time directly impacts overall link latency and reduces the maximum achievable throughput. A reasonable limit for the additional latency introduced by the switching mechanisms described in this Chapter is set to 1 bit time. This way, the power saving mechanisms would not trade-off with the maximum link throughput.

A regular push/pull current mode LVDS driver circuit and a conventional receiver have been used to implement the design techniques needed to quickly switch off the high speed serial circuitry. In this Chapter, the current mode approach has been adopted rather than the voltage mode approach because it provides robust impedance matching and higher immunity to power supply noise. Voltage mode circuits can achieve similar (or even higher) precision in impedance matching, but to do so they need extra calibration circuitry. These extra circuits contribute to increasing the driver power consumption and introduce an area penalty. In principle, therefore, current mode drivers ensure a better signal integrity and higher data rates for an identical channel than their voltage mode counterparts. On the other hand, CML (Current Mode Logic) [121] or LVPECL (Low-Voltage Positive Emitter-Coupled Logic) [122] drivers are simpler to design as they do not require common-mode adjustment circuitry. However, when turning them OFF and back ON quickly, it is necessary to keep their common mode for fast recovery, thus making the design more complex. As the technique proposed is based on efficiently switching large current sources ON/OFF, the solution presented in this Chapter for the push/pull LVDS driver can be directly extended to other current mode standards.

This Chapter discusses the circuit level implementation of a switchable current mode link suitable to efficiently transmit AER information. First of all, advantages of switchable high speed serial links when transmitting asynchronous data will be presented, as well as the required modifications in the AER protocol to handle the link switching mechanism. After these system level design considerations, the design techniques employed to efficiently switch on/off the driver and receiver circuits will be described. Extensive experimental results of a $0.35\mu m$ CMOS prototype of the current mode circuitry integrated with the event-driven SerDes are presented to demonstrate the effectiveness of the approach.

5.2 Switchable high speed serial links

Fig. 5.1 shows the desired waveforms for the supply current and differential mode voltage of switchable high speed serial links. Instantaneous current consumption switches from a high current in the order of mA when circuits are transmitting data to a low

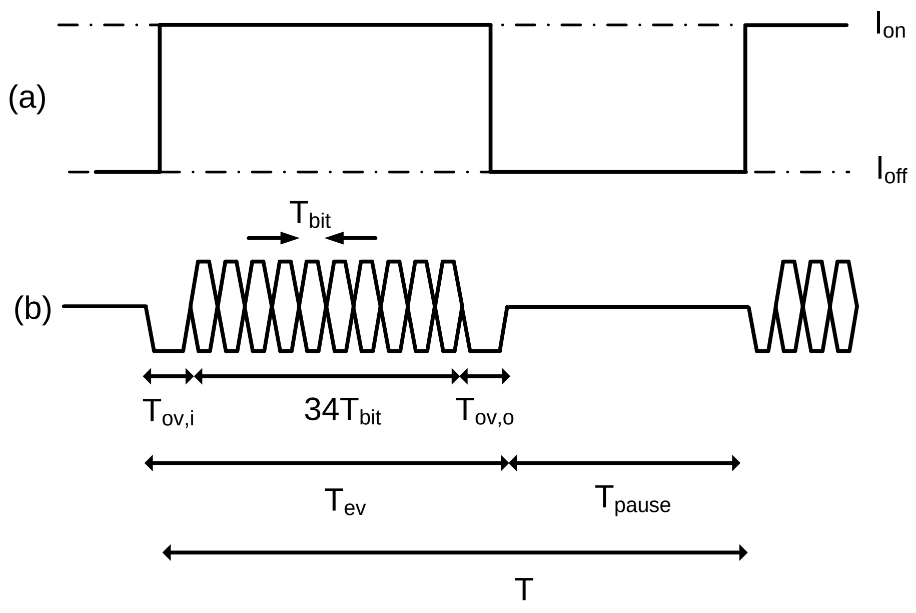


Figure 5.1: (a) Desired waveform of the supply current drawn by the switchable driver or receiver when an 1/T event rate is being transmitted. (b) Differential mode voltage of the serial data for this current consumption pattern.

current in the order of hundreds of μA in pauses. Transitions between both states must be very sharp to keep a high throughput and low latency in the link. Let T_{ev} be the time required by the SerDes scheme to transmit a single event and T_p the time that the link remains idle between two consecutive transmissions. The mean current consumption for an event flow with an average 1/T event rate is

$$\bar{I}_{switched} = I_{ON}\frac{T_{ev}}{T_{ev}+T_p} + I_{OFF}\frac{T_p}{T_{ev}+T_p} \tag{5.1}$$

A conventional current mode driver is not switched OFF when a pause is detected. This leads to a constant current consumption independent of the event rate and given by $\bar{I}_{conv} = I_{ON}$. Let us define a figure of merit for the power saving effect of the switchable link as

$$\alpha = \frac{\bar{I}_{switched}}{\bar{I}_{conv}} = \frac{I_{OFF}}{I_{ON}} + \frac{T_{ev}}{T}\left(1 - \frac{I_{OFF}}{I_{ON}}\right) \tag{5.2}$$

where $T = T_{ev}+T_p$ is the time difference between two consecutive events ($min\,(T) = T_{ev}$ for $T_p = 0$). Case $\alpha = 1$ corresponds to a conventional current mode link without current switching. The lower the value for α, the higher the power saving effect achieved. The ratio between the ON/OFF currents depends on the driver/receiver circuit implementation. The second term of eq. (5.2) introduces the dependence of α with event

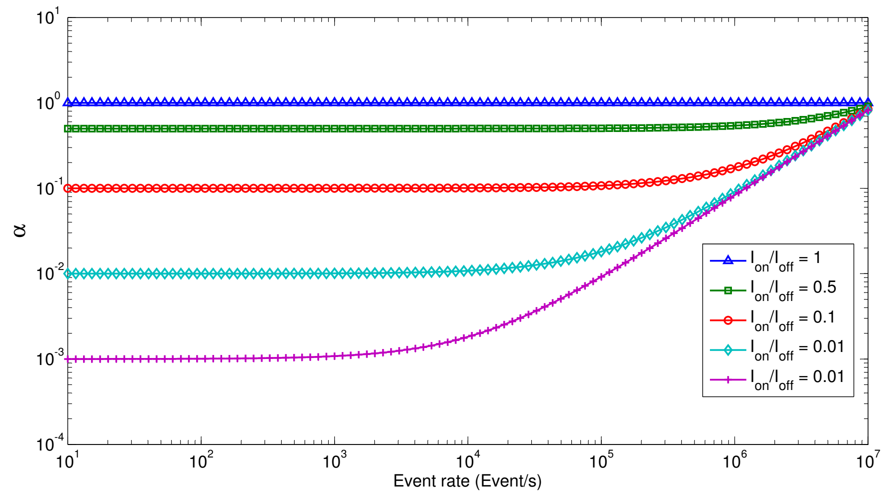


Figure 5.2: Figure of merit α vs event rate parametrized by the on/off current ratio.

rate. Fig. 5.2 shows a family of curves that plot α versus event rate for different ON/OFF current ratios. For very low event rates, α saturates to its minimum value, determined by the relation between ON and OFF currents. For medium and high data rates, the figure of merit decreases linearly with data rate. The I_{OFF}/I_{ON} ratio is the basic design variable for optimizing serial link energy efficiency.

5.3 AER protocol modification

ON/OFF switchable LVDS driver and receiver pad pairs need to handle extra signals to control the ON/OFF switching. In particular, the sender takes the decision to turn OFF the driver when there is no data to transmit. When a pause occurs, the differential output lines are made to evolve to the same common-mode voltage, as there is no current flowing through the termination resistor. In principle, we could take advantage of the serial handshaking signals to tell the driver and receiver pads whether a bit-serial events is being transmitted or not. However, the handshaking signals used in circuits presented in Chapter 4, only indicate the start of the bit-serial transmission and not its duration. A modification to the traditional AER protocol is therefore required to provide the timing information to detect pauses, switch OFF bias currents, force a constant output value, and freeze the deserializer.

Fig. 5.3 illustrates the changes introduced in the SerDes scheme to handle the switching mechanism. The solution proposed in Chapter 4 is presented in Fig. 5.3(a) for comparison purposes. All blocks in black do not need any modification and they work the same way it was described in Chapter 4. Only the physical LVDS driver/receiver pair is changed and the block "*ackSergen*" is added to manage the serial handshaking signals generation. These modifications are highlighted with red color in Fig. 5.3(b).

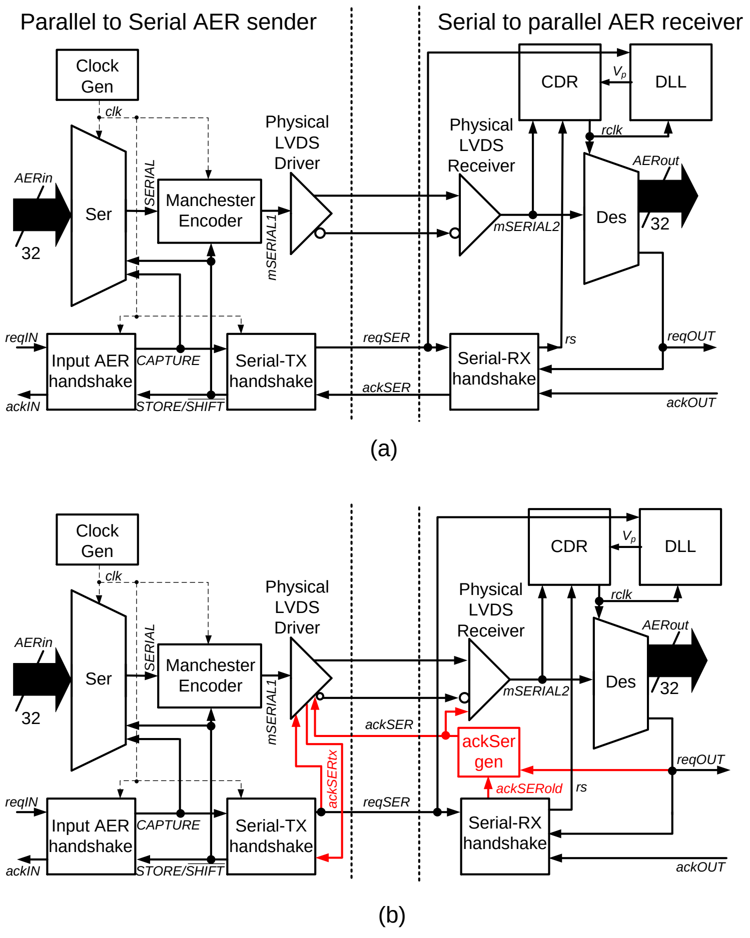


FIGURE 5.3: Block Diagram of the bit serial AER LVDS link (a) already reported implementation [123] (b) modification to handle the ON/OFF switching mechanism. Changes from one implementation to another are highlighted.

Fig. 5.4(a) presents the AER protocol waveforms for the burst mode SerDes implementation of Chapter 4. In this case, the transmitter sets a low level in *reqSER* when new data is ready to transmit. If the receiver is ready to handle a new event, it returns the acknowledge *ackSER*, enabling the transmission. Then *reqSER* returns to a high level, the transmitter considers that handshaking transaction completed and the receiver sets *ackSER* to high level while the bit-serial data transmission continues until it finishes.

The protocol modification proposed is shown in Fig. 5.4(b). Keeping *ackSER* at low level until bit-serial data transmission has finished provides a proper timing reference for the driver/receiver to switch ON/OFF. This way, the driver and receiver get activated when *reqSER* and *ackSER* have a falling edge, respectively, and are disabled when a rising edge occurs at *ackSER*. The deserializer generates a pulse in *reqOUT* when the event has been completely decoded and deserialized. Signal *ackSER*

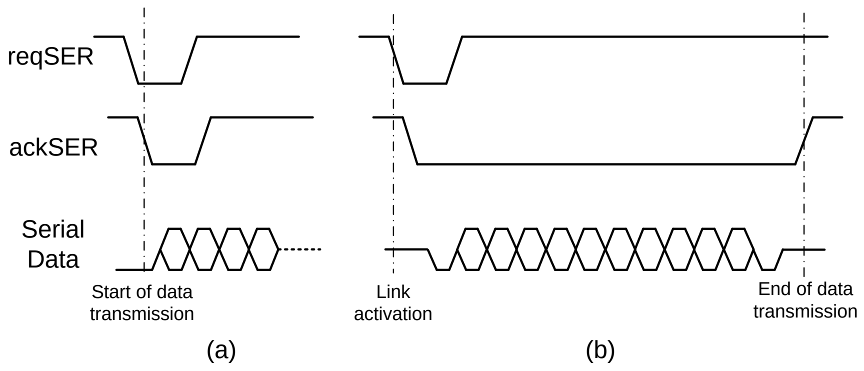


Figure 5.4: (a) AER protocol used in Chapter 4 (b) Modification for the switchable version.

is deactivated afterwards to switch OFF the link until the arrival of the next event.

Fig. 5.3(b) shows the modifications introduced to handle the AER protocol modification. Signal *ackSER* is generated by the new block *ackSergen*, which receives signal *reqSER* and the original acknowledge generated by the *Serial-RX handshake* block *ackSERold* when the receiver can accept a new event. A falling edge at *reqSER* activates the transmitter and the serial outputs evolve to a known logic value (zero in this case). After a propagation delay, the request signal *reqSER* arrives at the receiver side, which is also turned ON if it has resources to receive new data. One constraint is that the high speed bit-serial signals must be stable at their zero value when the receiver is activated. The propagation delay of the *reqSER* path must therefore be longer than that of the high speed serial path. Signals *reqSER* and *ackSER* must be propagated along two standard digital pads (one in the transmitter chip and the other in the receiver chip). If the driver is designed to recover from pauses with sub-bit delays, the communication can be established without asynchronous hazards.

The receiver generates *ackSER* using the simple circuit shown in Fig. 5.5. The deserializer generates a pulse in *ackSERold* when it detects that *reqSER* is at low level and a new event can be processed. This pulse generates a reset for the falling edge triggered flip-flop shown in Fig. 5.5, causing the activation of *ackSER*. This signal is sent back to the transmitter and stays activated until a falling edge at *reqOUT* occurs. At this point, the flip-flop is triggered and captures its input, which is tied to the voltage supply. This makes *ackSER* return to high level and the link is turned OFF, waiting for the next event.

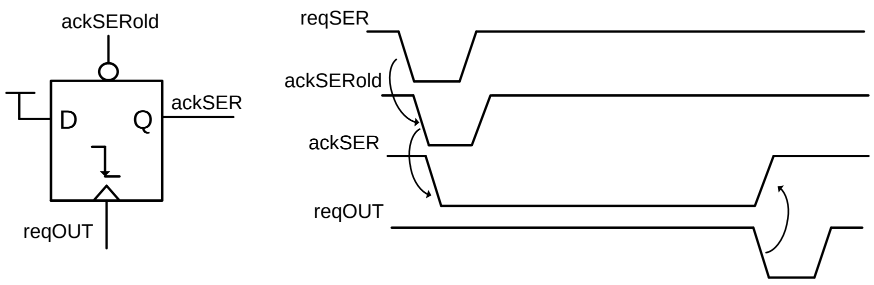


Figure 5.5: Circuit that generates *ackSER* active throughout the event transmission at the receiver side, labeled "*ackSergen*" in Fig. 5.3(b).

5.4 Driver circuit

A typical push pull current mode transmitter acts as a current source with switched polarity [108]. Output current flows through the load resistance, establishing the correct differential output voltage swing. MOS switches are used to change the output current polarity depending on the transmitted logic level. Output common mode is controlled by a feedback loop which senses it, compares it to a reference and acts over the current sources. The high precision voltage reference is generated by a bandgap circuit.

The main power consumption contribution in a current mode driver comes from current sources. If a 350mV differential amplitude is desired and the load resistor is typically 50Ω (two parallel 100Ω resistors located in the driver and receiver), current sources of 7mA are needed. These current sources can be switched OFF during pauses to save power. However, two design considerations must be taken into account to achieve real burst mode operation:

- When the driver is switched OFF, output common mode must be retained until the next transmission. If common-mode information is lost, it will take a longer time to recover.
- The ON/OFF switching mechanism must be fast enough not to slow down the event transmission rate. That is why very fast, low noise switchable current sources are required to implement an efficient switchable link.

Fig. 5.6 shows the schematic of the proposed switchable current mode driver. Driver input signals are the single ended bit-serial stream *dataSer* and the serial AER protocol signals *reqSER* and *ackSER*. The "ON/OFF Controller" generates the differential version of *dataSer*, labeled *INVOA* and *INVOB*, for the NMOS switches. It also generates and drives the enable signal *enTX* which is active during event transmission to switch ON the driver. This signal is the result of a NAND operation between *reqSER* and *ackSER*. When the driver is enabled, bias voltages *PBIAS* and *NBIAS* are driven by the CMFB (Common Mode Feedback) circuit. In this configuration, the driver works in the same way as the classic non-switchable one [108] and output common mode is controlled by the CMFB circuit. When the driver is disabled, voltages *PBIAS* and

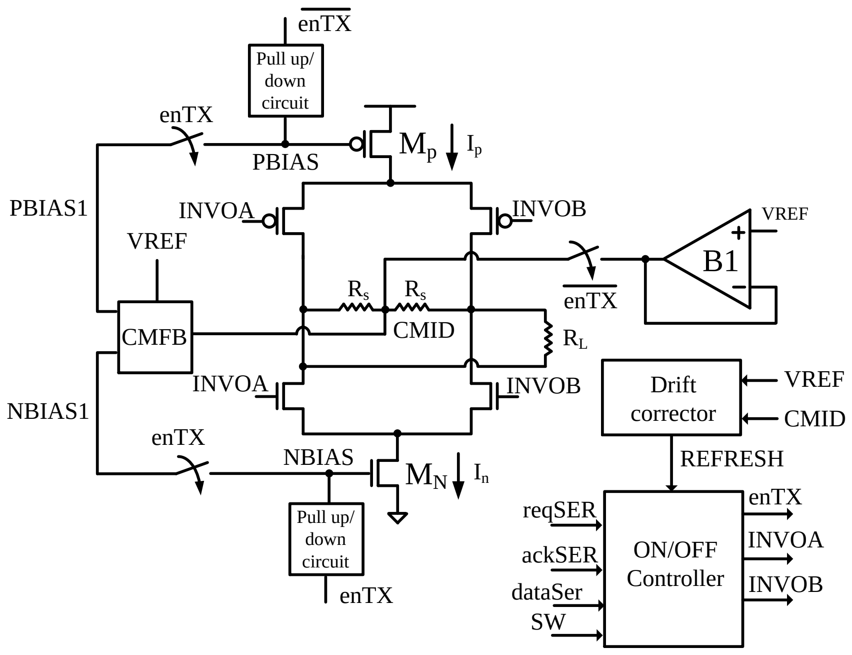


FIGURE 5.6: Proposed switchable current mode driver.

NBIAS are pulled up or down switching OFF the current sources.

Resistors $R_s = R_L/2 \approx 50\Omega$ are used to terminate the transmission line impedance. A double termination scheme with resistors at both transmitter and receiver sides is used to improve impedance matching and reduce reflections which may deteriorate signal integrity. The output common voltage is sensed by splitting the resistor $2R_s$ into two elements. When the driver is active, block CMFB closes the loop and sets the bandgap voltage *VREF* as output common mode. During pauses, a voltage buffer *B1* directly forces *VREF*, but not through the feedback loop. As output currents are very low during pauses, the buffer response time can be designed in the order of ms. Hence, the buffer can be designed to have a very low static power consumption. The CMFB circuit is always driven by a constant voltage at its input. Information contained in *PBIAS* and *NBIAS* about the output common mode and the differential amplitude is therefore maintained during pauses and immediately restored in the transmission mode.

Fig. 5.7 shows the "ON/OFF Controller" schematics. This block generates the *enTX* signal (Fig. 5.7(a)) when *reqSER* or *ackSER* are active at low level indicating that a serial handshaking is being carried out. An intermediate level of logic disables the switching operation when configuration signal *SW* is taking place. Moreover, the "ON/OFF Controller" also adapts the rail-to-rail input signal to a differential format compensating the lag between *INVOA* and *INVOB* with a dummy transmission gate, as shown in Fig. 5.7(b). Tapered buffers *B2* and *B3* make it possible to maintain a low input parasitic capacitance while providing enough driving capability to handle

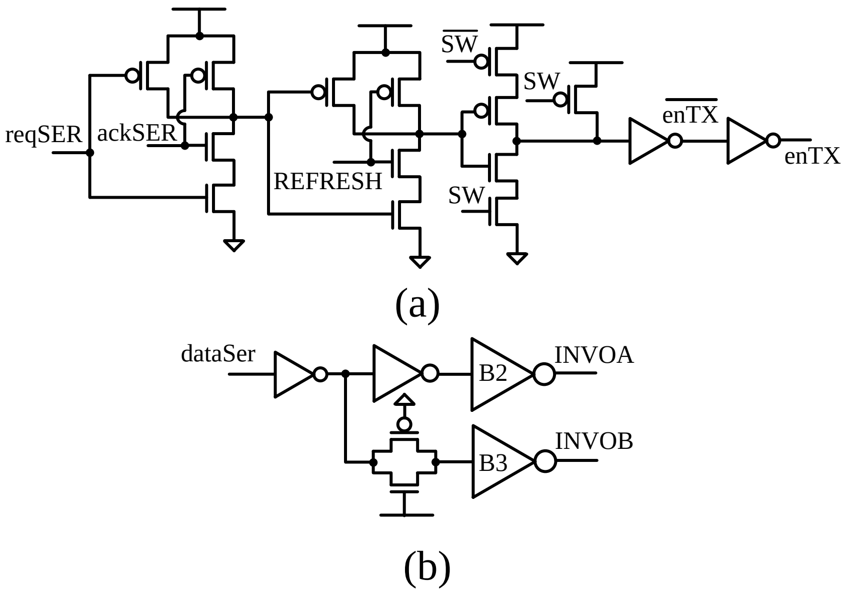


Figure 5.7: "ON/OFF Controller" schematics for (a) driver activation signal generation and (b) input rail-to-rail serial stream handling.

the LVDS output switches.

Fig. 5.8(a) shows the schematics of the CMFB circuit used in this design. The input differential pair compares a filtered version of the sensed common mode voltage *CMID* against reference *VREF*. Current imbalance generated by the comparator is mirrored to the driver current sources I_n and I_p with a gain K=100. If the feedback loop is stable, the steady state will converge to a common mode equal to reference *VREF*. The gain K is chosen high to keep the *CMFB* static current consumption much lower than in the high speed path. Passive components $R_c = 1.5K\Omega$ and $C_c = 9.6pF$ introduce a low frequency compensation pole that ensures common mode control loop stability. To achieve a differential amplitude at the driver load of V_{od}, the CMFB circuit bias current must be

$$I_{bias} = \frac{2V_{od}}{KR_L} \tag{5.3}$$

The passive pull up/down circuits used in the design and their basic operation are shown in Fig. 5.8(b) [116]. The dashed line component C_p represents the parasitic capacitor associated with the PMOS and NMOS transistor gate nodes. Design capacitors C_{pp} are used to pull the gate voltage of the transistors up/down, drastically reducing the transition times. When the driver is turned ON, gate voltages are imposed by the CMFB circuit (V_{ON} voltages). When a pause starts, signals *enTX* and *enTXnot* switch and there is charge redistribution between the parasitic capacitor C_p and the design capacitor C_{pp}. This causes a change in the gate voltage that can be computed

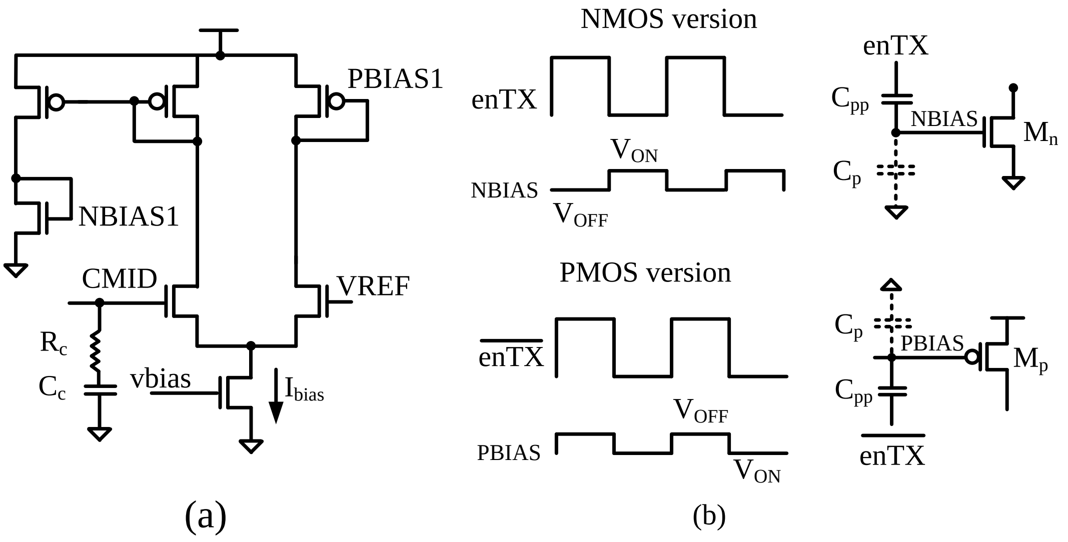


Figure 5.8: (a) CMFB circuit of Fig. 5.6 (b) Passive pull up/down circuits in Fig. 5.6

using the charge conservation equations before and after the switching

$$Q\left(t_{before}\right) = \left(C_p + C_{pp}\right) V_{ON} \tag{5.4}$$

$$Q\left(t_{after}\right) = C_p V_{OFF} + C_{pp}\left(V_{OFF} - V_{DD}\right) \tag{5.5}$$

By applying the charge conservation principle ($Q\left(t_{before}\right) = Q\left(t_{after}\right)$) the voltage step is

$$\left|\Delta V\right| = \left|V_{ON} - V_{OFF}\right| = \frac{C_{pp}}{C_p + C_{pp}} V_{DD} \tag{5.6}$$

Voltage step $|\Delta V|$ can be adjusted through C_{pp} to control the OFF currents I_{OFF} at M_n and M_p preferably in the sub-threshold region to reduce static power. These OFF currents are added to the bias currents of auxiliary blocks I_{aux} as static power consumption during pauses, but the power efficiency is not compromised if these OFF currents are one order of magnitude below I_{aux}. However, if the voltage step between ON and OFF situations is reduced, the ON/OFF turning mechanism is faster and does not suffer from current peaking. The trade-off between the turning ON speed (T_{rec}) and idle current consumption (I_{OFF}) is shown in Fig. 5.9, versus C_{pp} for different technology corners, obtained through simulation . We chose a 1.3pF for the PMOS current source because no significant variation in I_{OFF} can be observed for larger C_{pp}, but speed is degraded as C_{pp} increases. The design capacitor for the NMOS counterpart, determined following the same design procedure, is 0.5pF.

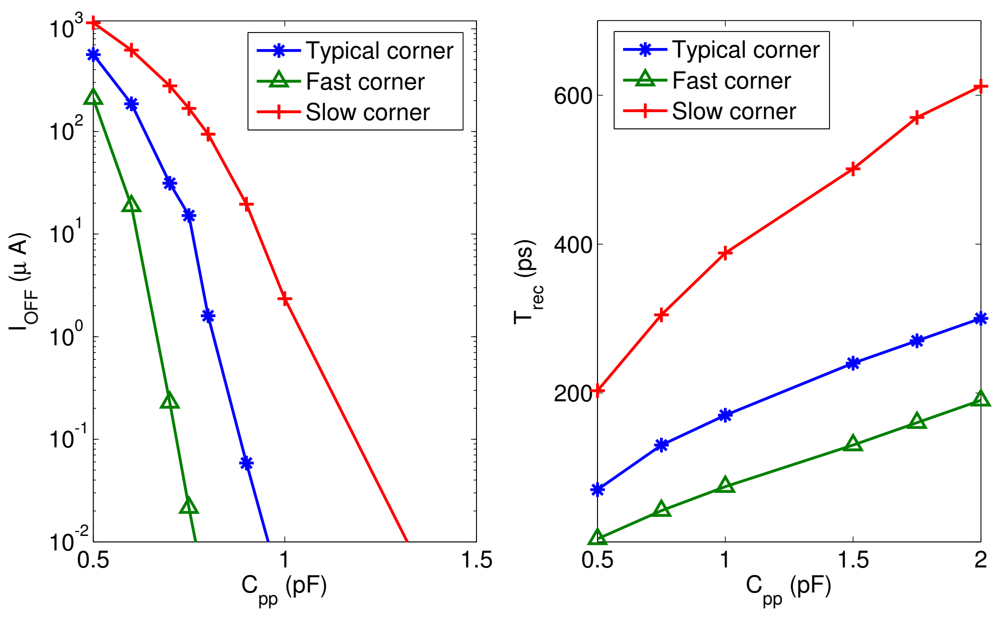


Figure 5.9: Trade-off between switching time after pauses and current consumption when the driver is turned OFF in the PMOS current source. C_{pp} is the design variable in both cases. Typical, fast and slow corners are represented.

During pauses, gate voltages *NBIAS* and *PBIAS* in Fig. 5.6 and Fig. 5.8(b) are left floating and suffer drift through leakage currents. This can make the OFF currents at M_N and M_P becoming excessively large, increasing static power and making it difficult for buffer *B1* to keep the common mode voltage *VREF* at *CMID* (because of mismatch between OFF currents at M_N and M_P). To avoid such situations the “Drift Corrector” block (see Fig. 5.6) detailed in Fig. 5.10 is added. It is a window comparator composed of two asymmetric comparators and an XNOR gate.

The circuit generates an activation pulse blue at signal *REFRESH* (see Fig. 5.6) whenever the output common mode leaves the comparator window ΔV_{comp}. This pulse will cause another one in the *enTX* signal to activate the driver. The pull up/down circuit will refresh the gate voltages at M_N and M_P driving them to their OFF currents. The driver will remain activated until the output common mode enters the comparison window.

5.5 Receiver circuit

The conventional LVDS receiver [108] uses a current to voltage conversion with a termination resistor. This voltage is processed by a continuous time preamplifier and a rail to rail comparator to provide a proper output for the deserializer circuit, as shown in Fig. 5.11. The last stage is usually a hysteresis comparator to avoid output glitches due to input noise. The implemented hysteresis comparator, shown in Fig. 5.11(c),

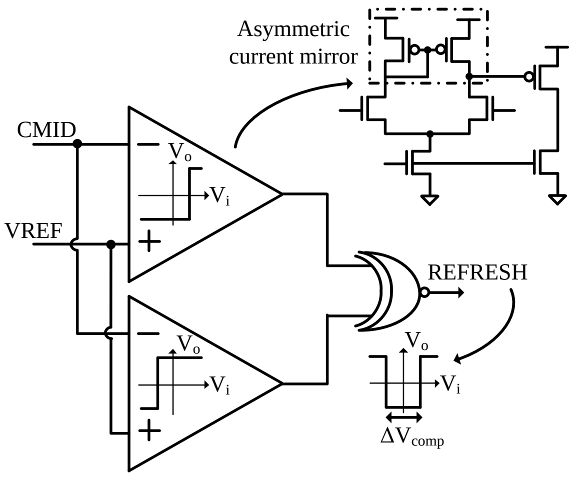


FIGURE 5.10: "Drift Corrector" circuit in Fig. 5.6 that detects drifts in the output common mode during pauses.

uses a positive feedback loop to achieve enough gain and speed to convert the low amplitude signal into a rail-to-rail signal. Hysteresis range can be controlled by the aspect ratios of the PMOS transistors as:

$$\Delta V_{hys} = \frac{2\left(\sqrt{\alpha}-1\right)}{\sqrt{1+\alpha}}\sqrt{\frac{2I_b}{\mu_n C_{ox}\left(W/L\right)_{IN}}} \tag{5.7}$$

where $\alpha = (W/L)_5/(W/L)_3$. A $\Delta V_{hys} = 130\text{mV}$ was chosen in this design.

However, the input common-mode range of this hysteresis comparator is limited, to keep the differential pair and feedback loop transistors in saturation. A preamplifier can be added at the comparator input to increase the common-mode range and relax its specifications, as shown in Fig. 5.11(a). Fig. 5.11(b) shows the preamplifier circuit. Its folded-cascode configuration enables a large gain-bandwidth product independent of the input common mode. The PMOS input differential pair output is set to a low DC voltage via the folded cascode structure to increase the input common mode range. Moreover, the internal node of the cascode configuration provides a low impedance output node which pushes the internal pole to high frequencies. The output stage load resistors determine the gain, output common-mode and bandwidth of the pre-amplifier,

$$V_{out,CM} = V_{DD} - R\left(I_{b2} - \frac{I_{b1}}{2}\right) \tag{5.8}$$

$$Gain = -g_{m,IN}R \tag{5.9}$$

The use of a wide input common mode range preamplifier allows lower aspect ratio devices for the comparator input stage and preamplifier load capacitance can be minimized. This reduces the resistors' impact on the frequency response, pushing the

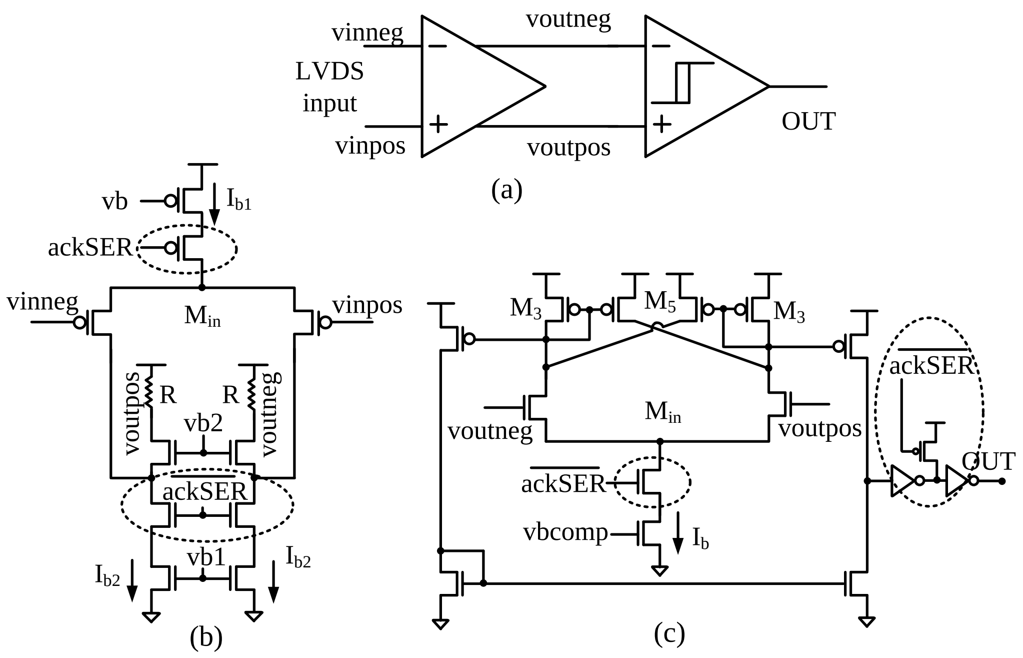


Figure 5.11: (a) Receiver architecture (b) Preamplifier (c) Hysteresis comparator

output pole to higher frequencies. For this design, with $I_{b1} = 2.2mA$ and $I_{b2} = 2mA$, a 1.2 $K\Omega$ resistor was chosen.

The receiver power consumption is mainly due to the bias currents of cascaded blocks in Fig. 5.11(a). These two circuits are not needed during the pauses if the comparator output is forced to have a constant value (zero in this design). Bias currents can therefore be disabled during pauses. A low level in the *ackSER* signal switches the receiver ON. The switches encircled with dashed lines in Fig. 5.11 are used to turn OFF the bias currents during pauses. These switches have been designed to enable very fast transitions of relatively high bias currents while maintaining a low ON resistance. A large current passes through these switches during regular operation and the voltage drop can affect the tail current source saturation.

5.6 Experimental results

A proof of concept prototype of the switchable serial AER link has been fabricated in 3.3V 0.35μm CMOS. All the components described in this Chapter were integrated as custom-made pads with ESD protections, ground and supply voltage decoupling capacitors and the analog circuit biasing resources. The current mode driver pad needs a 370x395μm^2 area, while the receiver pad area is 270x340μm^2. Fig. 5.12 shows a microphotograph of the test chip where the main parts have been highlighted.

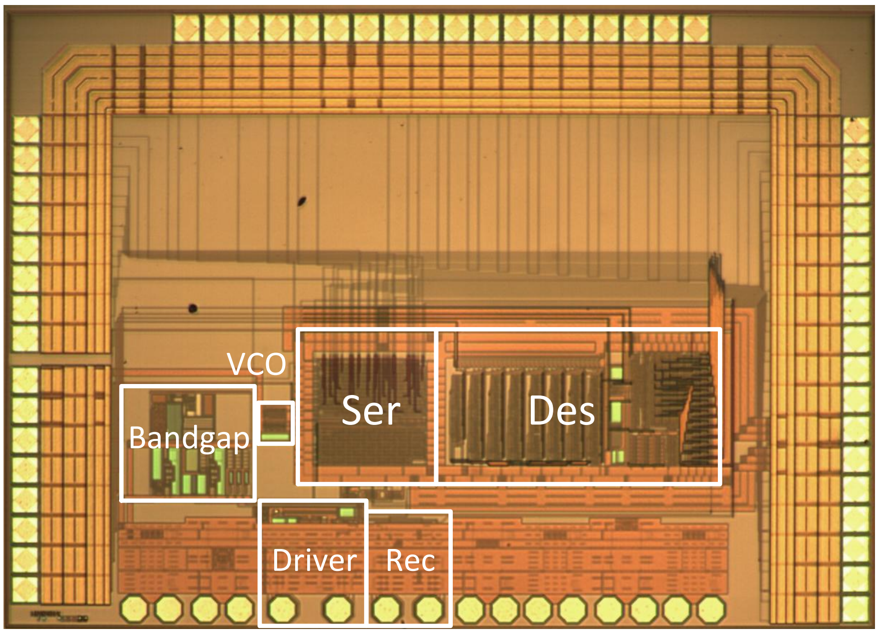
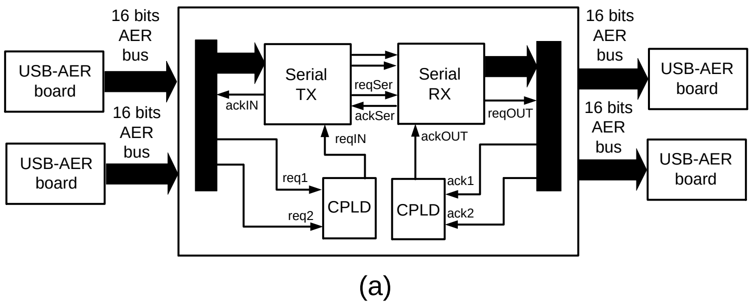


Figure 5.12: Microphotograph of fabricated test prototype.

The bandgap circuit is used to generate a stable, PVT independent common mode voltage and bias currents. A stand-alone VCO (Voltage Controlled Oscillator) is used to generate the master transmission frequency in a programmable way. The test channel used in the experiments comprised a pair of PCB traces forming a 100Ω differential microstrip line of 5cm length. The test set-up used in the experiments is identical to that of the event-driven SerDes described in Chapter 4 and it is repeated here in Fig. 5.13(a) for convenience. A photograph of this set-up is shown in Fig. 5.13(b).

Fig. 5.14 presents the measured AER handshake protocol signals at (a) the transmitter parallel input (*reqIN*, *ackIN*), (b) the serial interface (*reqSER*, *ackSER*) and (c) the receiver parallel output (*reqOUT*, *ackOUT*). When a new 32-bit event is ready to be transmitted, the CPLD receives the request signals from the USB-AER boards and activates *reqIN*. This action indicates to the serial AER transmitter that it must latch the event and respond with an acknowledge *ackIN* to the USB-AER sender boards. The delay between *reqIN* and *ackIN* is 15ns for a 500MHz transmission master clock. The *ackIN* remains active during 68ns, corresponding to the 34 clock cycles required for the serialization process [123]. While the channel is busy with an event, the transmitter stops the sender side by keeping *ackIN* at low level.

Serial AER signals *reqSER* and *ackSER* are used to provide the timing information to switch the high speed serial driver and receiver ON and OFF. Before starting with a new event serial transmission, *reqSER* is activated and the receiver acknowledges with *ackSER* if it can handle a new event. The delay in the serial handshaking protocol



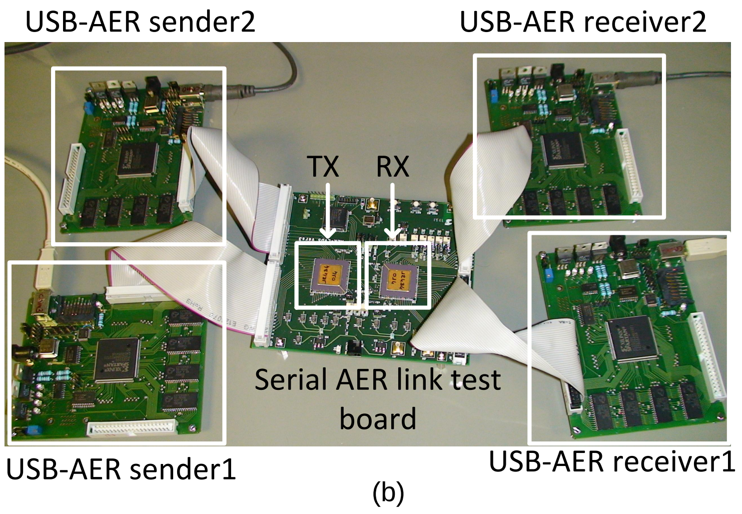


Figure 5.13: Test set-up to generate a 32 bit AER pattern.

is 6.6ns. The main difference between this implementation and the one described in Chapter 4 is the duration of the *ackSER* activation. In this switchable implementation, *ackSER* is kept active until the whole event bit data have been transmitted: 82ns for the 500Mbps bit rate. When the event is properly recovered from the input serial stream, the output AER interface communicates it to the receiving USB-AER boards. The 24ns delay between signals *reqOUT* and *ackOUT* is due to the USB-AER board logic and the extra latency introduced by the CPLD operation.

Fig. 5.15 shows the measured differential mode of the serial signal at the receiver input. An Agilent DSO81304B Infinium oscilloscope with 5GHz bandwidth probes was used for this measurement. The bit rate was set to 500Mbps and the event rate was configured to be 4.7Mevent/s. The ON and OFF switching times are 1ns (less than the bit time). Compared to the duration of the whole event transmission, these times are negligible and do not compromise the serial link latency or maximum event rate. The differential amplitude obtained is 350mV. Fig. 5.16 shows the eye diagram for the Manchester encoded stream measured at the receiver input. A 43ps of rms jitter was measured, very similar to the values measured in Chapter 4 where a non-switchable driver [108] was used. This suggests that the introduction of the switching mechanism

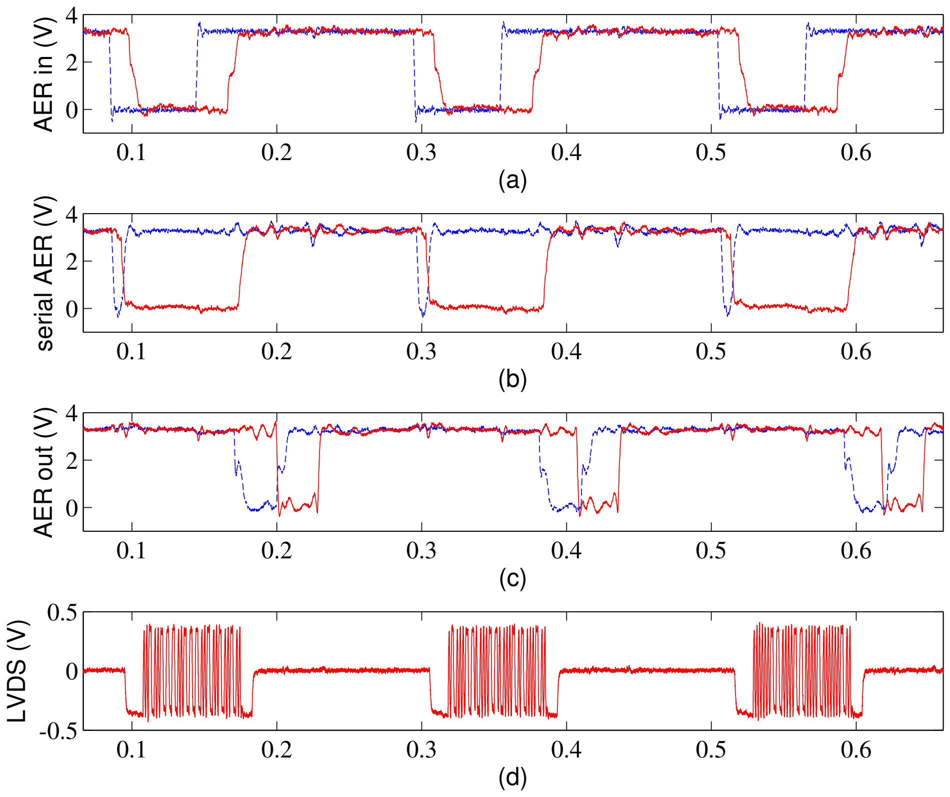


Figure 5.14: AER protocol management at (a) *reqIN* (dotted line) and *ackIN* (continuous line) signals (b) *reqSER* (dotted line) and *ackSER* (continuous line) signals (c) *reqOUT* (dotted line) and *ackOUT* (continuous line) signals (d) differential mode of the LVDS signal. Time scale is μs

does not limit the jitter of the whole link.

The maximum event rate can be obtained by shorting the *reqOUT* and *ackOUT* signals and measuring the input to output request delay. For the 500MHz clock frequency, the maximum achievable event rate is 11Mevent/s, corresponding to a 92ns latency. The link operation tolerates a wide range of transmission frequencies, continuously tunable using the on-chip VCO. For the maximum frequency operation, corresponding to a 710MHz master clock, a 66ns input to output request latency was measured. This leads to a maximum event rate of 15Mevent/s. This is also the maximum event rate we reported previously [123] using the non-switchable driver [108]. Consequently, using the presented switchable driver/receiver pair does not introduce any delay penalty, but does introduce important rate-dependent power savings.

Fig. 5.17 shows how the power consumption of the switchable link scales down with event rate. The high speed circuitry can be programmed to switch the bias currents or not, thus allowing a fair comparison between both situations. If the switching mechanism is deactivated, a roughly constant driver current consumption of 7.5mA is obtained for all the event rates. In the receiver side, current consumption ranges

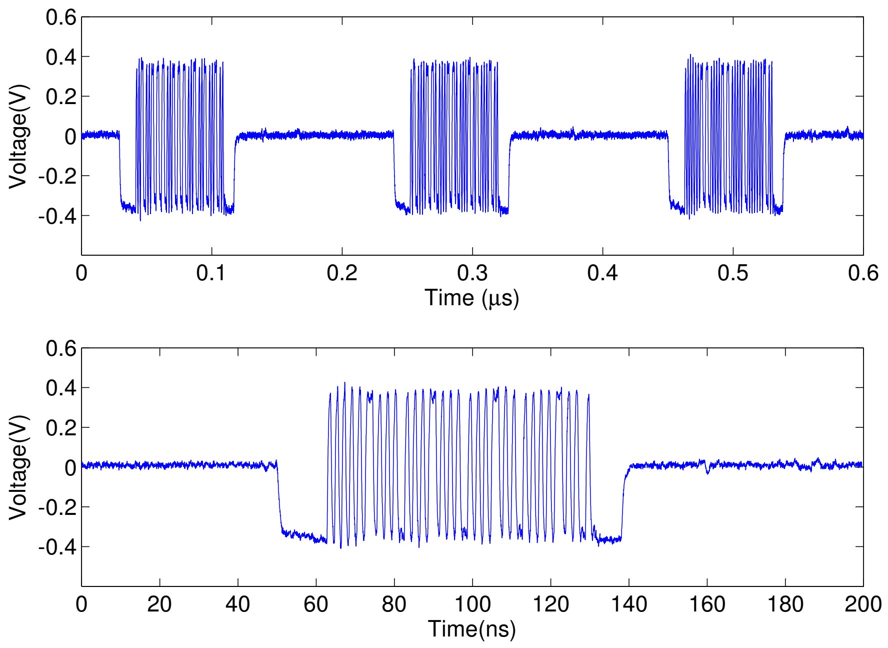


Figure 5.15: Measured high speed serial signal at the receiver input for the current-mode driver with two different horizontal time scales.

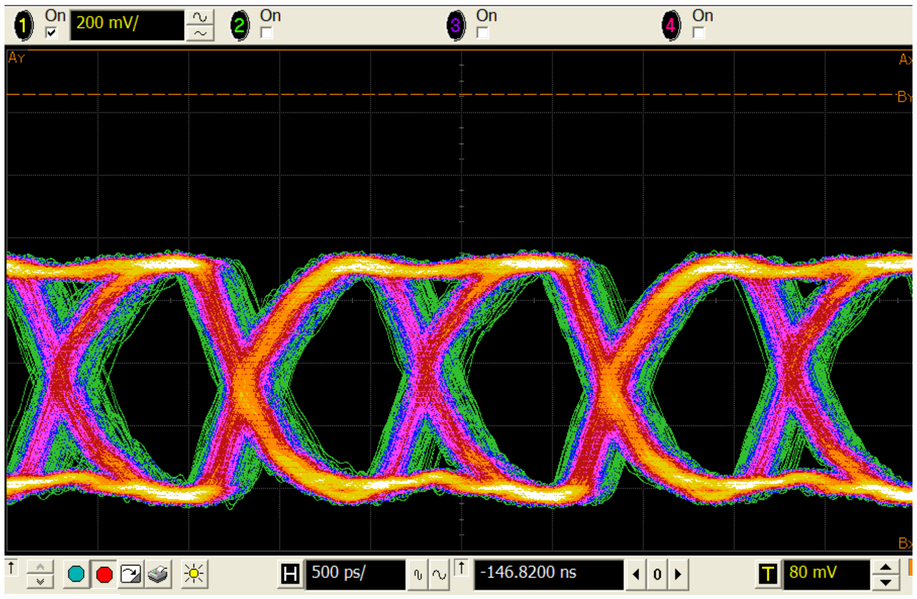


Figure 5.16: Eye diagram for the current-mode driver at receiver input.

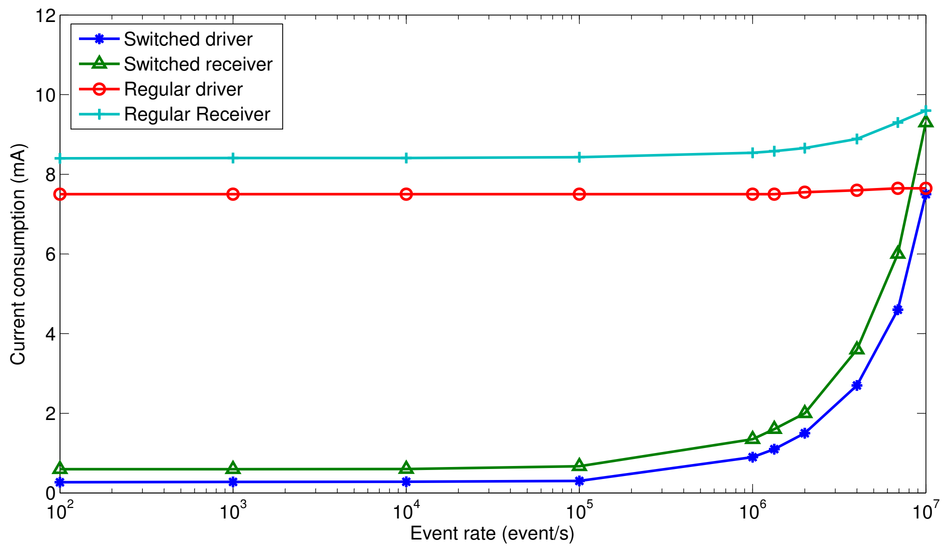


Figure 5.17: Current-mode switchable driver and receiver current consumption versus event rate.

from 9.6 to 8.4mA for the maximum and minimum event rate situations, respectively. When the switching mechanism is activated, the maximum receiver and driver power consumptions are almost the same because the circuits stay turned ON for most of the time. However, current consumption decreases with event rate. The driver and receiver current consumption curves saturate at $270\mu A$ and $570\mu A$, respectively. These values correspond to the quiescent current consumption of the I/O circuitry when the link is switched OFF. The receiver idle current is high due to the non-optimum biasing circuits for the preamplifier and hysteresis comparator. However, this current can be dramatically reduced if an optimized low power biasing circuit is used for the receiver.

Testing the circuit that detects drifts in the output common mode during pauses can be cumbersome if we let the leakage currents charge/discharge the driver internal nodes. This approach will require several hours of observation and a sophisticated test set up. To speed up the common mode drift effect, we introduced extra spurious currents in the differential output lines using the set up shown in Fig. 5.18(a). The oscilloscope differential probe introduces a common mode resistor between the differential lines and ground which draws a current proportional to the output common mode. This current is larger than the common mode retain buffer bias. In this scenario, buffer *B1* (see Fig. 5.6) fails at controlling the common mode and the resulting dynamics can be observed in Fig. 5.18(b) by the action of the drift correction circuit of Fig. 5.10. When the common mode correction circuit detects the drift from the nominal value, it fires a refresh pulse and the driver is activated. As the CMFB circuit keeps the information about the steady state of the control loop, the output common mode is immediately recovered. When the driver is turned OFF again, the common mode drifts again. This effect allows us to test the common mode refresh circuit without waiting hours for

Table 5.1: Performance comparison with other LVDS drivers

	[117]	[116]		[108]	[118]	[120]		[119]	This work
Technology	0.35μm SiGe BiCMOS	0.35μm CMOS		0.35μm CMOS	0.18μm CMOS	0.18μm CMOS		SiGe BiCMOS	0.35μm CMOS
f_t (GHz)	50	15		15	120	120		100	15
Supply Voltage(V)	1.7-3.5	1.8	1.8	3.3	1.9	1.8	0.5	2.5	3.3
Diff Amp (mV)	265-300	340	340	412	200-430	450	250	400	350
Bit Rate (Gbps)	1-2	1.4	1.2	1.2	2.5	3.5	3	10	1.42
Bit Rate / f_t	0.02-0.04	0.09	0.08	0.08	0.02	0.03	0.025	0.025	0.095
Area (mm^2)	0.068	0.11	0.14	0.2	0.07	0.04	0.02	N/A	0.15
Area (Mλ^2)	2.22	3.59	4.57	2.61	2.47	3.58	3.10	N/A	3.10
$\bar{I}_{max}$ (mA)	6-7	12.8	7.1	13	2.5-4.8	7.6	2.5	6.25	7.5
$\bar{I}_{min}$ (mA)	6-7	12.8	7.1	13	2.5-4.8	7.6	2.5	6.25	0.27

current source gate voltage charging/discharging due to leakage currents. As can be seen the comparator window (see Fig. 5.10) has a value of $\Delta V_{comp} = 0.290V$ and the comparator response time is 153ns.

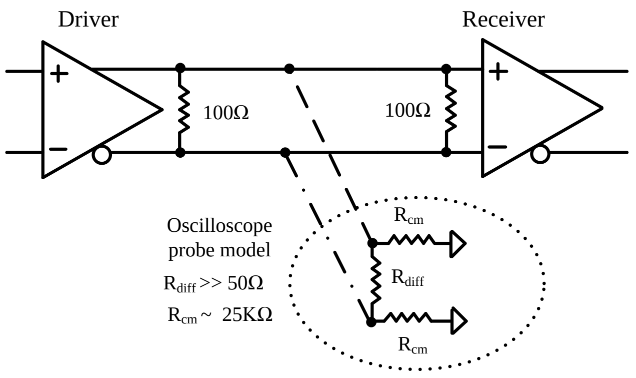
Table I shows a comparison with state of the art LVDS drivers. This implementation is comparable in terms of data rate, layout area and differential mode amplitude with similar works, but it is the only one which scales power consumption with event rate. $\bar{I}_{max}$ corresponds to the current consumption when the link is continuously transmitting data and $\bar{I}_{min}$ is for the minimum activity situation. The maximum physical bit rate (for $T = T_{ev}$) for this work is 1.42Gbps, which corresponds to twice the transmitted data bit rate of 0.71Gbps because of the Manchester encoding used in the SerDes circuits.[1]

5.7 Conclusion

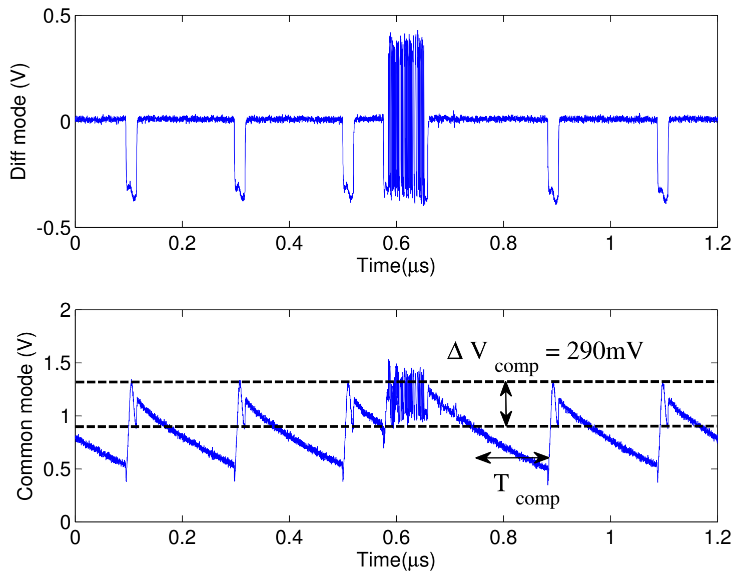
In this Chapter, we have described the design of a current mode switchable driver and receiver circuit pair intended to be used in high speed serial AER links. In this implementation, turning on/off the I/O circuits during pauses allows power consumption to be scaled with the transmitted event rate. A proof of concept prototype was fabricated in standard 0.35μm CMOS along with SerDes circuits discussed previously in Chapter 4. The driver realization achieves a factor of 30 in terms of current consumption saving compared with non-switchable solutions. The receiver current saving factor is 17, but this can be improved by reducing the static power consumption associated with the biasing circuits. Due to the high speed mechanisms applied to switch the driver/receiver current sources, the link is able to settle within its nominal values in 1ns. This way, the power saving is achieved without trading off against the link maximum event rate.

[1]References [108], [116], [118] and [120] in Table I do not disclose the f_t of the technology used. This number has been estimated from other similar technologies.

The static power consumption reduction is clear from the experimental results provided in this Chapter. The question from this point on is: can the power consumption in the transmission mode be further reduced to increase the link energy efficiency? The next Chapter addresses this problem by proposing a voltage mode approach for the I/O circuits. Due to their impedance termination scheme, voltage mode circuits enable a potential reduction in terms of power consumption compared to the current mode circuitry. The power saving effect in the transmission mode and switching mechanisms to disable the static current consumption in pauses could be combined to obtain a more power efficient implementation.



(a)



(b)

FIGURE 5.18: (a) Load model for the oscilloscope differential probe (b) Output common mode and differential mode waveforms when the driver is loaded with probe.

6 Voltage Mode Switchable I/O Circuitry for Low Power Serial Transmission of AER Streams

6.1 Introduction

As it was shown in the previous Chapter, switching off the high speed circuitry during pauses is an efficient power saving technique for asynchronous AER serial links. A current mode implementation of the switchable link was discussed with a push/pull driver example. Although the current mode approach is the preferred by the industry, there are other techniques than, in certain environments and transmission channels, can be more power efficient. Voltage mode implementations, where there are no large current sources which must be switched on/off with sharp rising and falling edges, are a very suitable option to further improve the energy efficiency of the serial AER link.

A voltage mode implementation of the circuitry presented in Chapter 5 will be the matter of the subsequent sections. First of all, advantages and disadvantages of a voltage mode realization compared to a current mode realization will be discussed. In this first Section, the range of applications for each solution in the framework of large scale spiking neural networks will be analyzed. The next Section will be devoted to the link design techniques developed to obtain the desired behavior, focusing on the impedance matching and the power management. Experimental results of a $0.35\mu m$ CMOS prototype of the voltage mode circuitry integrated with the event-driven SerDes described in Chapter 4 are also provided to validate the proposed approach.

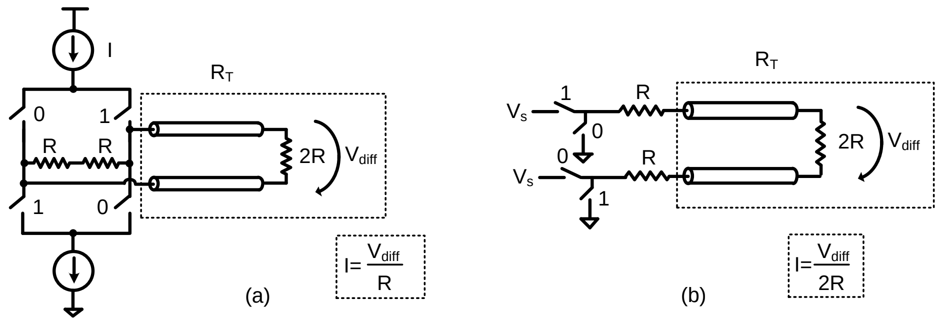


FIGURE 6.1: Comparison between current-mode (a) and voltage-mode (b) termination schemes. V_{diff} is the differential output amplitude and I is the bias current needed to achieve this amplitude.

6.2 Voltage Mode versus current mode drivers

In current mode LVDS drivers a large constant stabilized current of several mA is fed to the differential transmission line termination resistor [108, 116]. Switches are included to change the polarity of the termination resistor (see Fig. 6.1(a)) to transmit a '0' or '1', but the source current is always constant. Switching OFF this huge current during inter-event pauses and back ON quickly is not trivial, because achieving nano second delays together with stabilized behaviour requires special driver sophistication. On the other hand, voltage mode drivers switch a constant voltage through the differential transmission line termination resistor. For this, the driver uses power transistors that operate as switches (see Fig. 6.1(b)). The current path can therefore easily be shorted and driver static power consumption could be minimized by simply turning OFF these power switches. Besides the ease of ON/OFF switching, Fig. 6.1 shows how the termination scheme used in voltage mode drivers requires two times less current flowing through the driver for a desired differential output amplitude V_{diff}. On the other hand, lower supply voltages can be used for voltage mode drivers due to their low common mode and low swing requirements. However, voltage mode drivers do require a pre-driver circuit to properly drive the power switches, and real power consumption is also influenced by pre-driver power.

In current-mode approaches, using AC coupling and a matched channel, it is possible to transmit data over several meters of cable length. On the other hand, voltage-mode drivers are intended to be used in short-haul interconnect while dissipating lower power [124]. Current-mode drivers are also more robust against supply noise and ground bounce due to the use of matched current sources to generate the output current. In voltage-mode implementations, the noise coupled into supply voltage or ground directly impacts on the output because there is a direct connection between those nodes.

While impedance matching is easy in current-mode drivers, it is an important issue in voltage mode circuits. In these solutions, the switch resistance is used to match the channel characteristic impedance. As this parameter is severely affected by temperature and process variations, some kind of calibration is required to keep reflections and differential output amplitude within the target values. Some published solutions use an additional control loop to tune the driver characteristic [124]-[125]. However, the extra loop requires significant static power. Other methods digitally tune the output differential amplitude [120], allowing the information to be stored in digital registers, and thus reducing static power consumption.

In this Chapter we decided to develop a voltage mode solution due to the potential reduction in dynamic power consumption and the simple implementation of the ON/OFF switching circuitry. Moreover, the inter-chip communication links are short in the system level design presented in Chapter 3 for a network on board solution, and this questions the possible advantage of a current mode link in terms of speed and signal integrity. However, impedance matching and driver and pre-driver supply voltage generation must be carefully designed to exploit the theoretical advantages of the voltage mode implementation.

6.3 Driver circuit

Fig. 6.2 shows a top level description of the proposed switchable voltage mode driver. Signals *INPOS* and *INNEG* are provided by the serializer circuit and contain the event information that must be transmitted. These rail to rail digital signals are processed by a two stage pre-driver designed to drive the high capacitive load at the driver input, while presenting a low input capacitance. This block also performs the task of switching OFF the transmitter operation when there is no data to be transmitted. Signal *en* enables the pre-driver operation when *reqSER* or *ackSER* signals are at low level, indicating an event transmission.

The matching of the transmission line impedance at the transmitter side is essential to ensure good signal integrity. In voltage mode implementations, the output differential voltage is created by switching a constant voltage across the termination resistor. In this design, the switching transistors are also used to match the line impedance. After analyzing the circuit in Fig. 6.1-(b), it can be stated that the impedance matching condition is achieved when the serial resistance of the switches used to create the output swing is equal to the receiver termination resistor. If this condition is satisfied, the differential amplitude is $V_s/2$, where V_s is the driver supply voltage. A dedicated calibration circuit to compensate the process variations is included to tune the driver termination impedance by measuring the output differential amplitude. V_s is generated by an internally compensated regulator which sets a constant value for this voltage and allows a very sharp switching of the supply current without compromising the regulated voltage transient response.

Fig. 6.3 shows the schematics of a low swing voltage mode pre-driver/driver using only NMOS transistors M1-M4 for line termination purposes. The desired differential output amplitude V_{diff} is generated at R_T by setting $V_s = 2V_{diff}$. R_T in Fig. 6.3 accounts for the off-chip resistive components (transmission line and receiver impedance

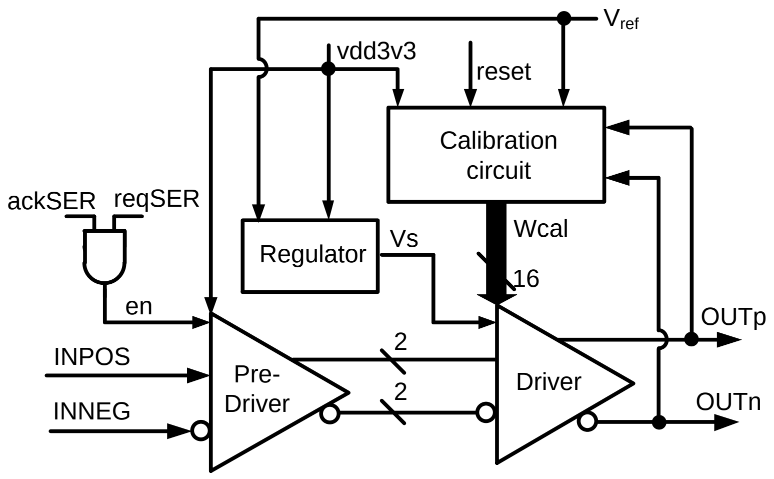


FIGURE 6.2: Top level description of proposed switchable voltage mode driver.

termination). The two R resistors in Fig. 6.1-(b) account for the on-chip termination resistance provided by transistors M_i. Impedance matching is digitally controlled by adjusting the width of M3-M4, using unitary transistors of (4.5/0.35) μm activated by (4.5/0.35) μm switches. Upper termination transistors M1-M2 are fixed-width transistors with (80/0.35) μm aspect ratio. This asymmetric control makes the calibration loop simple, but the output common mode is not tuned. This implementation uses a 16 bit thermometric control for the impedance. Post-layout Montecarlo simulations with mismatch and process variations show that this number of bits is sufficient to limit the output differential amplitude error to within 7% of 3-σ error. The same analysis revealed 3-σ error of 25% from the nominal value of 600mV for the output common mode.

A tri-state pre-driver is used to switch OFF the driver during pauses. During regular operation, pre-driver inverters are not at high impedance and driver transistor gates are forced to proper values. During pauses, these inverters are tri-stated, with M3-M4 open and M1-M2 closed. In this situation, no static current flows through the termination resistor as the output terminals are both tied to V_s. Hence, the overall driver consumption is reduced to the pA range, corresponding to auxiliary digital circuit leakage currents.

Fig. 6.4 shows a system level description of the calibration circuit used to achieve impedance matching. A comparator senses the output differential mode and compares it to an internal reference. The presence of R_T is required during calibration to take into account all the resistive components in the signal path that contribute to the final differential amplitude. A digital controller analyzes this information to act on a shift register which stores the impedance control thermometric code. This code activates a number of fingers in the digitally programmable transistors to control the output differential amplitude. The calibration algorithm starts by setting the calibration word

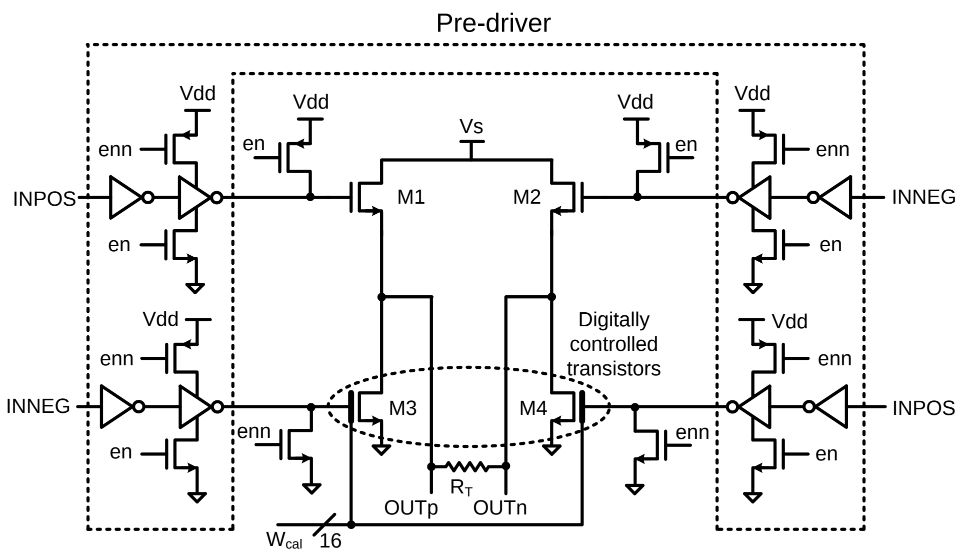


Figure 6.3: Proposed Voltage-Mode driver and Pre-Driver circuits.

to the minimum switch resistance situation (all the fingers disabled). At every calibration step, the differential amplitude is compared with the reference and the calibration word shifted if the amplitude is lower than the target. The process stops when the programmed differential amplitude is detected to be higher than the target and this is considered as the optimum calibration word.

Fig. 6.5 shows the schematics of the proposed differential mode comparator. Two source degenerated differential pairs are used to compare differential amplitudes which may be in the order of 500mV. Amplitudes in this range need to be processed by a highly linear differential pair to generate currents proportional to the target differential amplitude without saturation. The currents generated by the input stage are combined through current mirrors in order to produce an output current proportional to the difference between a reference differential amplitude and the programmed amplitude. A resistor divider is used to generate precise internal references. All static power consumption is eliminated when the driver is not being calibrated using an enable switch.

6.4 Power management for switching drivers

The driver requires the output current to switch from the mA range during regular transmission to the nA range during pauses. Furthermore, instantaneous current pulses supplied by V_s must be very sharp for a high speed switching operation. Typical voltage regulators using a low frequency dominant pole for loop stability are not able to react

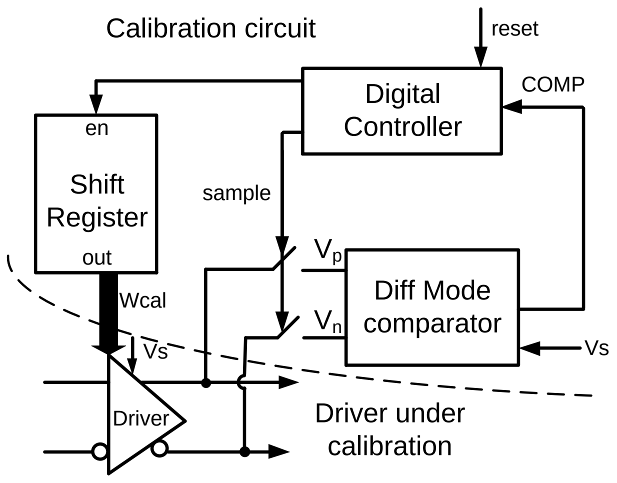


FIGURE 6.4: Calibration circuit system level description.

at the speed demanded by the required instantaneous output current. A large external capacitor is used to compensate the loop and provide the needed charge packets during transients. Even if the load is not demanding current, the regulator power transistor can draw several mA due to the loop dynamics.

Internally compensated regulators do not employ this large output capacitor. The dominant pole is located in an internal node and the relatively small output capacitor cannot be efficiently used to provide instantaneous output current pulses. These kinds of regulators require internal transient response compensation in order to achieve similar specifications as externally compensated regulators. However, they allow the pass transistor current to change at the same speed as the output current. Fig. 6.6(a) shows the regulator schematics used in this work to generate the driver voltage supply V_s.

Several design techniques have been used to combine internal compensation with good transient response. Stability is achieved using a two stage amplifier with a Miller compensation capacitor C_c. This capacitor splits the dominant amplifier internal pole and the output pole, improving the system phase margin. The pass element is built with a source follower because no low drop-out is required as $V_s < 1V$ is well below $V_{dd} = 3.3V$. If output voltage increases, NMOS overdrive is reduced and output current is lowered. Otherwise, the loop acts in the opposite way, increasing the output current. Using an NMOS transistor as pass element helps stability because V_s is a low impedance node and the output pole is pushed to high frequencies.

A fast feedback path between the output node and the pass transistor gate is implemented to reinforce the transient compensation and improve the regulator slew rate. The compensation circuit is based on differentiating the output voltage V_s and generating a current that is injected or subtracted from the pass transistor gate node V_{gate}. If a positive voltage variation at V_s is detected, then the current drawn by the

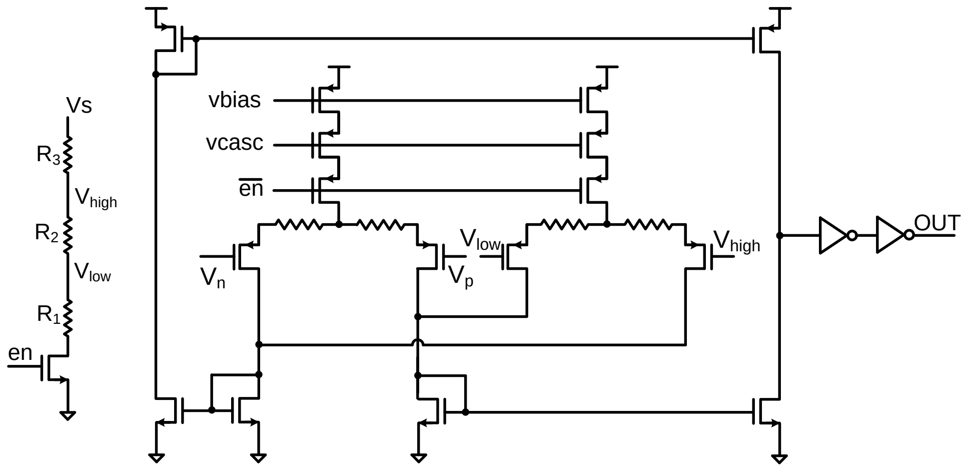


Figure 6.5: Differential mode comparator circuit.

pass transistor is excessive for the current load conditions and its gate voltage must be decreased by subtracting current from V_{gate}; that current is instantaneously provided by the N-type transistor of the fast feedback path. A negative slope in V_s means that the load is demanding current that the regulator cannot provide. Thus, the fast feedback path injects current in V_{gate} to increase the pass transistor driving capabilities and improve the response time to the change in the load conditions.

This extra loop may impact stability by pushing the output pole to lower frequencies, degrading the phase margin. The transient compensation has therefore been carefully designed, since there is a trade-off between stability and transient response to variations in the load conditions. Montecarlo simulations with process and mismatch variations for the worst case scenario for stability (no load) reveal a 62^o mean phase margin with 1.2^o of 1-σ variation, demonstrating the robustness of the employed design procedure.

6.5 Receiver circuit

The common mode generated by the calibration circuit is not well controlled as only one branch is implemented as a digitally controllable transistor. Moreover, the reference voltage V_s is around 1V, leading to a low output common mode for a 3.3V voltage supply. The receiver front-end must therefore be implemented with PMOS transistors and input common mode range must be optimized. This must be combined with techniques for quickly turning the bias currents ON/OFF.

The receiver circuit is shown in Fig. 6.7. It uses the same two-stage architecture than the receiver proposed for the current mode solution described in Chapter 5. The pre-amplifier has been optimized for processing a very low common mode. DC voltages at the PMOS differential pair output are set low enough to enlarge input common mode range. This allows to amplify the low common mode signal generated by the transmitter

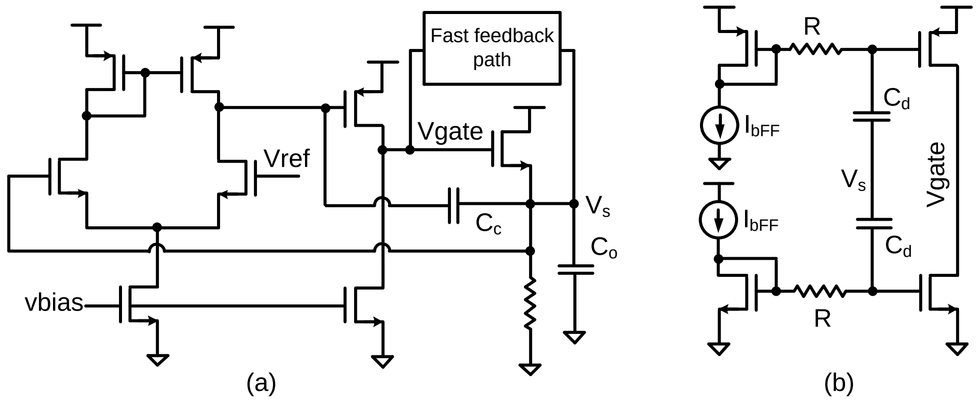


FIGURE 6.6: (a) Regulator circuit with internal compensation and NMOS pass transistor (b) Fast feedback path used for transient response compensation.

and adapting the common mode to the next stage. A rail-to-rail version of the serial signal is obtained using a continuous time comparator, whose design constraints are alleviated due to the first amplification and adaptation stage.

Receiver power consumption is mainly due to preamplifier and comparator bias currents. However, these two circuits are not needed during pauses if the comparator output is forced to have a constant value (zero in this design). Their bias currents can therefore be disabled during pauses. Signal *ackSER* is used to switch the receiver ON/OFF, as it has been done in the current mode version. The switches marked with dashed lines in Fig. 6.7 are used to turn OFF the bias currents in the OFF state.

6.6 Experimental results

A test prototype for the switchable I/O circuits was fabricated in 3.3V 0.35μm CMOS. All the components described in this Chapter were integrated as custom-made pads with ESD protections, along with ground and supply voltage decoupling capacitors and a circuit providing the analog biases. The voltage mode driver needed a 530x490μm^2 area, while the receiver took 270x341μm^2. Fig. 6.8 shows a microphotograph of the fabricated chip, the main parts of which have been highlighted. The serial AER Manchester-encoding SerDes circuit [123] handles the parallel AER flow and creates the proper signals for the driver/receiver pair. The transmission channel consists of a pair of PCB traces forming a 100Ω differential microstrip line 5cm long. The test set-up is exactly the same used for the current mode prototypes as both chips were designed to be assembled in the same PCB.

Fig. 6.9 shows how the AER protocol signals flow through the voltage mode serial link for a 500Mbps bit rate. The delay between *reqIN* and *ackIN* at the input parallel AER interface is 15ns and the serialization process duration is 68ns. The serial

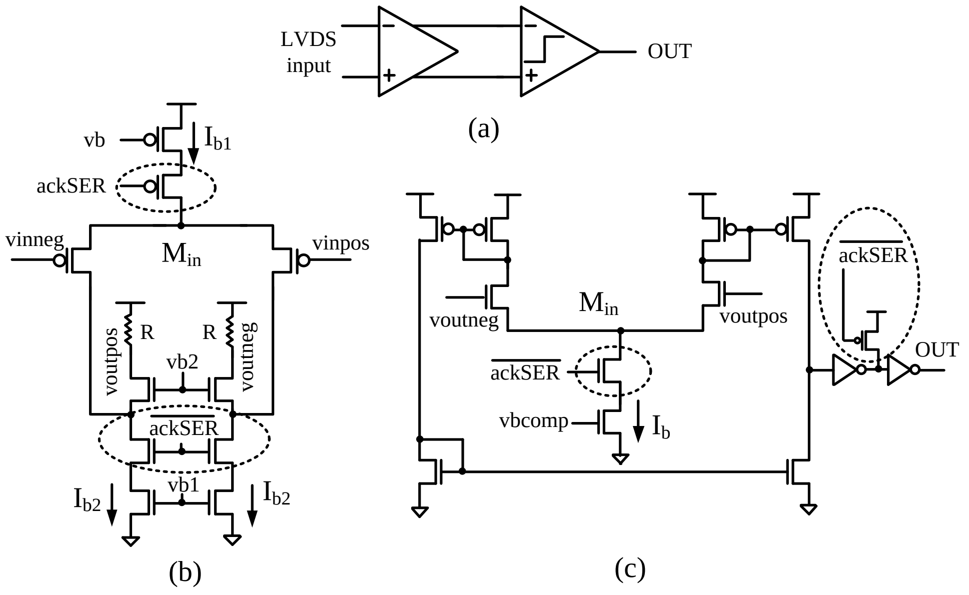


Figure 6.7: (a) Receiver architecture (b) Preamplifier (c) Continuous time comparator.

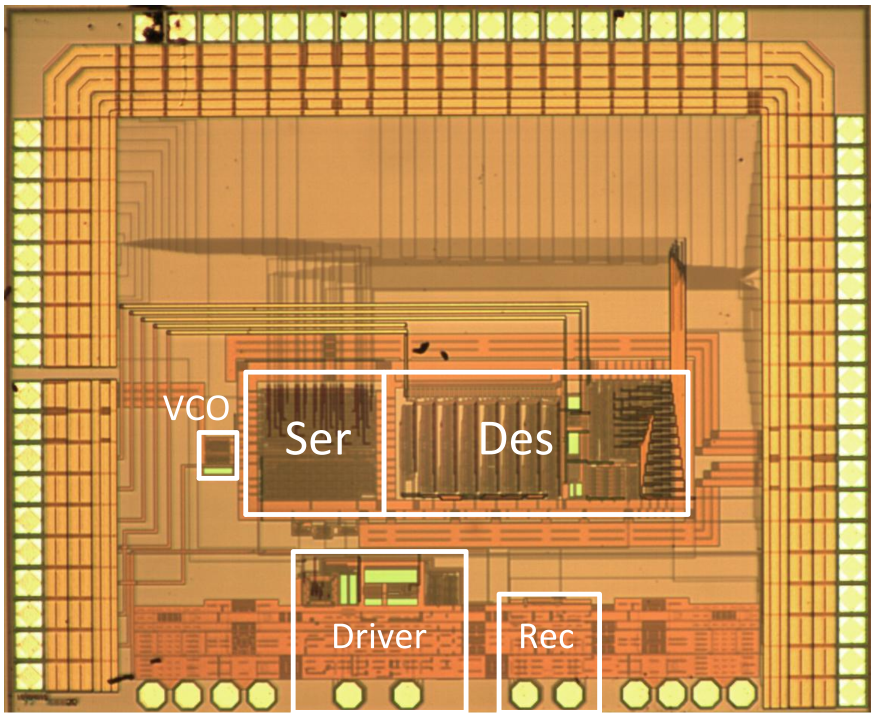


Figure 6.8: Microphotograph of the test chip.

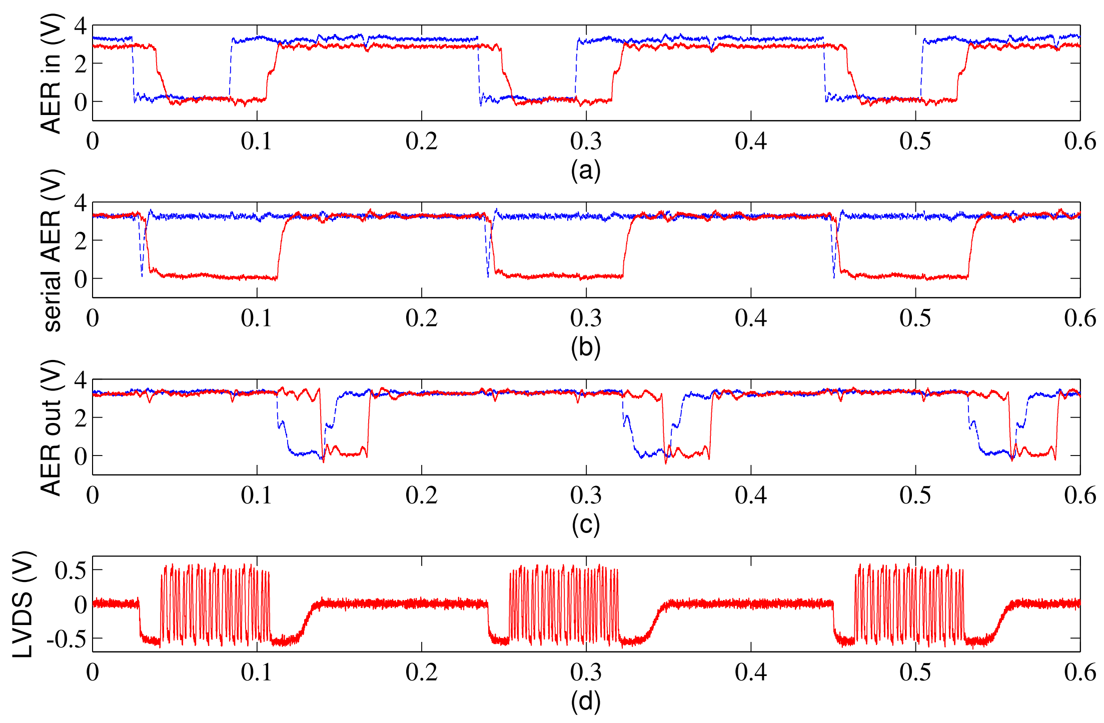


Figure 6.9: AER protocol management at (a) *reqIN* (dotted line) and *ackIN* (continuous line) signals (b) *reqSER* (dotted line) and *ackSER* (continuous line) signals (c) *reqOUT* (dotted line) and *ackOUT* (continuous line) signals (d) differential mode of the LVDS signal. Time scale is μs

handshaking protocol introduces a delay of 4.6ns between the *reqSER* and *ackSER* and the acknowledge stays activated for 81ns. The output handshaking protocol with the USB-AER test boards introduces a delay of 23ns. For a 500Mbps bit rate, an input to output request latency of 89ns is measured, leading to a maximum event rate of 11.2Mevent/s.

Fig. 6.10 shows the differential mode of the high speed bit stream at the receiver input. Waveforms were acquired with the Agilent DSO81304B Infinium oscilloscope with 5GHz bandwidth probes. The bit rate is set to 500Mbps and a 5Mevent/s 32 bit event rate stream is generated with the USB-AER boards. The differential amplitude measured after calibration is 520mV, which represents a 4% error with respect to the target amplitude of 500mV ($V_s = 1V$). Switching ON time is 1.5ns, while the switching OFF time is 7ns. The latter was made longer to minimize the voltage drop observed in the regulator when the output current is switched from maximum to minimum values. Fig. 6.11 shows the eye diagram measured at the receiver input. A 40ps of rms jitter was obtained for this implementation, very similar to the value reported previously [123] with the conventional non-switchable LVDS driver [108].

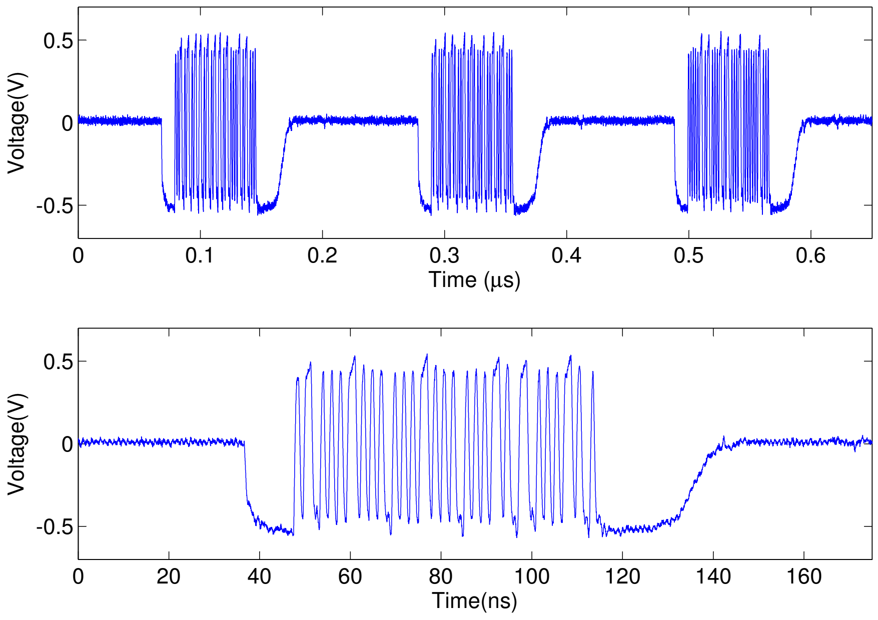


Figure 6.10: Measured high speed serial signal at receiver input for the voltage mode driver.

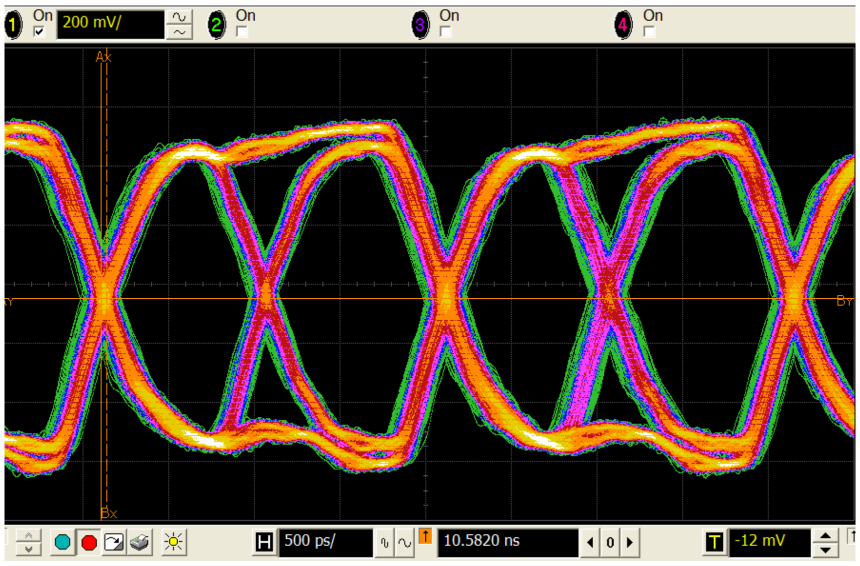


Figure 6.11: Eye diagram for the voltage mode driver at receiver input.

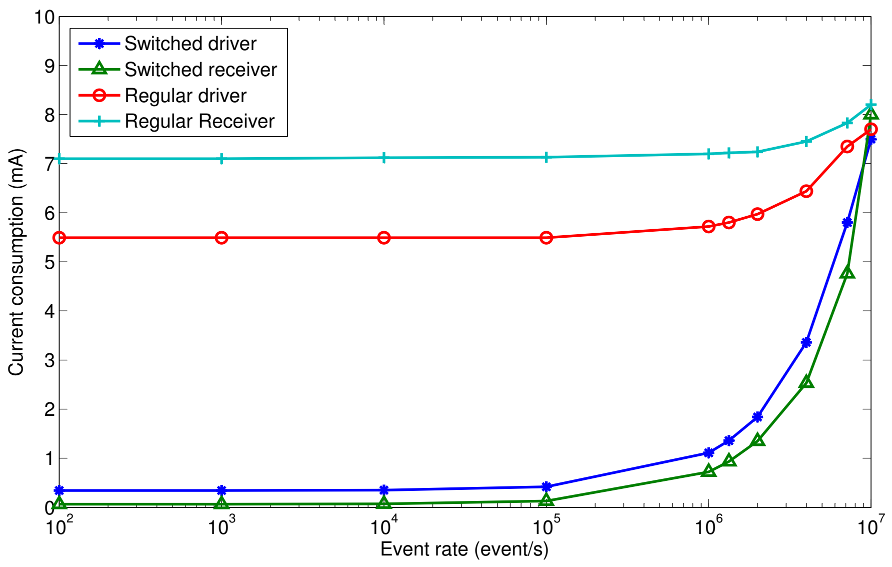


Figure 6.12: Voltage-mode switchable driver and receiver current consumption versus event rate.

Transmission frequency was varied by tuning the on-chip high frequency VCO (Voltage Controlled Oscillator) to achieve the maximum event rate. In this case, the link worked correctly at a maximum bit rate of 638Mbps, with an input to output request latency of 73ns. This reveals that the maximum achievable event rate combining the burst mode SerDes architecture proposed previously [123] with the switchable I/O circuitry presented in this Chapter is 13.7Mevent/s.

Fig. 6.12 shows the current consumption dependence with the event rate for the case $V_s = 1V$. Current consumption scales down with the event rate, reaching a minimum of $343\mu A$ for the driver and $62.5\mu A$ for the receiver when the link operates below 10Kevent/s. The test chip has a control bit which disables the switching mechanism resulting in a conventional voltage mode link. If the circuits are configured to not switch OFF during pauses, current consumption for the minimum event rate is 5.5mA and 7.1mA for the driver and receiver, respectively. In the maximum event rate situation, driver and receiver stay turned ON most of the time and current consumption is the same for the switching and non-switching implementations. This maximum current consumption is 7.7mA for the driver and 8.2mA for the receiver.

The lower current consumption is due to the regulator quiescent current. Decreasing this current leads to a larger gap in the pass transistor gate voltage for ON and OFF situations. This trades off with the regulator transient response because compensation circuits have to be faster in charging/discharging the pass transistor gate node. However, this problem can be alleviated in more advanced technologies which enable a faster regulator step response without compromising stability.

Differential amplitude is controllable through an off-chip analog bias voltage. Power

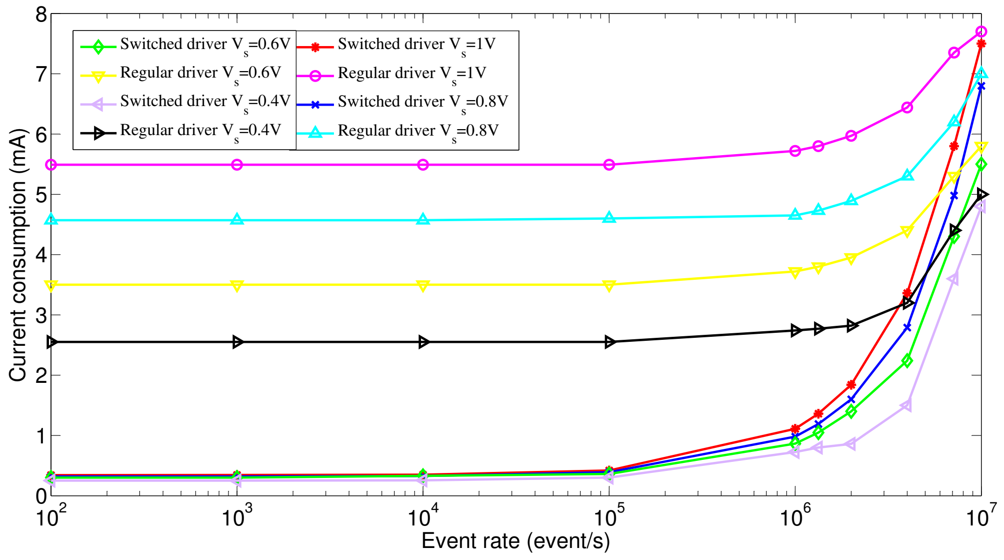


Figure 6.13: Voltage-mode switchable driver and receiver current consumption versus event rate for different V_s voltages.

consumption while transmitting events is proportional to the differential amplitude defined by V_s. The amplitude control calibration loop selects the optimum calibration word to achieve the required driver switches resistance and have a $V_s/2$ amplitude. Fig. 6.13 shows how the selected differential amplitude affects driver current consumption. Current consumption for higher event rates can be reduced by setting a lower amplitude, but it remains roughly constant for very low event rates. In this situation, current consumption is not given by the current delivered to the load, but by the regulator quiescent current.

Table 6.1 compares between voltage mode implementations of high speed drivers reported in literature. Two figures of merit are provided in an attempt to perform a fair design-based comparison. The first is $FoM1 = (V_{dd}/P_{max}) \times V_{AMP}$, which is proportional to the differential voltage amplitude on the transmission line over total driver current consumption. The second multiplies the first by factor BR/f_t (BR = bit rate) to include the speed. The higher each figure of merit is, the more efficient one can consider the design. This work is the only solution that scales down link power consumption with event rate. The driver design is comparable to other state of the art solutions in terms of area, supply voltage and differential amplitude. Bit rate and area data have been normalized by the corresponding technology transition frequency[4] f_t

[1]w/o internal regulator and single ended output

[2]w/o internal regulator

[3]power consumption of one whole TX chip

[4]References in Table 6.1 do not provide their technology f_t, but this number has been estimated from other similar technologies.

TABLE 6.1: Performance comparison with other voltage mode drivers

	[124]	[125][1]	[120][2]		[126][3]	[127][3]	This work
Technology	90 nm	0.18 μm	0.18μm		65 nm	90 nm	0.35 μm
f_t(GHz)	180	120	120		250	180	15
Power Supply (V)	1	1.8	1.8		1 (pre-driver) 1.5 (driver)	1.2	3.3
V_{amp}: Diff Amp(mV)	150	125	450	88	500	600	200-500
BR: Bit Rate(Gbps)	6.25	3.6	3.5	3	8.5	8	1.28
Bit Rate / f_t	0.035	0.03	0.029	0.025	0.034	0.044	0.085
Area (mm^2)	0.15	0.198	0.044	0.024	0.065	0.138	0.26
Area/λ^2	74	24	5	3	61	68	8.5
$\bar{P}_{max}$ (mW)	2.26	7.86	22.1	9.0	96	101	15.8-25.4
$\bar{P}_{min}$ (mW)	2.26	7.86	22.1	9.0	96	101	1
Impedance Matching	Pre-Driver Supply Voltage	Pre-Driver Supply Voltage	Number of fingers		Source series termination	Pass Transistor	Number of fingers
FoM1=$V_{dd}V_{amp}/P_{max}$	66.4	28.6	36.7	57.6	6.5	7.13	42-65
FoM2=FoM1$\times$BR/f_t	2.32	0.86	1.06	1.44	0.22	0.31	3.57-5.52

and λ to allow a fair comparison among designs. The physical bit rate for this work is 1.28Gbps which corresponds to twice the Manchester encoded transmitted data bit rate of 0.64Gbps.

6.7 Conclusion

In this Chapter, we have described the design and test of a voltage mode high speed I/O interface optimized for transmitting burst mode data flows. The proposed circuits exploit the asynchronous nature of AER traffic to disable the static driver and receiver currents in pauses, as it was done with the current mode implementation described in Chapter 5. However, the voltage mode technique imposes new design constraints which have been addressed throughout this Chapter. Besides the design of an efficient switching mechanism to turn off the high speed circuitry, design techniques regarding the impedance matching mechanism, the power management of the high speed driver and the low common mode of the received signal have been discussed. Experimental results are provided for a 3.3V 0.35μm CMOS prototype which achieves a current consumption reduction factor between the maximum and minimum event rate situations of 22 for the driver and 130 for the receiver at 500Mbps.

This Chapter finishes the description of event-driven communication circuits for a power efficient and high performance VLSI implementation of AER chips. Our ultimate research goal is to integrate hundreds of AER units in a PCB to carry out sophisticated spike-driven vision processing tasks. However, there are many open research issues which must be addressed before designing a fully integrated solution for the network units.

An optimized hardware platform for prototyping more advanced designs would be desirable to explore the challenges arisen from an AER multi-chip assemble. A dedicated board with AER friendly connectivity resources will allow the neuromorphic engineers to test large scale systems reducing the engineering time. This board should include high speed serial links resources for board to board connectivity and conventional parallel AER connectors to interface with the existing AER chips. A versatile configuration infrastructure is also mandatory because AER modules use to have a high degree of programmability. Based on these ideas, the next Chapter will describe the design of a general purpose AER prototyping board intended to be used in large scale AER-based neural systems.

7 Future Outlook: The Node Board

7.1 Introduction

Chapters 4, 5 and 6 describe how to implement the communication layer between neighbor chips in a Network on Board system where hundred of AER modules operate to implement a user-defined spiking neural network architecture. However, the definition of spike-based systems is still an open issue which is currently under research, specially in the area of ConvNets [128, 129]. For example, for this kind of neural networks there is still much work to do on the implementation of new features in the convolution modules, improving their resolution, the NoC integration of many convolution units or the development of spike-based training algorithms. Although VLSI solutions offer the best performance in terms of power consumption, processing latency and integration density, an FPGA-based environment allows the designer to define the architecture of every network node which can implement custom-designed event processing units. Moreover, it is a flexible platform to test several routing schemes and network communication layers for different applications and choose the most suitable one for large scale event-driven neural networks.

Due to these considerations, having a hardware level programmable solution for prototyping real AER-based networks before going to the VLSI solution is very convenient. The infrastructure should be fully programmable at all levels. The lower level of programmability refers to the definition of every network block functionality. Once the hardware architecture is defined, there must be the possibility of changing the computation parameters on the fly. At the same time, the system also has to interact with the existing neuromorphic hardware which uses generic AER parallel ports. This way, the AER network could map mixed mode systems where, for example, convolution units could operate together with other pieces of neuromorphic hardware.

This Chapter describes the design of a custom board which fulfills the requirements described before. The Node Board (NB) uses a XC6SLX150T Spartan-6 Xilinx FPGA as a core element to implement the communication infrastructure described in Chapter 3 and to interface with other AER chips through parallel ports. A high end FPGA is required to provide enough slice and memory resources to implement the network computation intelligence and high speed serial channels to support the inter-board communication interface. The FPGA-based full-duplex high speed serial link described in Chapter 3 is intended to communicate nodes with each other. The board has also been designed to be assembled with neighbor boards in a scalable and expandable way using the ideas presented in Chapter 3.

7.2 Board architecture

Fig. 7.1 shows a block diagram of the NB where the main parts have been highlighted. A XC6SLX150T Spartan-6 Xilinx FPGA is the core element which provides the logic, memory and connectivity resources required to implement the network units. Two parallel AER connectors can link the FPGA device with AER modules. Extra pins in the conventional 40-IDC (Insulated Displacement Connector) AER connector not used for the event handshaking can implement a configuration interface with the event-based processing chip. The inter-board communication interface is implemented using the FPGA Rocket I/O GTP transceivers through east, west, south and north full-duplex links. SATA connectors host the communication interfaces as they offer a low cost and high bandwidth solution. A pin header connector provides a low speed interface between boards to implement a daisy chain configuration network, which distributes the configuration information throughout the NB array and runs in parallel with the event-based network.

Besides the connectivity resources, the NB has internal blocks which are necessary for a proper operation of the FPGA features. High-efficiency switching power supplies are used to bias the FPGA core and I/O interfaces by generating 1.2V, 2.5V and 3.3V from a master voltage of 5V. Linear regulators are used to provide clean supply voltages to the GTP transceiver circuitry, the reference clock generator and the configuration memories. A differential 100/125MHz low jitter clock generator feeds the reference clock to the GTP transceivers and a low cost 100MHz single-ended clock provides an alternative clock for user-defined applications. Two platform FLASH memories, XCF08PFSG48C and XCF32PFSG48C, have been included to store the FPGA programming file in a non-volatile way.

7.2.1 Parallel connectors

The NB has four parallel connectors where two AER ports and two configuration interfaces can be connected. An SMD (Surface Mount Device) male 40 pins IDC connector is used for the AER connectivity. The connector has dedicated ground contacts and 5V and 3.3V regulated DC voltages which can be supply voltages for

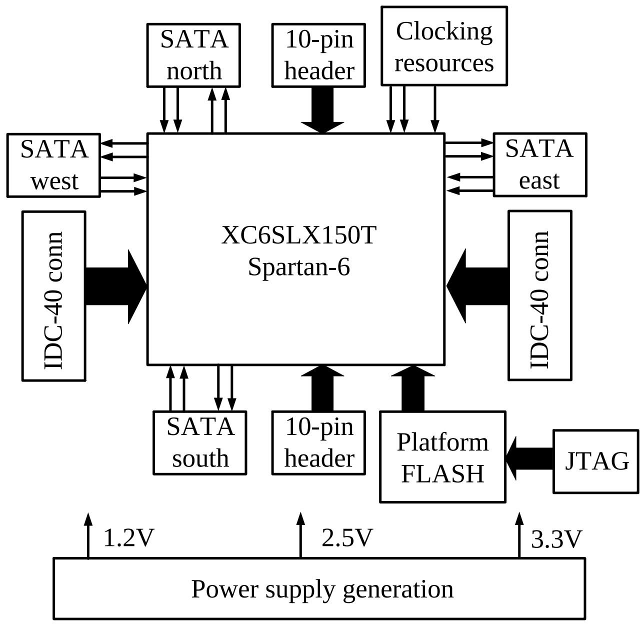


Figure 7.1: Block diagram of the node board.

Table 7.1: AER connectors pin-out

Name	IDC-40 pin	IN	OUT	Name	IDC-40 pin	IN	OUT
GND	40	–	–	5V	39	–	–
3.3V	38	–	–	AER27	37	Y22	AA1
AER26	36	V21	Y1	AER25	35	W22	W1
AER24	34	U22	V2	AER23	33	V22	V1
AER22	32	T22	U1	AER21	31	T21	T2
GND	30	–	–	ACK	29	Y21	U3
AER20	28	P21	T1	AER19	27	R22	R1
GND	26	–	–	AER18	25	P22	P2
GND	24	–	–	AER17	23	N22	P1
GND	22	–	–	REQ	21	N20	N3
AER16	20	M22	N1	GND	19	–	–
AER15	18	L22	M1	AER14	17	M21	M2
AER13	16	K21	L1	AER12	15	K22	K1
AER11	14	H22	K2	AER10	13	J22	J1
AER9	12	G22	H1	AER8	11	H21	H2
AER7	10	F21	G1	AER6	9	F22	F1
AER5	8	D22	F2	AER4	7	E22	E1
AER3	6	C22	D1	AER2	5	D21	D2
AER1	4	B21	C1	AER0	3	B22	B1
GND	2	–	–	3.3V	1	–	–

TABLE 7.2: Configuration connectors pin-out

Name	Pin Header1	FPGA	Name	Pin Header2	FPGA
GND	1	–	GND	1	–
3.3V	2	–	3.3V	2	–
GPO_0	3	R19	GPO_0	3	G16
GPO_1	4	R20	GPO_1	4	F17
GPO_2	5	T19	GPO_2	5	D18
GPO_3	6	T20	GPO_3	6	C19
GPO_4	7	U19	GPO_4	7	C18
GPO_5	8	U20	GPO_5	8	B20
GPO_6	9	V19	GPO_6	9	B18
GPO_7	10	W20	GPO_7	10	A18

the AER chip connected on top of the NB. The rest of the pins are routed to general purpose FPGA pads to be used as user-defined signals. Besides the AER addresses information, the connector can also transfer configuration parameters to the hosted AER chip. Table 7.1 shows the pin-out and FPGA connectivity for the IN and OUT AER connectors.

Apart from the two IDC-40 pins connectors, there are other two 10 pin headers. These connectors can propagate configuration information between NBs through an SPI or JTAG chain, depending on the configuration set-up. These two pin headers have ground and 3.3V supply voltage connections and all pins are connected to general-purpose FPGA pads. Table 7.2 shows the connections between the FPGA and pin headers.

7.2.2 High speed serial transmission

The NB has connection resources to implement four bidirectional high speed serial AER links to transfer events between adjacent neighbors. The physical layer is provided by the Xilinx Rocket I/O technology. Fig. 7.2 shows the block diagram of a Spartan-6 GTP tile which includes two full-duplex high speed serial channels and a shared PLL block to generate a master frequency for transmitters and receivers. Each tile is composed by two GTP transmitter/receiver pairs which can be programmed to implement any standard (PCIe, SATA, XAUI,. . .) or user-defined protocol. The built-in dedicated hardware includes some useful blocks which are going to be exploited in the design:

- **8b/10b encoders/decoders:** this block automatically encodes the user data flow into a DC-free code which is suitable to be transmitted through an AC-coupled channel. Thus, the number of transmitted ones and zeros are kept the same by adding an extra overhead of 2 bits every 8 bits of data.
- **Transmitter emphasis/Receiver equalization:** both mechanisms are programmable and allow the user to compensate channel losses and dispersion. The differential amplitude is also configured by the user, providing a versatile physical layer which can adapt to a great variety of channels.

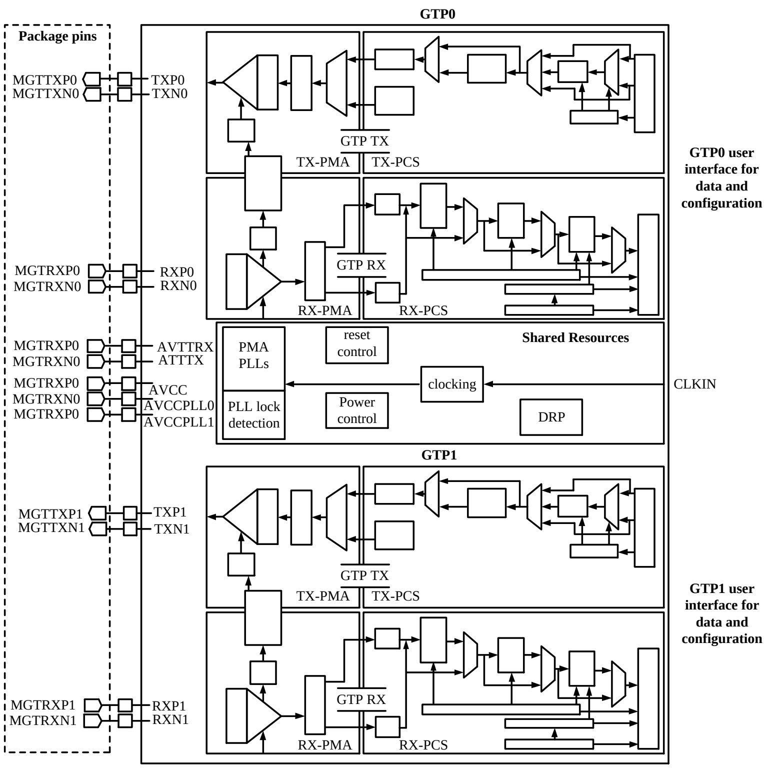


Figure 7.2: Spartan 6 FPGA GTP tile configuration including two full-duplex high speed serial links and the shared clocking resources.

- **Comma detection and alignment:** transmitter sends data packets grouped in 8, 10, 16, 20, 32 or 40 bits, depending on the encoding scheme and the word length selected by the user. At start-up, the receiver has to detect special packets with a dedicated alignment word to guess where the data flow starts and properly deserializer data. The user only has to make sure that the alignment character is sent when the link is powered up.
- **Clock correction:** transmitter and receiver have buffers and FIFOs to compensate clock disparities between their internal clock domains. These differences are even higher between different boards. For this reason, a clock correction block is implemented in a receiver elastic buffer to compensate these disparities. Clock correction actively prevents the receiver elastic buffer from getting too full or too empty by deleting or replicating special idle characters in the data stream.

Fig. 7.3 shows a schematic diagram of the Rocket I/O auxiliary components included in the NB. The master clock for the serialization and clock recovery are generated by a low jitter PLL which is shared between two transceivers. An external LVDS low jitter reference clock is multiplied by a user-programmable integer number to get the desired data rate. This reference clock can be internally propagated to several tiles through a low jitter clock propagation network. We have used this feature to include a single reference clock on the board for the two tiles needed. Jitter performance of

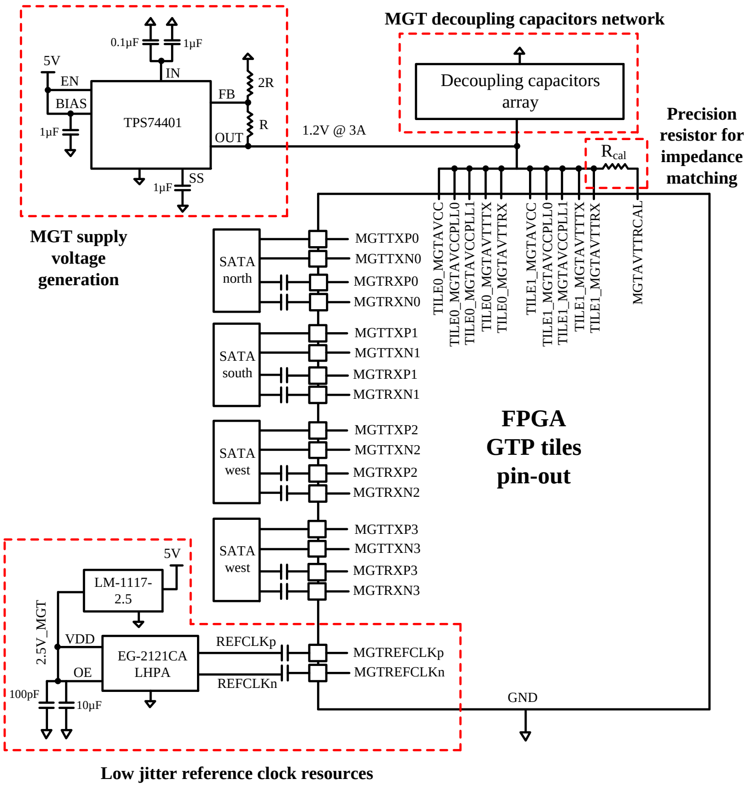


FIGURE 7.3: Board design to host the Rocket I/O transceivers.

the reference clock directly impacts on the high speed signal quality generated by the transceiver. Hence, a crystal-based solution, like the EG-2121CA LHPA part, has been selected for this design to achieve an appropriate jitter performance. Moreover, a clean supply voltage is also important to achieve a low noise clock generation. An LM-1117 linear regulator provides a 2.5V supply voltage with a proper noise rejection to bias the clock generator.

Termination impedance is also a major issue to ensure a good signal integrity. As all termination resistors are integrated in the FPGA and the receiver has to adapt itself to different characteristic impedance transmission lines, the termination resistor value is configurable through an external precision resistor, as it is shown in Fig. 7.3. All tiles within the same bank share the same configuration word. The calibration procedure is performed once during the FPGA configuration, not affecting the link operation nor the power consumption afterwards.

The Rocket I/O circuitry in the Spartan-6 FPGA requires a single analog power supply at a nominal voltage level of 1.2V. Noise on the GTP analog power supply can cause degradation in the performance of the GTP transceiver. The most likely form of degradation is an increase in jitter at the output of the GTP transmitter and reduced jitter tolerance in the receiver. Typical noise sources are the own regulator intrinsic noise, coupling with other circuits or the power distribution network. All these sources must be taken into account and minimized for a proper design.

In our GTP sub-block design shown in Fig. 7.3, a dedicated TI TPS74401 linear regulator provides the GTP supply voltage. The regulator is placed as close as possible to the FPGA to reduce the coupling between the GTP supply voltage plane and the other elements. This also reduces the resistance and inductance of the supply voltage path, minimizing the noise caused by the load dynamics. A linear regulator is chosen over a switching one to minimize the supply voltage noise and reducing the power supply generation area in the PCB. This is done at the cost of increasing the overall power consumption due to the lower efficiency of linear regulators over their switching counterparts. Decoupling capacitors are also placed close to the GTPs supply voltage pins. Xilinx recommendations have been followed in order to provide a proper noise filtering at high frequencies combining $0.22\mu F$ and $4.7\mu F$ capacitors.

An AC-coupling termination scheme has been adopted for the reference clock and all receiver inputs, as can be seen in Fig. 7.3. In this technique, DC blocking capacitors are placed in series with both signals of the differential pair, in addition to the standard resistive load termination. Using AC-coupling, transmitter and receiver ends have decoupled common modes. This feature guards against differences in ground potential between different PCBs which can affect the link operation. As a large number of boards want to be assembled, this coupling technique is mandatory to keep a low error rate in the communication. The only drawback is that AC-coupling needs DC-balanced data and a 8b/10b encoding technique is necessary.

Routing high speed signals from the FPGA ball-grid to SATA connectors needs controlled impedance transmission lines to achieve good signal integrity. Fig. 7.4 shows the 10 layer stack-up used for the NB, highlighting the stripline structure which implements the high speed signal transmission lines. Two ground planes isolate the high speed differential signals layer from the rest, reducing the impact of interferences in the transmission and providing electromagnetic fields shielding. As it is illustrated in Fig. 7.4, differential coupled lines with W=2mm and s=0.12mm have been designed to have 100Ω of differential characteristic impedance. Lines width and separation have been obtained using the technology parameters of dielectric and metal layers provided by the PCB vendor and these approximate equations:

$$\begin{cases} Z_o = \frac{60}{\sqrt{\epsilon_r}} ln\left(\frac{1.9(h+t)}{0.8W+t}\right) \\ \\ Z_{diff} = 2Z_o\left[1 - 0.347e^{-2.9\frac{s}{h}}\right] \end{cases} \tag{7.1}$$

7.2.3 Configuration resources

An FPGA device offers the flexibility needed for the current application because the user can implement dedicated event-based processors or communications/configuration gateways with AER-based VLSI chips inside the network nodes. Two FLASH memories, XCF08PFSG48C and XCF32PFSG48C, are included as a non-volatile storage platform for the user designs. A JTAG chain connects the host PC running Xilinx

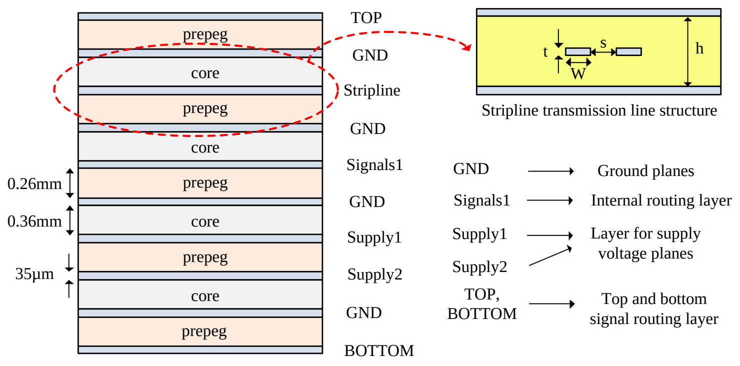

prepeg
core
TOP
GND
Stripline
GND
Signals1
GND
Supply1
Supply2
GND
BOTTOM
0.26mm
0.36mm
35µm
t
s
W
h
Stripline transmission line structure
GND → Ground planes
Signals1 → Internal routing layer
Supply1 → Layer for supply voltage planes
Supply2
TOP, BOTTOM → Top and bottom signal routing layer

FIGURE 7.4: Layer stack-up for the node board.

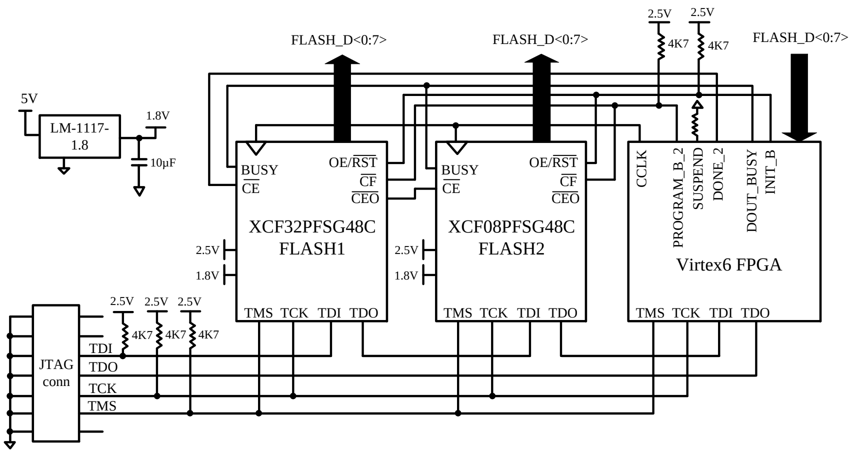


FIGURE 7.5: Platform Flash memories and JTAG configuration chain connection.

iMPACT software, the memories and the FPGA itself for configuration purposes.

Fig. 7.5 shows the JTAG chain connection diagram. The FPGA can read its configuration information from the on-board platform Xilinx FLASH memories or receiving configuration data from a host PC through the JTAG interface by bypassing the memories. In this last option, memories do not take part in the configuration process and everything is handled by the PC. This is very useful in debugging modes where the user wants to change the FPGA configuration to correct errors. If the configuration file is stable after the debugging process, it is interesting to store it on-board. In this situation, the FPGA is configured in a master mode to read the programming file from the platform FLASH. A low complexity LM1117-1.8V linear regulator generates a 1.8V supply voltage that the FLASH memories need to operate.

In this last situation, the configuration is controlled by the FPGA which generates a master clock, called *CCLK* in Fig.7.5. The process starts when, after power-up, FLASH memories send a reset to the FPGA through the *FPGA_PROG* pin. From this point on, *FLASH1* delivers an 8-bit configuration word every *CCLK* clock cycle. The FPGA reads this information until the FLASH memory internal counter saturates. While *FLASH1* is enabled, signal $\overline{CEO}$ makes *FLASH2* outputs stay in a high impedance state. This signal toggles when the *FLASH1* internal counter saturates and *FLASH1* gets disabled. At the same time, *FLASH2* is enabled and configures the FPGA. When the FPGA detects the end of a configuration process, signal *FPGA_DONE* is activated and memories are disabled. Any error occurred during configuration is indicated by *XFLASH_INIT#* and the process is aborted by reseting the platform FLASH memories.

7.2.4 Power supply design

In previous Sections, the power supply design for the GTP transceivers and their clocking resources were described. Due to the low noise requirements, linear regulators have been chosen sacrificing power efficiency. However, the digital core does not have these stringent constraints on noise and a switching regulator can be afforded for the Spartan-6 core and I/O banks supply voltages. Unlike the linear regulator, the switching regulator does not depend on the voltage drop between the regulator input and output voltages to provide regulation. Therefore, the switching regulator can supply large amounts of current to the load while maintaining high power efficiency. It is not uncommon for a switching regulator to maintain efficiencies of 95% or greater.

The disadvantages of the switching regulator are complexity of the circuit and noise generated by the regulator switching function. Switching regulator circuits are usually more complex than linear regulator circuits. This shortcoming in switching regulators has been addressed by several switching regulator component vendors. Normally, a switching power supply regulation circuit requires a switching transistor element, an inductor, and a capacitor. Depending on the necessary efficiency and load requirements, a switching regulator circuit might require off-chip switching transistors and inductors. Besides the component count, these switching regulators require very careful placement and routing on the printed circuit board to be effective. Texas Instruments (TI) offers very compact modules which include the switching regulator IC and the discrete components required to obtain efficiencies up to 95% and 6A output currents.

Fig. 7.6 shows a block diagram for the power supply generation used in the NB. Three switching regulators (*pth05050wad* and *pth04000wah* TI modules) are used to generate 1.2V for the FPGA core voltage supply and 2.5V-3.3V for the general purpose I/O banks. All blocks take the global 5V input DC voltage and generate an output which is controllable through a 0.05W 1% precision resistor connected between $V_{o,adjust}$ and *GND* pins. The voltage provided by the switching regulator is filtered using two large capacitors and a ferrite bead. This ferrite introduces a resistive component in the filter and limits the filter quality factor at high frequencies improving the regulator transient response.

The switching regulators have an Auto-Track function to power up the whole system in sequence. Auto-Track works by forcing the module output voltage to follow a voltage presented at the *Track* control pin on a volt-by-volt basis. When the output voltage

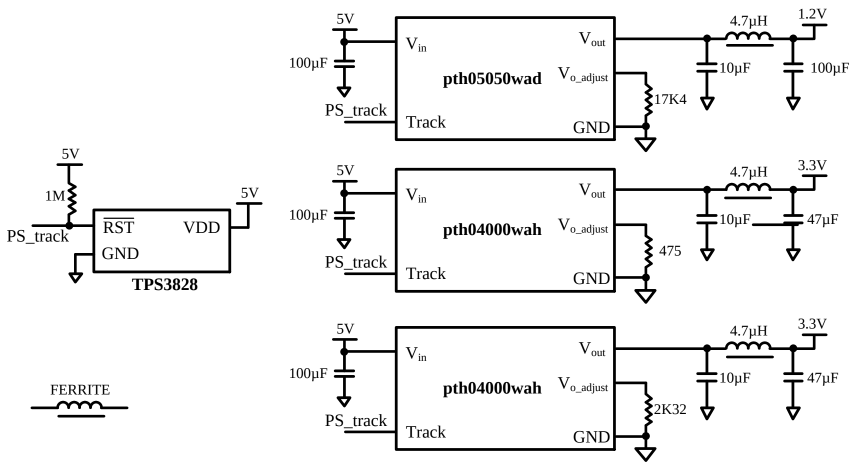


FIGURE 7.6: Block diagram for the Spartan-6 supply voltage generation.

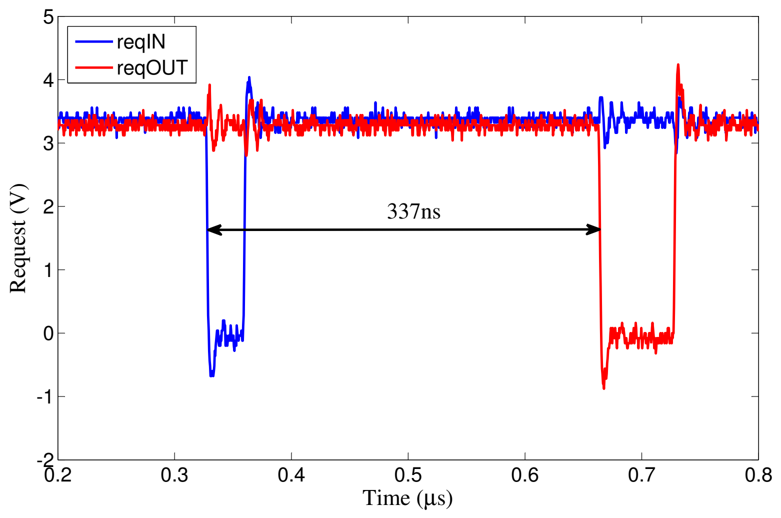
reaches its target value, the process is disabled and the regulator keeps this state independently of the voltage applied at pin *Track*. The process is governed by a shared signal *PS_track* generated by a supervisor circuit TPS3828. This circuit monitors the master 5V supply voltage and ties *PS_track* to ground while this voltage has not reached a 1.1V internal reference. When the threshold is reached, a pulse in *PS_track* triggers the system power up process after an internal fixed delay of 200ms. This way, all supply voltages are enabled at the same time.

7.3 Experimental results

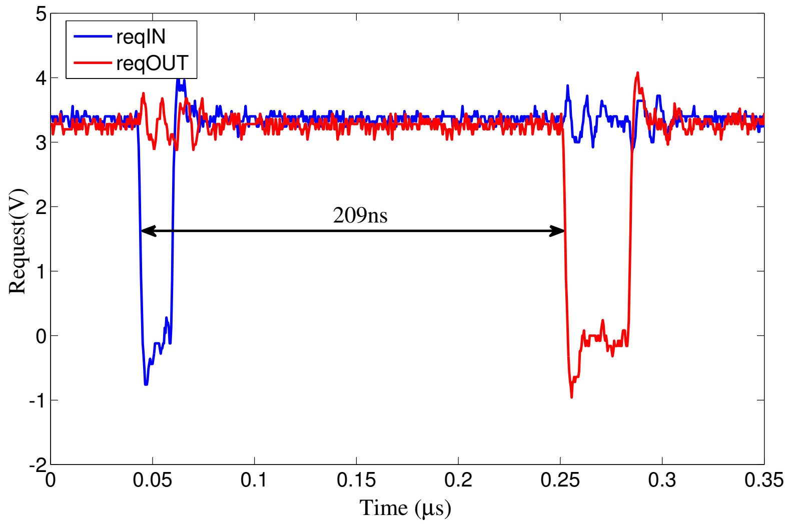
This Section presents some preliminary test results for a prototype board to check the Rocket I/O high speed serial link operation and the event transmission integrity. The test consists of generating a Rocket I/O wrapper for the FPGA high speed serial link to combine it with AER to serial protocol converters presented in Chapter 3-Fig. 3.13. This is the same experiment set to characterize the full-duplex Rocket-I/O-Based Parallel-Serial AER Interface in Chapter 3. We tested the link at two different bit rates, 1Gbps and 2.5Gbps, to observe differences in the performance which can be useful for future users of this board.

Fig. 7.7 illustrates the link latency measurement for both data rates. A very low event rate is programmed in the event generator which feeds the high speed serial interface. This way, we can determine the channel latency by measuring the delay of request signals at the transmitter input and receiver output. The 1Gbps interface presents 337ns of input to output latency, while the 2.5Gbps link reduces it down to 209ns. These latencies are obtained configuring 16 bits of word length in the serial/-parallel interface. If we measure the input handshaking cycle duration, instead the input to output request delay, the maximum event rate can be determined. It resulted to be 17Mevent/s for the 1Gbps link and 42Mevent/s for the 2.5Gbps interface.

In order to test the flow control mechanism, we forced an overflow situation in the link by setting different transmitting and receiving rates at the test circuitry. The acknowledge generation at the receiver side is slowed down and the transmitter is configured to send events at a high event rate. The receiver needs to stop the transmitter side when it detects that its buffer is saturating and the procedure must ensure the event flow integrity. This expected behavior can be confirmed by measuring the transmitter input and the receiver output request signals. Fig. 7.8 shows these waveforms for the 2.5Gbps interface. The error rate is monitored by on-FPGA test circuits to



(a)



(b)

Figure 7.7: Input to output request latency induced by the serial signal propagation for (a) 1Gbps and (b) 2.5Gbps.

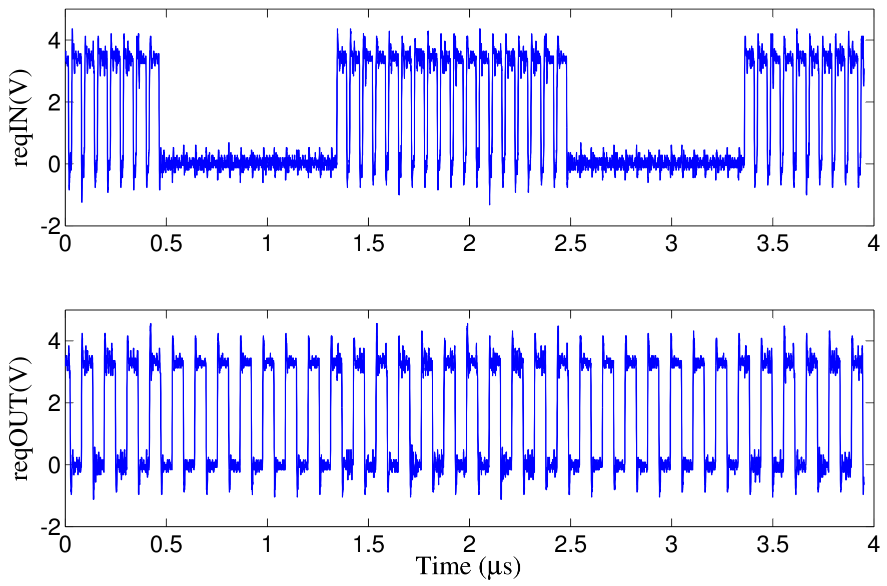


FIGURE 7.8: Flow control mechanism verification in the Node Board high speed serial links.

confirm the event stream integrity in this overflow situation.

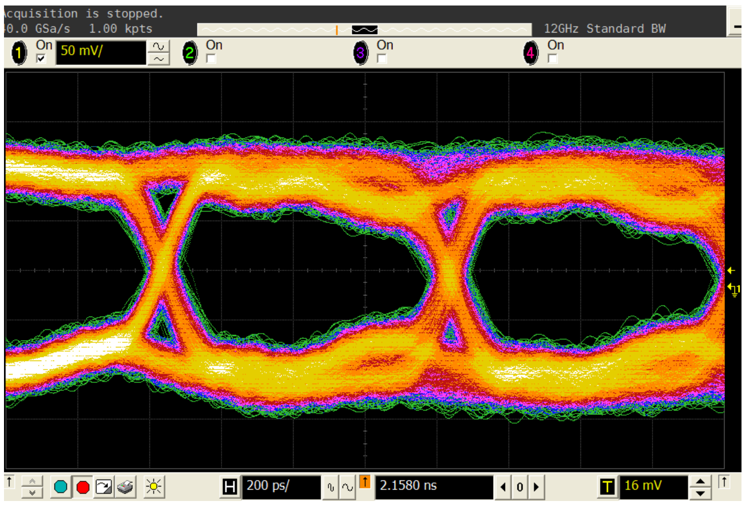
The eye diagram is the basic signal integrity measurement to ensure a low error rate event transmission. Fig. 7.9 shows the eye diagram for both tested bit rates. The eye is open in both situations, demonstrating that the physical design allows to achieve reasonable event rates with very low error rates. However, the eye for the 2.5Gbps bit rate is not as widely open as in the 1Gbps situation. For this experiment, the pre-emphasis and de-emphasis circuits were disabled and the differential amplitude was set to a value of 100mV. Fig. 7.10 shows the eye diagram obtained for the same bit rate, but the channel compensation circuits are enabled and the amplitude is increased. In this case, we can observe how the eye is clearly open. This is an example of the high degree of programmability offered by an FPGA solution and how it can be designed to improve the system performance.

7.4 Multiple NB Assembly

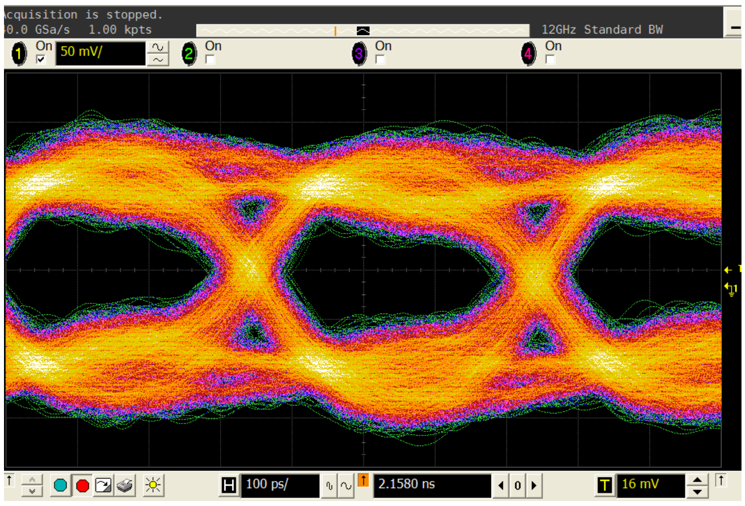
At the time of this writing only one preliminary NB has been available for a short time. The immediate future goal is to assemble 9 of them, as shown in Fig. 7.11. A 3×3 system can be easily assembled exploiting the board connectivity resources. A NB preliminary prototype photo represents the network units in Fig. 7.11 and board interconnection has been drawn over to give an idea of the system aspect. When enough boards are available, the connections between blocks would be implemented through

SATA cables.

As can be seen in Fig. 7.11, the system can be easily expanded by adding more NBs in idle links which are not connected through SATA cables. User only would have to re-configure the system to take into account these new blocks and run the experiments again. Parallel connectors at every network node are available to inject traffic in the network, collect output events or even establish internal test points to monitor the system state.



(a)



(b)

Figure 7.9: Eye diagram measured at receiver coupling capacitors for (a) 1Gbps and (b) 2.5Gbps.

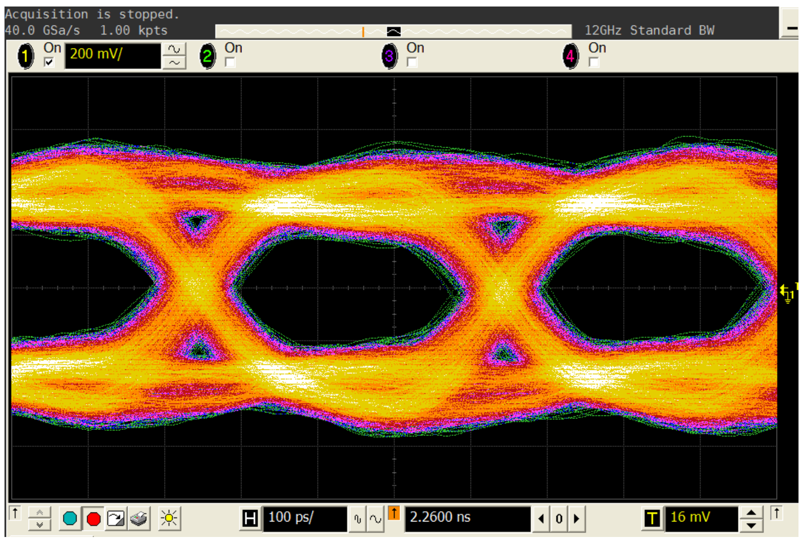


FIGURE 7.10: Improved eye diagram for the 2.5Gbps link using channel compensation circuits and transceiver programmability.

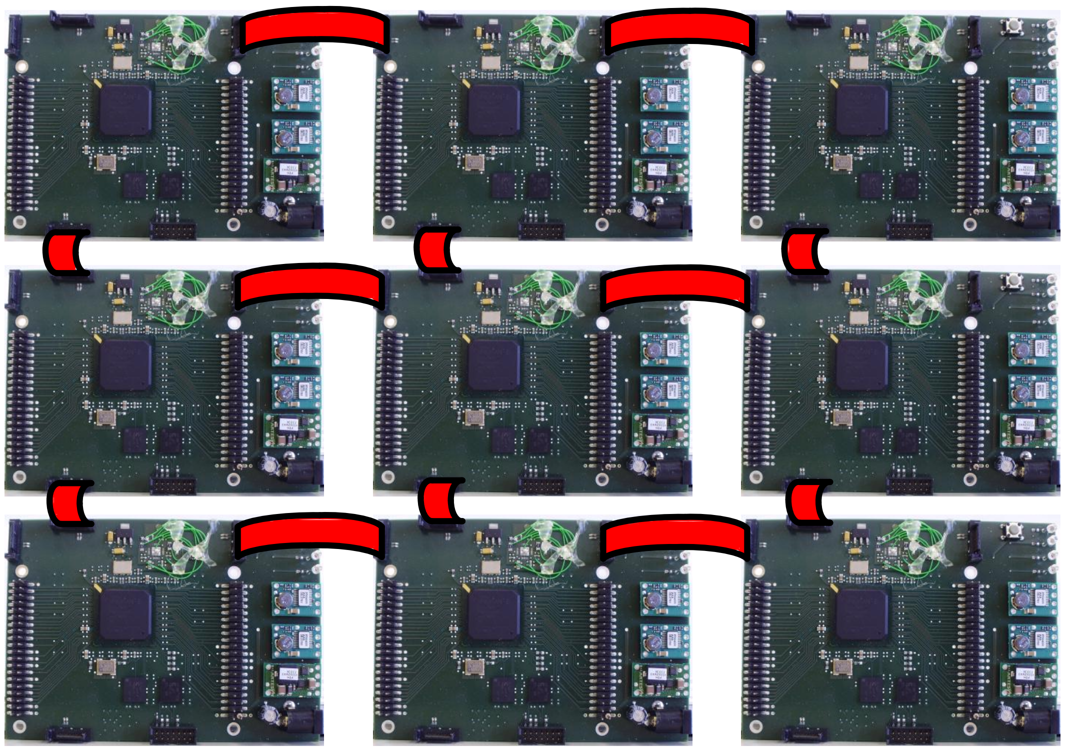


FIGURE 7.11: Photo montage of a 3 × 3 NB system connected through SATA cables.

7.5 Conclusion

A FPGA-based prototyping board design has been described in this Chapter. This node board can be assembled with its counterparts to form up large scale spike-based neural networks which can be tested and configured on the fly by programmable hardware resources. This platform has AER specialized connections ranging from classic parallel AER connectors for existing neuromorphic chips to high speed serial links for board to board connections. The user can plug his AER modules in a node board and build a connection network in the dedicated FPGAs using the router modules described in Chapter 3. Digital event processing blocks can be mapped into the FPGA along with router circuits due to the high integration density of the Spartan 6 FPGA. Input/output configuration ports can support low frequency configuration networks which can be added in parallel with the event communication network. SPI or JTAG networks can be designed using this configuration port and the reconfiguration capabilities of the FPGA hardware.

The event transmission layer is implemented with the Rocket I/O transceivers integrated in the Spartan 6 FPGA and hosted by SATA connectors and cables. The on-chip circuits provide channel coding, comma detection, pre/de-emphasis, channel alignment,. . . to support the full duplex serial AER channel described in Chapter 3. Clocking and power management resources have been included on-board to ensure a good signal integrity in the board to board communication. Very preliminary experimental results included in this Chapter demonstrates that the board is functional and all the features work properly.

8 Conclusions

Along this dissertation, properties of spike-driven processing have been exploited to increase the computational capabilities of traditional computer vision systems. Millions of neurons have to communicate in a network to perform sophisticated vision processing tasks which can be useful in real world environments. We took Convolutional Neural Networks as implementation example among other neural networks because of their good scalability properties and potential efficient hardware implementation. Moreover, spike-based convolution modules have been recently proposed [20–23] to overcome the limitations of their frame-based counterparts. They efficiently perform convolutions in real time operating over the input event stream and providing an output event flow. Assembling these modules in large scale systems represents the first step for building spike-based ConvNets.

A hardware infrastructure to implement event-driven large scale AER systems has been discussed in this dissertation. Starting from a system level design of a hierarchical, scalable and expandable AER network, we have proposed several basic building blocks which are essential to achieve a large scale of integration and low power consumption. Serializing the classic parallel AER link becomes a major issue to meet the scalability and reliability specifications of such a large scale neural systems. Moreover, a communication network layer for the system organization is mandatory to achieve large connectivity densities with minimum hardware resources. The main contributions of this work can be summarized as follows:

- A communication layer for scalable AER networks has been designed to assemble convolution chips in large scale systems. The most suitable topologies, addressing schemes and routing algorithms have been discussed in Chapter 3. Destination and source driven router hardware architectures have been proposed to manage the AER traffic in the network nodes. The pros and cons of every solution have also been analyzed to clarify their field of application. The system level design has been validated in an FPGA prototyping platform where large networks of convolution modules have been mapped to perform some feature extraction and very simple recognition tasks. Extensive experimental results are provided for a NoC realization and a multi-board extension.

- This multi-board design is supported by a full-duplex AER high speed serial link based on the Xilinx Rocket I/O technology. The serial link provides high communication bandwidths to multiplex several parallel AER links over a single differential pair. A flow control mechanism through the full-duplex link is also embedded into the event data transmission to prevent a slow AER receiver to get overflown by a fast transmitter. Up to 50Mevent/s rates have been obtained for 32 bit events using this solution with 232ns input to output latencies.
- This FPGA-based serial link has important limitations when the architecture wants to be mapped on a VLSI chip. Due to the asynchronous nature of AER events, the link is idle most of the time but it keeps transmitting data for synchronization purposes. A more power efficient solution has been designed by exploiting the asynchronous nature of AER events with a burst-mode Serialization/Deserialization VLSI circuit. Although there are burst-mode designs reported in the literature, they are not optimized to transmit AER data because they suffer from long latencies, high complexity, high power consumption or stringent jitter budgets. In Chapter 4, we propose a novel instant-startup jitter-tolerant Manchester-encoding Serializer/Deserializer scheme which overcomes the limitations of conventional burst-mode serial links. A $0.35\mu m$ CMOS proof of concept prototype has been fabricated and tested. The circuit achieves a maximum event rate of 15Mevent/s with 66ns of input to output latency, while scaling the power consumption with the event rate. This is a feature which is not found in other burst mode Serializer/Deserializer circuits.
- One of the most important limitations in terms of energy efficiency of the prototype reported in Chapter 4 is that a conventional current mode LVDS driver and receiver pair is used. These circuits consume a significant amount of static power during pauses, even when there are no data being transmitted. Pauses have been used to save power in these circuits by switching off this static power consumption when it is not needed. However, the driver and receiver design has to be modified in order to perform this task without trading off with the maximum achievable throughput. Extremely fast switch on/off times are required to keep the maximum event rate unaltered and scale down the power consumption with event rate. Chapter 5 shows experimental results for a $0.35\mu m$ CMOS prototype designed to accomplish the design goals. At a 500Mbps bit rate, the maximum event rate achievable is 4.7Mevent/s for 32-bit events. In this situation, current consumption is 7.5mA and 9.6mA for the driver and receiver, respectively. However, if event rate is lower than 20-30Kevent/s, current consumption has a floor of 270μA for the driver and 570μA for the receiver. The measured switching on/off times are in the order of 1ns, while the whole event takes 82ns to be transmitted. The serial link is operative up to a 710Mbps bit rate, which corresponds to a maximum event rate of 15Mevent/s. The maximum measured event rate is 13.7Mevent/s which corresponds to a bit rate of 638Mbps and 73ns of input to output latency.

- The work done for current mode drivers has been extended for voltage mode circuits too. The voltage mode approach presents better power efficiency in short-haul links where very high data rates are not required. In order to further reduce the power consumption, we have investigated switching implementations of voltage mode drivers suitable for AER links. Techniques for quickly switching on/off the driver, achieving accurate impedance matching and managing the driver bias currents are presented throughout Chapter 6. A 0.35μm CMOS prototype with the voltage mode high speed serial I/O circuits has been designed and tested. It can scale the current consumption from 7.7mA (driver) and 8.2mA (receiver) at 10Mevent/s rate to 343μA (driver) and 62.5μA (receiver) for an event rate below 10Kevent/s. These measurements correspond to a 500Mbps bit rate and 32 bits events. In this case, the link worked properly at a maximum bit rate of 638Mbps, with an input to output request latency of 73ns.
- A new FPGA-based board (Node Board) for large scale AER systems has been designed as a prototyping tool to rapidly map software event-based ConvNets into a hardware platform. The FPGA can host several digital event-based modules or it can communicate with analog/digital neuromorphic chips mounted on extension boards. The NB has resources to implement four high speed full-duplex serial channels to receive/transmit events from/to the board neighborhood and to propagate configuration parameters along the network. The design guidelines and board description are included in Chapter 7, along with some preliminary experimental results.

This dissertation paves the way for the design of large scale spike-based ConvNets hardware, but there is still a lot of work to do in this field. This work is going in two different but convergent directions. On one side, there is a need for a strong theoretical background about the design of event-driven sophisticated ConvNets in terms of novel architectures and convolution modules system level design. On the other side, further improvements are possible in the hardware units which will host the convolution modules. The future work lines are listed below:

- There is still room for improvement in the Serialization/Deserialization scheme proposed in Chapter 4. Simple design modifications to switch off some unneeded circuitry can greatly reduce the idle power consumption. The serializer idle power consumption is due to useless clock transitions which occur during pauses when the circuit state does not change. Incorporating the clock gating technique to the serializer design will push the low event rate current consumption in the pA range. Finally, the ADC-DAC structure area occupation can be reduced significantly by using an asynchronous converter scheme which can reduce the comparator-count.
- A problem which has not been addressed yet is the clock generation for burst mode serial links. As the Serializer/Deserializer architecture does not have stringent jitter requirements, some novel low cost clocking strategies can be proposed to achieve low power consumption and avoid the need of an external clock reference. That will reduce the hardware complexity of a multi-chip assembly on a PCB. Moreover, the idea of switching off unused components during pauses to

save power is also suitable for the clocking circuit. However, this should be done without affecting the throughput by using fast settling clocks.

- The current mode driver proposed in Chapter 5 can be improved in future implementations. Reducing the static power consumption is the first goal in this design. Improving the receiver bias circuits is the easiest way to do that, but other low power common mode feedback loops for the driver can also be considered as an option. On the other hand, reducing the number of low-speed signals used for the flow control and switching arbitration would be very interesting. That will reduce the convolution modules pin-count and will improve the system level integration. Some common mode signaling technique can be investigated to indicate the receiver where to switch on and the flow control mechanism can rely on the full-duplex link, like in the FPGA implementation.
- The voltage mode implementation calibration loop does not properly track the temperature variations as calibration is only performed at power up and takes hundred of μs. Proposing re-calibration procedures which track the differential amplitude without affecting the driver throughput will be highly desirable to improve the system reliability. Reducing the quiescent current of the regulator which generates the driver supply voltage will improve the link energy efficiency by lowering the idle static current consumption.
- However, the most interesting area of future work would be in exploiting the opportunities offered by the Node Board. Although the NB hardware architecture has been preliminarily validated with experimental results in Chapter 7, features included in this hardware architecture have not been sufficiently exploited. The next step will be mapping large scale AER systems on this architecture and explore possible problems or limitations which can arise in a hardware implementation. That will give us an important insight of the working principles of spike-based neural networks and all this experience can be applied to the final VLSI implementation. All the information collected in this design step will be very useful in subsequent VLSI implementations which can target practical applications.

9
List of Publications

Journal papers:

- C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. "Low Power Switchable Voltage Mode High Speed Serial Link for Modular Scalable AER Chip Grids", *IEEE Journal of Solid-State Circuits*, submitted for publication.
- C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. "A 0.35μm Sub-ns Wake-up Time ON-OFF Switchable LVDS Driver-Receiver Pair for Rate-Dependent Power Saving in AER Bit-Serial Links", *IEEE Transactions on Biomedical Circuits and Systems*, submitted for publication.
- C. Zamarreño Ramos, A. Linares Barranco, T. Serrano Gotarredona and B. Linares Barranco. " 'Pre-Structured Mesh AER': A Multi-Casting Modular and Scalable Assembly Approach for Reconfigurable Neuromorphic Pre-Structured AER Systems", *IEEE Transactions on Biomedical Circuits and Systems*, under review.
- L. Camuñas Mesa, C. Zamarreño-Ramos, A. Linares-Barranco, A. Acosta-Jiménez, T. Serrano-Gotarredona and B. Linares-Barranco. "An Event-Driven Multi-Kernel Convolution Processor Module for Event-Driven Vision Sensors", *IEEE Journal of Solid-State Circuits*, in Press.
- C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. "An Instant-Startup Jitter-Tolerant Manchester-Encoding Serializer/Deserializer Scheme for Event-Driven Bit-Serial LVDS Interchip AER Links", *IEEE Transactions on Circuits and Systems I: Regular Papers*, in Press.
- C. Zamarreño Ramos, L. A. Camuñas Mesa, Jose A. Pérez Carrasco, T. Masquelier, T. Serrano Gotarredona, and B. Linares Barranco, "On Spike Timing Dependent Plasticity, Memristive Devices, and Building a Self-Learning Visual Cortex," *Frontiers in Neuromorphic Engineering (inaugural issue)*, /Front. Neurosci./ *5:*26, 2011. doi: 10.3389/fnins.2011.00026, 17 March 2011.

- L. Camuñas Mesa, A. Acosta Jiménez, C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “A 32x32 Pixel Convolution Processor Chip for Address Event Vision Sensors with 155ns Event Latency and 20Meps Throughput”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol.58, no.4, pp.777-790, April 2011.

Conference proceedings

- C. Zamarreño Ramos, R. Kulkarni, T. Serrano Gotarredona, J. Silva Martínez and B. Linares Barranco. “Voltage mode driver for low power transmission of high speed serial AER link”. *IEEE International Symposium on Circuits and Systems (ISCAS 2011)*, pp. 2433-2436, 15-18 May 2011.
- L. Camuñas Mesa, J.A. Pérez Carrasco, C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “On scalable spiking CovNet hardware for cortex-like visual sensory processing systems”. *IEEE International Symposium on Circuits and Systems (ISCAS 2010)*, pp. 1659-1662, May 30 2010-June 2 2010.
- J.A. Pérez Carrasco, C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “On neuromorphic spiking architectures for asynchronous STDP memristive devices”. *IEEE International Symposium on Circuits and Systems (ISCAS 2010)*, pp. 1659-1662, May 30 2010-June 2 2010.
- L. Camuñas Mesa, J.A. Pérez Carrasco, C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “Neocortical frame-free vision system and processing through scalable spiking CovNet Hardware”. *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp.1-8, 18-23 July 2010.
- C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “OTA-C oscillator with low frequency variations for on-chip clock generation in serial LVDS-AER links”. *IEEE International Symposium on Circuits and Systems (ISCAS 2009)*, pp. 2657-2660, 24-27 May 2009.
- C. Zamarreño-Ramos, T. Serrano-Gotarredona and B. Linares-Barranco. “Low power LVDS transceiver for AER links with burst mode operation capability”. *Conference on Design of Circuits and Integrated Systems (DCIS 2009).*
- C. Zamarreño Ramos, T. Serrano Gotarredona and B. Linares Barranco. “OTA-C oscillator with low process and temperature variations for on-chip frequency reference in serial LVDS-AER links”. *Conference on Design of Circuits and Integrated Systems (DCIS 2008).*
- C. Zamarreño Ramos, R. Serrano Gotarredona, T. Serrano Gotarredona and B. Linares Barranco. “LVDS interface for AER links with burst mode operation capability”. *IEEE International Symposium on Circuits and Systems (ISCAS 2008)*, pp. 644-647, 18-21 May 2008.

10
Appendix I: AER convolution modules in FPGA

10.1 Introduction

This Appendix provides the VHDL description [1] of the FPGA-based convolution module used in Chapter 3. This block has been synthesized along with the source and destination driven routers to test the system level behavior of large scale AER based convolution systems. This is a very lightweight implementation of a convolution processor which follows a sequential approach to compute the operations associated with every input event. When a new input event is received, the processor fetches data from a neuron state RAM and a kernel weights RAM, depending on the input address. When kernel and neuron status are ready, the integrate and fire operation is performed for every neuron in the projection field (kernel range) of the input event address: the weight is added to the previous neuron state and, if an upper or lower threshold is reached, a signed output event is fired. The computation cycle is repeated neuron by neuron for all neurons within the kernel projection field of the input event.

In previously reported VLSI implementations [20–23], the computation is performed row by row and in a pipeline way. This reduces input event processing latency to a few hundred of ns. In the present FPGA-based sequential implementation we suffer from latencies in the order of $3\mu s$. However, this particular sequential approach allows a high scale of integration of convolution modules in the same FPGA optimizing the hardware resources needed. As we wanted to test large scale systems implement on a single FPGA or multiple FPGAs environment, we chose this sequential implementation sacrificing processing speed. A more detailed description of the sequential convolver features will be provided in this Appendix. Moreover, the VHDL code used for the

[1]This source code is available upon request to bernabe@imse-cnm.csic.es

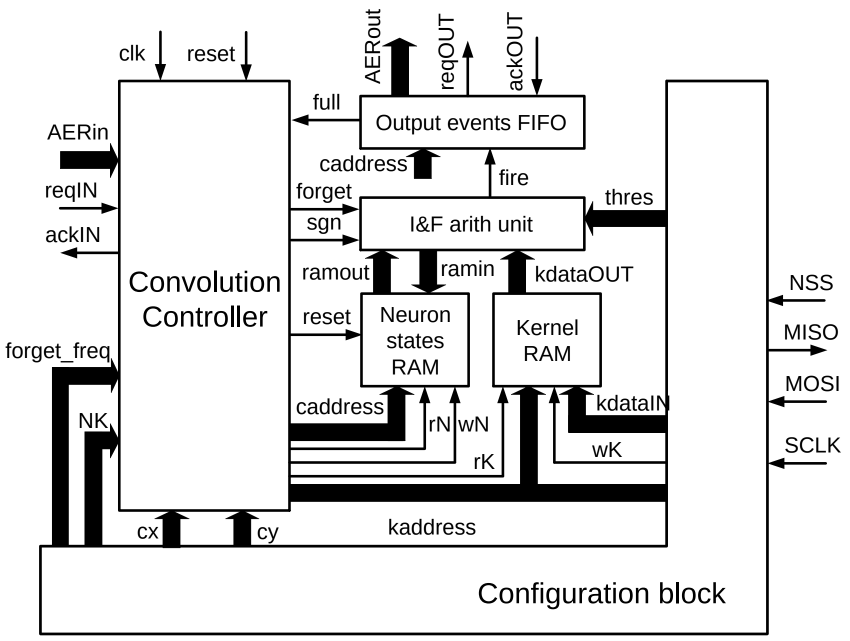


Figure 10.1: Block diagram for the sequential AER convolution block.

experiments discussed in Chapter 3 will be discussed.

10.2 Sequential convolution module

The sequential approach is based on an intensive use of the FPGA internal RAM memory. In VLSI convolution chips, a 2D array of silicon neurons implement the integrate and fire operation with dedicated and independent hardware resources. Having such a multiplicity of resources per neuron in an FPGA implementation is highly inefficient in terms of slices occupation per convolution block. In the present VHDL description, all neural states are stored in a "Neuron State RAM", the kernel is stored in a "Kernel RAM", and all arithmetic operations are performed by one single "Integrate-and-fire Arithmetic Unit", which updates the desired neurons' state sequentially. This results in a slower convolution processor, but with a more efficient resource allocation within an FPGA. For example, a convolution module composed by a 64x64 array of neurons, a maximum kernel of 11x11 elements and a data codification of 8 bits needs up to 33Kbytes. The Virtex-6 FPGA used in the experiments described in Chapter 3 have 29952Kbytes, leading to a theoretical maximum number of convolution units of 907. Hence, the most limiting factor is not the FPGA internal memory resources, but the slices occupied by the logic.

Fig. 10.1 shows the hardware architecture of the sequential AER convolution block. We can distinguish the next main building blocks in this diagram:

- **Kernel and neuron states RAMs:** they store the kernel weights and neurons' state values for the whole convolution module. They are mapped into internal FPGA RAM resources.
- **Convolution controller:** manages the input AER port and controls the sequential convolution operation through a synchronous FSM. It generates the kernel and neuron state memory addresses to apply the stored kernel over the projective field given by the input spike address. Moreover, it manages the forgetting effect through an internal counter which indicates when the neurons must leak. It triggers a process where the whole neuron state memory is read and its content is modified according to the chosen leak rate.
- **I&F arithmetic unit:** operates over the data provided by the kernel and neuron state memory by adding/subtracting both data depending on the input event sign. If the result of this arithmetic operation reaches the positive/negative threshold (both are symmetric with respect to zero), an output event is fired and the state of the neuron is set to zero. Otherwise, the result of the operation is updated in the neuron state memory. This operation is performed for every neuron within the projection field of the input event.
- **Output events FIFO:** when the I&F arithmetic block decides to send out an event, this event address and sign is written on an output FIFO, to not slow down the convolution computation with the output handshaking. This FIFO block also manages the output AER protocol and generates a *full* signal for the controller if the FIFO is reading an overflow condition. In this case, the controller stops the computation until the FIFO is ready to store new events.
- **Configuration block:** configuration parameters are received through an SPI interface and decoded by a configuration block. The first task of this block consists of writing the kernel memory with the kernel weights. Apart from the kernel, the convolution units receive the following parameters:
  - *thres*: neurons firing threshold.
  - *cx,cy*: offset which modifies to the input address. The kernel will be applied over the address given by "input x-address" + "*cx*" and "input y-address" + "*cy*", where *cx*-*cy* can be either positive or negative.
  - *forgetfreq:* this value is compared with an internal counter to generate the forgetting pulses with a desired frequency. This forgetting period is expressed in clock cycles.
  - *NK*: number of kernel rows/columns. Kernels are always considered to be squared.

10.3 VHDL code for the convolution block

The VHDL code for the convolution block described in Chapter 3 is listed below. Each Subsection corresponds to different parts of the convolution block hierarchy. The input event and forgetting effect management and the arithmetic units are described in a top module, along with some configuration resources. The auxiliary blocks are described in separated files which implement generic RAMs, FIFOs and SPI decoding blocks. The spike-based convolution module VHDL description is available upon request (bernabe@imse-cnm.csic.es).

10.3.1 Entity declaration

The entity declaration shows the input/output signals of the top module. Signals (*aer_in_data*, *aer_in_req_l*, *aer_in_ack_l*) and (*aer_out_data*, *aer_out_req_l*, *aer_out_ack_l*) represent the input and output ports, respectively. Signals *NSS*, *SCLK*, *MOSI*, *MISO* and *PD* correspond to the SPI interface which provides the configuration information. Finally, the convolution unit provides a configuration interface with the event router through signals *xADD*, *yADD*, *RTdata*, *reqW*, *ackW* and *RTadd*.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity aer_conv_random is
Port (
clk : in std_logic;
rst : in std_logic;
led : out std_logic_vector( 5 downto 0 );
-- AER input
aer_in_data : in std_logic_vector( 15 downto 0 );
aer_in_req_l : in std_logic;
aer_in_ack_l : out std_logic;
-- AER output
aer_out_req_l : out std_logic;
aer_out_ack_l : in std_logic;
aer_out_data : out std_logic_vector( 15 downto 0 );
-- Micro interface
MICRORST: in std_logic;
STR : out  STD_LOGIC;
NSS : in  STD_LOGIC;
SCLK : in  STD_LOGIC;
MOSI : in  STD_LOGIC;
MISO : out  STD_LOGIC;
PD: in std_logic;
test: out std_logic;
xADD : out std_logic_vector(3 downto 0);
yADD : out std_logic_vector(3 downto 0);
RTdata : out std_logic_vector(7 downto 0);
reqW : out std_logic;
ackW : in std_logic;
RTadd : out std_logic_vector(5 downto 0);
-- For testing purpose
Aux : out std_logic_vector (35 downto 0)
);
end aer_conv_random;
```

10.3.2 Blocks and signals declaration

The current convolution module uses RAM memories to store the kernel weights and neurons's state. Output events are written in a FIFO before sending them to other modules using the AER protocol. The configuration parameters are fed into the convolution module through an SPI interface. All these tasks are performed by auxiliary blocks, whose description will be provided in subsequent Subsections. Internal signals definitions are also included below.

```
architecture Behavioral of aer_conv_random is
   component SPI_SLAVE
    Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           STR : out  STD_LOGIC;
           NSS : in  STD_LOGIC;
           SCLK : in  STD_LOGIC;
           MOSI : in  STD_LOGIC;
           MISO : out  STD_LOGIC;
  WR: out STD_LOGIC;
  ADDRESS: out STD_LOGIC_VECTOR(7 downto 0);
  DATA_OUT: out STD_LOGIC_VECTOR(7 downto 0));
   end component;

 COMPONENT confROUTER
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         SPIdata : IN  std_logic_vector(7 downto 0);
         SPIvalid : IN  std_logic;
         PD : IN  std_logic;
         xADD : OUT  std_logic_vector(3 downto 0);
         yADD : OUT  std_logic_vector(3 downto 0);
         RTdata : OUT  std_logic_vector(7 downto 0);
         RTadd : OUT  std_logic_vector(5 downto 0);
         reqW : OUT  std_logic;
         ackW : IN  std_logic
        );
    END COMPONENT;

component conv_matrix
   generic(RAM_Size: in integer;
     RAM_Addr_Size: in integer;
     RAM_Data_Size: in integer);
    Port ( address : in std_logic_vector(RAM_Addr_Size -1 downto 0);
           data : in std_logic_vector(RAM_Data_Size-1 downto 0);
           ramout: out std_logic_vector(RAM_Data_Size-1 downto 0);
  wrram: in std_logic;
  enable: in std_logic;
  RST_N: in std_logic;
  CLK: in std_logic
  );
   end component;

COMPONENT aer_out
PORT(
REQ_N : OUT std_logic;
ACK_N : IN std_logic;
Data_AER : OUT std_logic_vector(15 downto 0);
Data_Fifo : IN std_logic_vector(15 downto 0);
Enable_Fifo : IN std_logic;
RD_Fifo : OUT std_logic;
CLK : IN std_logic;
RST_N : IN std_logic
);
END COMPONENT;

COMPONENT ramfifo
generic (TAM: in integer; IL: in integer; WL: in integer);
PORT(
clk : IN std_logic;
wr : IN std_logic;
rd : IN std_logic;
rst_n : IN std_logic;
empty : OUT std_logic;
full : OUT std_logic;
data_in : IN std_logic_vector(WL-1 downto 0);
data_out : OUT std_logic_vector(WL-1 downto 0);
mem_used: out std_logic_vector(IL-1 downto 0)
);
END COMPONENT;

   component spblockram
   generic (RAM_Size: in integer:= 128;
      RAM_Addr_Size: in integer:=7;
      RAM_Data_Size: in integer:=8);
   port (clk : in std_logic;
    we  : in std_logic;
    a   : in std_logic_vector(RAM_Addr_Size-1 downto 0);
    di  : in std_logic_vector(RAM_Data_Size-1 downto 0);
    do  : out std_logic_vector(RAM_Data_Size-1 downto 0));
   end component;

  signal iaer_in_req_l,  saer_in_req_l: std_logic;
  signal iaer_out_ack_l, saer_out_ack_l: std_logic;
  SIGNAL maddress :  std_logic_vector(6 downto 0);
  SIGNAL enable :  std_logic;
  SIGNAL emite, remite :  std_logic;
  SIGNAL evento :  std_logic_vector(11 downto 0);
  signal fifo_wr, fifo_rd, fifo_empty, fifo_full: std_logic;
  signal fifo_data_in, fifo_data_out: std_logic_vector(15 downto 0);
  signal not_fifo_empty: std_logic;
  signal mem_used: std_logic_vector(3 downto 0); --
  signal ramout: std_logic_vector (7 downto 0);
  signal caddress: std_logic_vector (11 downto 0);
  signal kaddress,keraddress: std_logic_vector (6 downto 0);
  signal NK: integer range 0 to 11;
  signal write_conv,ker_we: std_logic;
  signal kerdata,kerdata_prev: std_logic_vector (7 downto 0);
  signal ca2Aux : std_logic_vector(7 downto 0);
  signal cdata: std_logic_vector (7 downto 0);
  signal init_we: std_logic;
  signal aerin_lfsr, aerin_we : std_logic;
  signal forget,start_forget,end_forget,read_forget,write_forget: std_logic;
  signal waitconv:std_logic;
  signal init: std_logic;
  signal wrram: std_logic;
  signal kk: std_logic_vector (15 downto 0);
  signal tparam: integer range 0 to 15;
  signal ker_kk: std_logic_vector (7 downto 0);
  signal tker_we: std_logic;
  signal end_ker: std_logic;
  signal toforget: std_logic_vector (32 downto 0);
  signal am_forget: std_logic_vector(8 downto 0);
  signal thresold: std_logic_vector (7 downto 0);
  signal cnt_sm: integer range 0 to 31;
  signal tled,nled: std_logic_vector (2 downto 0);
  signal pre_spi_valid,spi_valid, latched_spd, spd, pd0: std_logic;
  signal spi_address: std_logic_vector (7 downto 0);
  signal pre_spi_data,spi_data,ker_data_in: std_logic_vector (7 downto 0);
  signal rst_l,hardreset,softreset,init_soft: std_logic;
  signal debug : std_logic_vector (11 downto 0);
  signal n_ProgKernel,ProgKernel : std_logic;
  signal countA,n_countA,countB,n_countB : std_logic_vector(3 downto 0);
  signal fire_level : std_logic_vector(8 downto 0);
  signal endINker : std_logic;
  signal cx,cy : std_logic_vector(6 downto 0);
  signal neuron : std_logic_vector(8 downto 0);

begin
Aux(15 downto 0) <= kerdata(7 downto 0) & keraddress(6 downto 0) & ker_we;
```

```
   rst_l <= not rst and microrst; -- and softreset;

   --led <= tled & nled;
--user configure the idle level in variable tresold, and fire level is
--internally calculated to obtain a double threshold for charging and
-- discharging process
fire_level(7 downto 1) <= thresold(6 downto 0);
fire_level(0) <= '0';
fire_level(8) <= '0';
```

10.3.3 Forgetting effect FSM and configuration parameter assignation

The forgetting effect FSM (Finite State Machine) manages the forgetting frequency, which is set by a configuration parameter. When a counter reaches a certain programmable threshold, a forgetting pulse is generated and other process is in charge of updating the neuron state memory with a certain fixed leak. Moreover, the configuration parameters are assigned with the information decoded by the SPI block.

```
   B_SPI_Y_Forgeting: process (rst_l,clk)
-- This process is in charge of receiving parameters from the SPI interface.
-- SPI address are ignored and SPI data bytes are expected to be received
--in this order: first 4 bytes for the forgetting counter threshold, then
-- 1 byte for the amount to be forgotten, then 1 byte for the size of the
-- kernel, then 1 control byte that is able to reset the convolver, and
-- then one byte for setting the global threshold for all the IF neurons
-- in order to control the output AER traffic generation.
  -- All these configuration parameters are received from the microcontroller
  -- through the parameters of an  ATC command, so PD='0'.
--
-- This process is also in charge of managing the forgetting counter
-- and signaling.

   variable cnt_forget: std_logic_vector (32 downto 0);
   begin
   if rst_l='0' then
    softreset<='1'; hardreset<='1'; cnt_forget := (others=>'0'); start_forget <= '0';
    cnt_sm <= 0; toforget <= "0" & x"0007A120";  am_forget <= "0" & x"00"; --255;
    thresold <= "01111111"; nled(0) <= '1'; nled(2) <= '1';
    NK <= 11; cx <= (others => '0'); cy <= (others => '0');
      elsif clk'event and clk='1' then
       if spd='0' and latched_spd='1' then
   cnt_sm<=0;
   elsif spd = '0' and spi_valid='1' and cnt_sm<16 then
    cnt_sm <= cnt_sm +1 ;
    case cnt_sm is
    when 0 => toforget (7 downto 0) <= spi_data;
    when 1 => toforget (15 downto 8) <= spi_data;
    when 2 => toforget (23 downto 16) <= spi_data;
    nled(0) <= not nled(0);
    when 3 => toforget (31 downto 24) <= spi_data;
    when 4 => am_forget <= "0" & spi_data;
    when 5 => NK <= conv_integer(spi_data);
    when 6 => if spi_data = x"AA" then softreset<='0';
    end if;
    when 7 => thresold <= spi_data;
    when 8 => cx <= spi_data(6 downto 0);
    when 9 => cy <= spi_data(6 downto 0);
    when others=> cnt_forget:=(others=>'0');
    end case;
elsif spd='1' and cnt_forget>=toforget then
```

```
  start_forget <= '1'; cnt_forget := (others=>'0');
  led(2) <= not nled(2); softreset<='1';
elsif spd='1' and cnt_forget < toforget and forget = '0' then
start_forget <= '0';    cnt_forget := cnt_forget +1; softreset<='1';
else start_forget <= '0'; softreset<='1'; hardreset<='1';
end if;
end if;
end process;
```

10.3.4 Neuron State Update FSM

This FSM manages the neuron array updating when a new input event arrives or there is a forgetting pulse. The neuron state and kernel RAM addresses are calculated depending on the input event address or the last neuron affected by the forgetting leakage. Moreover, the arithmetic operations for the Integrate&Fire neurons are also implemented in this FSM. The RAM addresses management and the neuron's state updating are coordinated by a counter which coordinate all these operations to properly perform the convolution operation. When a forgetting pulse arrives, the FSM goes around the neuron state RAM position by position subtracting a programmable quantity to the stored neuron's states.

```
   B_conv: process (rst_l, clk)
--
-- This big process is responsible of the correct processing of the
-- 64x64 convolution result. Therefore, several operations are managed here:
--  When a forgetting operation is signaled, this process access all the
-- arrays elements and decrement the forgetting amount.
--  When a soft reset is received, this process initializes the
--    convolution matrix.
--  When a new AER input event is received, this process access both the
--    kernel matrix and the corresponding neighborhood of the convolution
--    matrix sequentially in order to add the kernel weights to the correct
--    convolution matrix elements. If the event is negative, CA2 version of
--    the kernel is added
--  When the addition operation is performed if the result is greater than
--    the global threshold, an output  AER event is produced and sent to
--    the AER out fifo.

  variable cnt: integer range 0 to 512;
variable pi,pj,ki,kj,ni,nj: integer range 0 to 15;
variable imi,imj: integer range 0 to 63;
variable cnt_init: integer range 0 to 8191;
variable suma: integer range -1024 to 1023;
variable tmp: std_logic_vector (7 downto 0);
variable forgetvalue, ramvalue: integer range 0 to 511;
variable caddressx,caddressy: integer range 0 to 63;
variable kaddressv: integer range 0 to 127;

function add_ker(n: in integer) return integer is
   variable o: integer;
begin
  case n is
    when 5 => o := 55;
 when 4 => o := 44;
 when 3 => o := 33;
 when 2 => o := 22;
 when 1 => o := 11;
 when others => o:= 0;
        end case;
  return o;
```

```
      end function;
begin
  if rst_l='0' then
       cnt:=0; end_ker <= '0'; init <= '1'; cnt_init := 0;
       init_we <= '0'; aer_in_ack_l <= '1';
   aerin_lfsr <= '0'; aerin_we <= '0'; end_forget <= '1';
   read_forget <= '0'; write_forget <= '0'; waitconv<='0';
   init_soft<='0'; nled(1)<='0'; forget <='0'; test<='0';
   tled<="000"; emite <= '0';
   led <= (others => '0');
  elsif clk'event and clk='1' then
    emite <= '0';
 if softreset='0' then -- Initialization by a software usb command
   init_soft<='1'; cnt_init:=0;
 end if;
  -- The forgetting timer reaches the forgetting state.
  -- The whole RAM is processed.
    if start_forget='1' then
    cnt_init := 0; read_forget <= '0'; write_forget <= '0';
    end_forget <= '0'; forget <= '1'; test <= '1';
  -- Forgetting ends when end_forget is set high
      elsif end_forget ='1' and forget='1' then
 forget <= '0'; test <='0';
       end if;
     -- Initialize the conv-RAM with the default threshold.
 if (init='1' or init_soft='1') and ProgKernel = '0' then
    aerin_lfsr<='1'; aerin_we <= '0'; write_conv <= '0';
    cnt:=0;    init_we <= '1';
    caddress <= conv_std_logic_vector (cnt_init,12);
 cdata <= thresold;
 if cnt_init < 4095 then cnt_init := cnt_init+1;
          else init<='0'; init_soft<='0';
 end if;
 -- Forgetting has less priority than initialization.
 elsif forget='1' and read_forget='0' and write_forget='0' then
 -- Forgetting implies to access one by one all the address
 -- of the conv-RAM. In this step the address is written on the
 -- address bus.
    caddress <= conv_std_logic_vector (cnt_init,12);
 init_we<='0'; read_forget <= '1'; aerin_lfsr <= '1';
 aerin_we <= '0'; write_conv <= '0';
 if (cnt_init=1) then nled(1) <= not nled(1);
 end if;
 elsif forget='1' and read_forget='1' and write_forget='0' then
 -- In this step, a read operation is due in the conv-RAM and
 -- a new value is calculated, by subtracting the am_forget.
 -- Negative values are not allowed.
 init_we<='1'; write_forget<= '1'; aerin_lfsr <= '1';
 aerin_we <= '0'; write_conv <= '0';
 forgetvalue:=conv_integer(am_forget); ramvalue:= conv_integer(ramout);
  suma:= ramvalue - forgetvalue;
 --set idle value when negative threshold is overflown
 if suma <= 0 then   cdata <= thresold;
 else          cdata <= conv_std_logic_vector(suma,8);
 end if;
       elsif forget='1' and read_forget='1' and write_forget='1' then
 -- In this step the counter for addressing the conv-RAM is
 --- incremented by one. One cycle wait state is necessary
 -- to conclude the write operation in the conv-RAM
 init_we <= '1';  read_forget <= '0';  write_forget <= '0';
 aerin_lfsr <= '1'; aerin_we <= '0';
 write_conv <= '0';
 if cnt_init < 4095 then cnt_init := cnt_init+1;
          else  end_forget <= '1';  cnt_init:=0;
 end if;
    end if;
   -- If no initialization nor forgetting states, normal operation state.
 if init='0' and forget='0' and init_soft='0' then
```

```
  if saer_in_req_l='0' and cnt<2 and end_ker='0' and spi_valid='0' then
  --If a new AER_in arrives, kernel matrix should be added to conv-RAM
  -- around the cell of the AER received. Several alternatives are possible
  -- depending on the cell position in the conv-RAM due to border conditions.
    tled<="111"; cnt:=cnt+1;
    init_we <= '0';  aerin_lfsr <= '1';   aerin_we <= '0';
    write_conv <= '0'; waitconv<='0';

 if(cy(6) = '1') then
imi:=conv_integer(aer_in_data(13 downto 8))+ conv_integer(cy(5 downto 0));
 else
imi:=conv_integer(aer_in_data(13 downto 8))- conv_integer(cy(5 downto 0));
 end if;

 if(cx(6) = '1') then
imj:=conv_integer(aer_in_data(5 downto 0))+ conv_integer(cx(5 downto 0));
 else
imj:=conv_integer(aer_in_data(5 downto 0))-conv_integer(cx(5 downto 0));
 end if;

 if (imi >= NK/2) and (imi < 64-NK/2) and (imj>=NK/2) and (imj<64-NK/2) then
   caddress <= conv_std_logic_vector((imi-NK/2),6) &
    conv_std_logic_vector(imj-NK/2,6);
 kaddress <= (others=>'0');
 pj:=0; pi:=0; kj:=0; ki:=0; nj:=NK-1; ni:=NK-1;
 tled<="001";
         elsif (imi < NK/2) and (imj>=NK/2) and (imj<64-NK/2) then
    caddress <= "000000" & conv_std_logic_vector(imj-NK/2,6);
 kaddress <= conv_std_logic_vector(add_ker(NK/2-imi),7);
 pj:=0; pi:=NK/2-imi;  kj:=0;
 ki:=NK/2-imi; nj:=NK-1; ni:=NK-1;
 tled<="010";
         elsif (imi>=64-NK/2) and (imj>=NK/2) and (imj<64-NK/2) then
    caddress <= conv_std_logic_vector((imi-NK/2),6)
      & conv_std_logic_vector(imj-NK/2,6);
 kaddress <= (others=>'0');
 pj := 0; pi := 0; kj := 0; ki := 0;
 nj := NK-1; ni := NK-1-(imi-(63-NK/2));
 tled<="011";
         elsif (imi >= NK/2) and (imi < 64-NK/2) and (imj<NK/2) then
    caddress <= conv_std_logic_vector((imi-NK/2),6) & "000000";
 kaddress <= conv_std_logic_vector(NK/2-imj,7);
 pj:=NK/2-imj; pi := 0; kj:=NK/2-imj; ki := 0;
 nj := NK-1; ni := NK-1;
 tled<="100";
         elsif (imi >= NK/2) and (imi < 64-NK/2) and (imj>=64-NK/2) then
    caddress <= conv_std_logic_vector((imi-NK/2),6)
     & conv_std_logic_vector(imj-NK/2,6);
 kaddress <= (others=>'0');
 pj := 0; pi := 0; kj := 0; ki := 0;
 nj := NK-1-(imj-(63-NK/2)); ni := NK-1;
 tled<="101";
         elsif (imi < NK/2) and (imj < NK/2) then
    caddress <= (others=>'0');
 kaddress <= conv_std_logic_vector(add_ker(NK/2-imi)+(NK/2-imj),7);
 pj:=NK/2-imj+1;pi:=NK/2-imi; kj:=NK/2-imj+1;ki:=NK/2-imi;
 nj := NK-1; ni := NK-1;
 tled<="110";
         elsif (imi>=64-NK/2) and (imj>=64-NK/2) then
    caddress <= conv_std_logic_vector((imi-NK/2),6)
     & conv_std_logic_vector(imj-NK/2,6);
 kaddress <= (others=>'0');
 pj := 0; pi := 0; kj := 0; ki := 0;
 nj := NK-1-(imj-(63-NK/2)); ni := NK-1-(imi-(63-NK/2));
 tled<="000";
         elsif (imj>=64-NK/2) and (imi<NK/2) then
    caddress <= "000000" & conv_std_logic_vector(imj-NK/2,6);
 kaddress <= conv_std_logic_vector(add_ker(NK/2-imi),7);
 pj := 0; pi := NK/2-imi;kj := 0; ki := NK/2-imi;
 nj := NK-1-(imj-(63-NK/2)); ni := NK-1;
 tled<="000";
          elsif (imi>=64-NK/2) and (imj<NK/2) then
    caddress <= conv_std_logic_vector((imi-NK/2),6) & "000000";
 kaddress <= conv_std_logic_vector(NK/2-imj,7);
 pj:=NK/2-imj; pi := 0; kj := NK/2-imj; ki := 0;
 nj := NK-1; ni := NK-1-(imi-(63-NK/2));
 tled<="111";
          end if;
        elsif saer_in_req_l='0' and spi_valid='0'
          and cnt=2 and end_ker='0' then
    init_we <= '0'; write_conv <= '1'; aerin_lfsr <= '1';
    aerin_we <= '0'; waitconv <= '0';
 cnt:=cnt+2; led <= "000001";
 if conv_integer(kerdata) < 128 then
    if neuron>=fire_level then
 evento <= caddress; emite <= '1'; cdata <= thresold;
 else cdata <= kerdata + ramout;
 end if;
 else
    suma := 256 - conv_integer(kerdata);
 if suma >= conv_integer(ramout) then cdata <= thresold;
 else cdata <= conv_std_logic_vector(conv_integer(ramout) - suma, 8);
 end if;
 end if;
       elsif saer_in_req_l='0' and spi_valid='0' and (cnt mod 2)=1
        and end_ker='0' and waitconv='0' then
    init_we <= '0';  aerin_lfsr <= '1'; aerin_we <= '0';
 led <= "000010";
 if conv_integer(kerdata) < 128 then
    if neuron>=fire_level then
 evento <= caddress; cdata <= thresold;
 if(fifo_full = '0') then
emite <= '1'; cnt:=cnt+1; write_conv <= '1';
else
write_conv <= '0';
end if;
 else cdata <= kerdata + ramout; write_conv <= '1'; cnt:=cnt+1;
 end if;
 else
 cnt:=cnt+1; write_conv <= '1';
    suma := 256 - conv_integer(kerdata);
 if suma >= conv_integer(ramout) then cdata <= thresold;
 else cdata <= conv_std_logic_vector(conv_integer(ramout)-suma,8);
 end if;
 end if;
       elsif saer_in_req_l='0' and spi_valid='0' and (cnt mod 2)=0
        and end_ker='0' and waitconv='0' then
    init_we <= '0'; aerin_lfsr <= '1'; aerin_we <= '0';
    write_conv <= '0'; waitconv <= '1';
 led <= "000011";
 if kj<nj then
   kj:=kj+1;
caddressx := conv_integer (caddress(5 downto 0));
caddress <= caddress(11 downto 6)
 & conv_std_logic_vector(caddressx+1,6);
kaddressv := conv_integer(kaddress);
kaddress <= conv_std_logic_vector(kaddressv +1,7);
end_ker<='0';
if (kj-1=nj) and (ki=ni) then end_ker<='1';
end if;
 elsif ki<ni then
   kj:=pj; ki:=ki+1;
caddressx := conv_integer (caddress(5 downto 0));
caddressy := conv_integer (caddress(11 downto 6));
caddress <= conv_std_logic_vector(caddressy+1,6)
& conv_std_logic_vector(caddressx+pj-nj,6);
```

```
kaddressv := conv_integer(kaddress);
kaddress <= conv_std_logic_vector(kaddressv+pj+NK-nj,7);
end_ker<='0';
 else end_ker <= '1';
 end if;
       elsif saer_in_req_l='0' and spi_valid='0' and (cnt mod 2)=0
        and end_ker='0' and waitconv='1' then
    init_we <= '0'; aerin_lfsr <= '1'; aerin_we <= '0';
    write_conv <= '1'; waitconv <= '0';
 cnt:=cnt+1; led <= "000100";
 elsif saer_in_req_l='0' and end_ker='1'  and spi_valid='0' then
 -- The input AER is already processed and a ACK is sent back
    init_we <= '0'; aerin_lfsr <= '0'; aerin_we <= '0';
    write_conv <= '0'; waitconv <= '0';
 aer_in_ack_l <= '0'; led <= "000101";
 elsif saer_in_req_l='1' and end_ker='1'  and spi_valid='0' then
 -- If the input REQ is deasserted, the ACK is also deasserted
 -- and a new input event can be processed
    init_we <= '0'; aerin_lfsr <= '0'; aerin_we <= '0';
    write_conv <= '0'; waitconv <= '0';
    aer_in_ack_l <= '1';end_ker<='0'; cnt:=0;
       else
      init_we <= '0'; aerin_lfsr <= '0'; aerin_we <= '0';
      write_conv <= '0'; waitconv <='0';
    aer_in_ack_l <= '1';
       end if;
end if;
     end if;
end process;

process(kerdata,ramout)
variable ad: integer range 0 to 511;
begin
ad := conv_integer(kerdata)+conv_integer(ramout);
neuron <= conv_std_logic_vector(ad,9);
end process;
```

10.3.5 Auxiliary blocks connection

We consider auxiliary blocks the kernel and neuron state RAMs, the output event FIFO and the configuration blocks. *SPI_SLAVE* receives the input SPI signals, decode them and present the information in a parallel way to be stores in the configuration registers. *B_ram_kernel* and *B_ram_ConvMat* are the convolution RAMs and *B_fifo_out* is the output FIFO. Process *B_sync* synchronizes the asynchronous input AER signals with the convolution module local clocks and *B_ifz_cfg_conv_ram* initializes the auxiliary memory after reset. *Crouter* generates the configuration signals for an event router which can be connected to the convolution module.

```
   debug <= "0000" & cdata;
   SPI_SLAVE : SPI_SLAVE port map
         ( CLK      => clk,
           RST      => rst,
           STR      => STR,
           NSS      => NSS,
           SCLK     => SCLK,
           MOSI     => MOSI,
           MISO     => MISO,
           WR       => pre_spi_valid,
           ADDRESS  => spi_address,
           DATA_OUT => pre_spi_data);
   B_ram_kernel: spblockram generic map (128,7,8)
PORT MAP(
   clk => clk,
   we => ker_we,
   a => keraddress,
   di => ker_data_in,
   do => kerdata_prev);

ca2Aux <= "00000001";
--choose CA2 or not depending of input event sign
process(aer_in_data(7),kerdata_prev,ca2Aux)
begin
if(aer_in_data(7) = '1') then
kerdata <= kerdata_prev;
else
kerdata <= not(kerdata_prev) + ca2Aux;
end if;
end process;

B_ram_ConvMat: conv_matrix
GENERIC MAP(4096,12,8)
PORT MAP(
address => caddress,
data => cdata,
ramout => ramout,
wrram => wrram,
enable => enable,
rst_n => rst_l,
CLK => CLK
);

B_sm_out: aer_out PORT MAP(
REQ_N => aer_out_req_l,
ACK_N => aer_out_ack_l,
Data_AER => kk,
Data_Fifo => fifo_data_out,
Enable_Fifo => not_fifo_empty,
RD_Fifo => Fifo_rd,
CLK => CLK,
RST_N => RST_L
);

   aer_out_data <= kk;

B_fifo_out: ramfifo
GENERIC MAP(16,4,16)
PORT MAP(
clk => clk,
wr => fifo_wr,
rd => Fifo_rd,
rst_n => RST_L,
empty => Fifo_empty,
full => Fifo_full,
data_in => fifo_Data_In,
data_out => fifo_Data_Out,
mem_used => mem_used
);

   B_sync: process(RST_L, CLK)
-- Synchronization process. All control input signals must be synchronized
-- respect to our internal clock
-- using a double flip-flop process.
   begin
if (RST_L = '0') then
iaer_in_req_l <= '0';  saer_in_req_l <= '0';
iaer_out_ack_l <= '0'; saer_out_ack_l <= '0';
spd <= '1'; pd0<='1';
latched_spd <= '0'; spi_valid <= '0';
```

```
spi_data <= (others =>'0');
elsif(CLK'event and CLK = '1') then
iaer_in_req_l <= aer_in_req_l;  saer_in_req_l <= iaer_in_req_l;
iaer_out_ack_l <= aer_out_ack_l; saer_out_ack_l <= iaer_out_ack_l;
pd0 <= PD; spd <= pd0;
   latched_spd <= spd; spi_valid <= pre_spi_valid;
spi_data <= pre_spi_data;
end if;
end process;

B_ifz_cfg_conv_ram: process(RST_L, CLK)
-- This process manages the initialization of the kernel memory during
-- any initialization command, due to an initial reset, a soft reset
-- or a hard reset.
variable first,one: boolean;
begin
if (RST_L = '0') then
maddress <= (others => '0'); tparam <= 0; tker_we <= '0';
first:=TRUE; one:=FALSE;
endINker <= '0';
elsif (CLK'event and CLK = '1') then
if (init='1' or init_soft='1') and endINker='0' then
if maddress <= "1111111" then tker_we <= not tker_we;
else tker_we <= '0';
end if;
if tker_we = '1' and maddress <= "1111111" then
maddress <= maddress +1;
if(maddress = "1111111") then
endINker <= '1';
end if;
end if;

elsif softreset='0' or hardreset='0' then maddress<=(others => '0');
elsif (spi_valid = '1' and spd = '1' and maddress<="1111111") then
 maddress <= maddress + 1;
elsif spd='0' then tker_we<='0'; maddress <= (others => '0');
else tker_we<= '0';
end if;
end if;
end process;

ker_data_in <= x"00" when (endINker='0') else
spi_data when (spi_valid='1' and endINker='1') else (others=>'0');
fifo_wr <= (emite and write_conv) and not fifo_full;

ker_we <= '1'
when (spd='1' and spi_valid = '1' and maddress <= "1111111"
and ProgKernel = '1')
else  tker_we when (init='1' or init_soft='1') else '0';
keraddress <= maddress
when (spi_valid='1' or init='1' or init_soft='1')
else kaddress;


not_fifo_empty <= not fifo_empty;

fifo_data_in(5 downto 0) <= evento(5 downto 0);
fifo_data_in(7 downto 6) <= (others => '0');
fifo_data_in(13 downto 8) <= evento(11 downto 6);
fifo_data_in(15 downto 14) <= (others => '0');

enable <= '0' when (fifo_full='1' or saer_in_req_l='0' or forget='1')
 else '1';
wrram <= init_we or write_conv;


--process to detect the end of the kernel programming;
process(ProgKernel,spd,spi_valid,maddress,NK,countA,countB)
```

```
begin
n_countA <= countA;
n_countB <= countB;
if(spd='1' and spi_valid = '1' and maddress <= "1111111") then
if(countA = conv_std_logic_vector(NK,4)) then
n_countB <= countB + 1;
n_countA <= "0001";
else
n_countA <= countA + 1;
end if;
end if;

if(countB = conv_std_logic_vector(NK,4)
and countA = conv_std_logic_vector(NK,4)) then
n_ProgKernel <= '0';
else
n_ProgKernel <= ProgKernel;
end if;
end process;

process(RST_L,CLK)
begin
if(RST_L = '0') then
ProgKernel <= '1';
countA <= "0000";
countB <= "0001";
elsif(CLK'event and CLK = '1') then
ProgKernel <= n_ProgKernel;
countA <= n_countA;
countB <= n_countB;
end if;
end process;


Crouter: confROUTER PORT MAP (
          clk => CLK,
          reset => RST_L,
          SPIdata => spi_data,
          SPIvalid => spi_valid,
          PD => ProgKernel,
          xADD => xADD,
          yADD => yADD,
          RTdata => RTdata,
          RTadd => RTadd,
          reqW => reqW,
          ackW => ackW
        );
end Behavioral;
```

10.3.6 Auxiliary blocks VHDL description

This Subsection lists the internal description of the auxiliary blocks needed for the convolution. The way these blocks are connected in the convolution module top level have been presented in the previous Subsection.

- **SPI configuration block:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;
```

```
entity SPI_SLAVE is
    Port ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           STR : out  STD_LOGIC;
           NSS : in  STD_LOGIC;
           SCLK : in  STD_LOGIC;
           MOSI : in  STD_LOGIC;
           MISO : out  STD_LOGIC;
  WR: out STD_LOGIC;
  ADDRESS: out STD_LOGIC_VECTOR(7 downto 0);
  DATA_OUT: out STD_LOGIC_VECTOR(7 downto 0));
end SPI_SLAVE;


architecture Behavioral of SPI_SLAVE is

component latch3 is
 Port ( CLK : in  STD_LOGIC;
  RST : in  STD_LOGIC;
  DATA_IN : in  STD_LOGIC_VECTOR (2 downto 0);
  DATA_OUT : out  STD_LOGIC_VECTOR (2 downto 0));
end component;
type STATE_TYPE is (IDLE,WAIT_FALL_EDGE,NEW_BIT,WAIT_NEW_BIT,NEW_RECIVED_WORD);
signal CS, NS: STATE_TYPE;
signal int_data: STD_LOGIC_VECTOR (2 downto 0);
signal RECIVED_DATA: STD_LOGIC_VECTOR (15 downto 0):=(others=>'0');
signal SCLK_int,MOSI_int,NSS_int,MISO_int: STD_LOGIC:='0';
signal test_out:std_logic_vector(15 downto 0):=x"5AA5";
signal i: integer range 0 to 16;
signal twr: std_logic;
begin

DATA_OUT<=RECIVED_DATA(7 DOWNTO 0) when twr='1' else
          (others=>'Z');
ADDRESS<=RECIVED_DATA(15 DOWNTO 8);
STR<='0';
MISO<=MISO_int;
WR<=twr;
latch0: latch3
 Port map( CLK =>clk,
  RST =>rst,
  DATA_IN(0)=>sclk,
  DATA_IN(1)=>mosi,
  DATA_IN(2)=>nss,
  DATA_OUT =>int_data);
latch1: latch3
 Port map( CLK =>clk,
  RST =>rst,
  DATA_IN=>int_data,
  DATA_OUT(0)=>sclk_int,
  DATA_OUT(1)=>mosi_int,
  DATA_OUT(2)=>nss_int);

process(CS,NS,SCLK_INT,MOSI_INT,i,recived_data,test_out,MISO_int)
begin
case CS is
when IDLE=>
tWR<='0';
MISO_int<='0';
if(sclk_int='0') then
NS<=IDLE;
  else
NS<=WAIT_FALL_EDGE;
end if;
when WAIT_FALL_EDGE=>
tWR<='0';
if(i<16) then
MISO_int<=test_out(15-i);
```

```
else
MISO_int<='0';
end if;
if(sclk_int='1') then
NS<=WAIT_FALL_EDGE;
else
NS<=NEW_BIT;
end if;
when NEW_BIT=>
tWR<='0';
NS<=WAIT_NEW_BIT;
if(i<16) then
MISO_int<=test_out(15-i);
else
MISO_int<='0';
end if;
when WAIT_NEW_BIT=>
tWR<='0';
NS<=WAIT_NEW_BIT;
if(sclk_int='1') then
  if (i=16) then
NS<=NEW_RECIVED_WORD;
MISO_int<='0';
  else
MISO_int<=test_out(15-i);
NS<=WAIT_FALL_EDGE;
  end if;
               end if;
when NEW_RECIVED_WORD=>
tWR<='1';
MISO_int<=MISO_int;
NS<=IDLE;
when others=>
tWR<='0';
MISO_int<=MISO_int;
NS<=IDLE;
end case;
end process;

process(CLK,RST,NSS_int,CS,NS)
begin
if(rst='1' OR (NSS_int='1' and  NS/=NEW_RECIVED_WORD) ) then
CS<=IDLE;
elsif(clk='1' and clk'event) then
CS<=NS;
if(CS=NEW_BIT) then
RECIVED_DATA<=RECIVED_DATA(14 downto 0) & MOSI_INT; --Leemos el dato.
i<=i+1;
elsif(CS=IDLE) then
i<=0;
end if;
end if;
end process;
end Behavioral;
```

- **Kernel weights RAM:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity spblockram is
 generic (RAM_Size: in integer:= 4096;
  RAM_Addr_Size: in integer:=12;
RAM_Data_Size: in integer:=8);
 port (clk : in std_logic;
  we  : in std_logic;
  a   : in std_logic_vector(RAM_Addr_Size-1 downto 0);
  di  : in std_logic_vector(RAM_Data_Size-1 downto 0);
  do  : out std_logic_vector(RAM_Data_Size-1 downto 0));
 end spblockram;

 architecture syn of spblockram is

 type ram_type is array(RAM_Size-1 downto 0) of std_logic_vector(RAM_Data_Size-1 downto 0);
 signal RAM : ram_type;
 signal read_a : std_logic_vector(RAM_Addr_Size-1 downto 0);

 begin
 process (clk)
 begin
  if (clk'event and clk = '1') then
  if (we = '1') then
  RAM(conv_integer(a)) <= di;
  end if;
  read_a <= a;
  end if;
 end process;

 do <= RAM(conv_integer(read_a));

 end syn;
```

- **Kernel weights RAM:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED."+";
use IEEE.STD_LOGIC_UNSIGNED."-";

entity conv_matrix is
generic(RAM_Size: in integer := 4096;
RAM_Addr_Size: in integer := 12;
RAM_Data_Size: in integer := 8);
    Port ( address : in std_logic_vector(RAM_Addr_Size -1 downto 0);
           data : in std_logic_vector(RAM_Data_Size-1 downto 0);
           ramout: out std_logic_vector(RAM_Data_Size-1 downto 0);
  wrram: in std_logic;
  enable: in std_logic;
  RST_N: in std_logic;
  CLK: in std_logic
  );
end conv_matrix;

architecture Behavioral of conv_matrix is

 component spblockram
 generic (RAM_Size: in integer:= RAM_Size;
    RAM_Addr_Size: in integer:=RAM_Addr_Size;
    RAM_Data_Size: in integer:=RAM_Data_Size);
 port (clk : in std_logic;
  we  : in std_logic;
  a   : in std_logic_vector(RAM_Addr_Size-1 downto 0);
  di  : in std_logic_vector(RAM_Data_Size-1 downto 0);
  do  : out std_logic_vector(RAM_Data_Size-1 downto 0));
 end component;

signal image_we: std_logic;
signal image_in:  std_logic_vector(RAM_Data_Size-1 downto 0);
signal image_out, imout_l: std_logic_vector(RAM_Data_Size-1 downto 0);
signal gray_lfsr: std_logic_vector(RAM_Data_Size-1 downto 0);
signal maddress: std_logic_vector(RAM_Addr_Size-1 downto 0);
signal iaddress: std_logic_vector(RAM_Addr_Size-1 downto 0);
signal levento: std_logic_vector(RAM_Addr_Size-1 downto 0);
signal LFSR: std_logic_vector(RAM_Data_Size+RAM_Addr_Size -1 downto 0);
signal en_lfsr, temite: std_logic;

begin

imagen: spblockram PORT MAP(
clk => clk,
we => image_we,
a => maddress,
di => image_in,
do => image_out);

ramout <= image_out;

ram_access: process(rst_n, data, address, enable,wrram)
begin
if (rst_n = '0') then
image_in <= (others =>'0');
image_we <= '0';
maddress <= (others =>'0');
else
image_in <= data;
image_we <= wrram;
maddress <= address;
end if;
end process;
end Behavioral;
```

- **Output event FIFO:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity aer_out is
    Port (REQ_N : out std_logic;
          ACK_N : in std_logic;
          Data_AER : out std_logic_vector(15 downto 0);
          Data_Fifo : in std_logic_vector(15 downto 0);
          Enable_Fifo : in std_logic;
 RD_Fifo: out std_logic;
 CLK : in std_logic;
          RST_N : in std_logic);
end aer_out;


architecture Behavioral of aer_out is

type STATE_TYPE is (IDLE, SendDATA);
signal CS, NS: STATE_TYPE;
signal cEvents,n_cEvents : std_logic_vector(10 downto 0);
attribute s: string;
attribute s of cEvents : signal is "yes";
begin

  SYNC_PROC: process (CLK, RST_N)
begin
       if (RST_N='0') then
              CS <= IDLE;
  cEvents <= (others => '0');
       elsif (CLK'event and CLK = '1') then
            CS <= NS;
cEvents <= n_cEvents;
       end if;
    end process;

    COMB_PROC: process (CS, Enable_Fifo, ack_n,cEvents)
    begin
 n_cEvents <= cEvents;
    case CS is

when IDLE =>
REQ_N <= '1';
         if (Enable_Fifo ='1' and ACK_N='1') then
NS <= SendDATA;
n_cEvents <= cEvents + 1;
else
         NS <= IDLE;
         end if;
RD_Fifo <= '0';

when SendDATA =>
REQ_N <='0';
if (ACK_N='0') then
NS <= IDLE;
RD_Fifo <= '1';
else
NS <= SendDATA;
RD_Fifo <= '0';
end if;
    end case;
end process;

Data_AER <= Data_Fifo;

end Behavioral;
```

- **Router configuration block:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.all;
USE IEEE.std_logic_unsigned.all;

entity confROUTER is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           SPIdata : in  STD_LOGIC_VECTOR (7 downto 0);
           SPIvalid : in  STD_LOGIC;
           PD : in  STD_LOGIC;
           xADD : out  STD_LOGIC_VECTOR (3 downto 0);
           yADD : out  STD_LOGIC_VECTOR (3 downto 0);
           regConf : out std_logic_vector(3 downto 0);
  writeRAM : out STD_LOGIC;
  ackWrite : in STD_LOGIC;
  addWrite : out STD_LOGIC_VECTOR(7 downto 0);
  dataWrite : out  STD_LOGIC_VECTOR (5 downto 0));
end confROUTER;

architecture Behavioral of confROUTER is
signal ADD,nADD : std_logic_vector(7 downto 0);
signal count,n_count : std_logic_vector(2 downto 0);
signal Byte,nByte : std_logic_vector(15 downto 0);
signal enREAD : std_logic;
signal nRegRout,RegRout : std_logic_vector(3 downto 0);
signal latSPI,n_latSPI : std_logic_vector(7 downto 0);
TYPE estate is (IDLE,READ_DATA,HANDSHAKE1,HANDSHAKE2);
SIGNAL current_state,next_state: estate;
begin

xADD <= ADD(3 downto 0);
yADD <= ADD(7 downto 4);
```

```
regConf <= RegRout;
addWrite <= Byte(7 downto 0);
dataWrite <= Byte(13 downto 8);

process(count,ADD,SPIdata,Byte,enREAD,latSPI,RegRout)
begin
nADD <= ADD;
nByte <= Byte;
nRegRout <= RegRout;

if(enREAD = '1') then
case count is
when "001" =>
nADD <= latSPI;
when "010" =>
nRegRout <= latSPI(3 downto 0);
when "011" =>
nByte(7 downto 0) <= latSPI;
when others =>
   nByte(15 downto 8) <= latSPI;
end case;
end if;

end process;

process(current_state,PD,SPIvalid,ackWrite,count,SPIdata,latSPI)
begin
writeRAM <= '0';
next_state <= current_state;
n_count <= count;
enREAD <= '0';
n_latSPI <= latSPI;

case current_state is
when IDLE =>
if(SPIvalid = '1' and PD= '0') then
next_state <= READ_DATA;
n_count <= count + 1;
n_latSPI <= SPIdata;
end if;
when READ_DATA =>
enREAD <= '1';
if(count = "100") then
next_state <= HANDSHAKE1;
else
next_state <= IDLE;
end if;
when HANDSHAKE1 =>
writeRAM <= '1';
if(ackWrite = '1') then
next_state <= HANDSHAKE2;
end if;
when HANDSHAKE2 =>
if(ackWrite = '0') then
next_state <= IDLE;
n_count <= "010";
end if;
when others =>
next_state <= IDLE;
end case;
end process;

process(clk,reset)
begin
if(reset = '0') then
current_state <= IDLE;
ADD <= (others => '0');
Byte <= (others  => '0');
```

```
count <= (others => '0');
latSPI <= (others => '0');
RegRout <= (others => '0');
elsif(clk = '1' and clk'event) then
current_state <= next_state;
ADD <= nADD;
Byte <= nByte;
count <= n_count;
latSPI <= n_latSPI;
RegRout <= nRegRout;
end if;
end process;

end Behavioral;
```

11 Appendix II: Matlab functions for NoC VHDL generation

11.1 Introduction

This Appendix contains Matlab functions[1] which write the VHDL code for a configurable NoC arranged in a 2D mesh. This code can be directly included in the FPGA project to implement the network with the input splitter, configuration resources and output channels multiplexing, as shown in Fig. 3.11 of Chapter 3. After executing the main function, the user will obtain a top VHDL block which connects the processing resources (convolution modules in our example) with the auxiliary blocks. Besides this file, the program creates the VHDL description of the input splitter and the configuration processor.

The user must provide the following input information:

- *netlist*: an AERST compatible netlist[2] describing the network connectivity which will be mapped into the VHDL description file.
- *topVHDL*: name of the VHDL top module which describes the NoC and auxiliary blocks connectivity. Adding a ".vhd" extension is recommended to handle the file with FPGA synthesis and implementation tools.
- *sizex, sizey*: number of rows and columns of the 2D mesh.
- *mode*: destination-driven and source-driven networks are built if this option is set to "dest" or "sour", respectively.
- *multiplicity*: the input splitter sends N output events for each input event. N is the splitter multiplicity.

[1]This source code is available upon request to bernabe@imse-cnm.csic.es

[2]AERST(AER Simulation Tool) is an event-driven simulator for AER systems, in which each AER module is user defined and behaviorally described [130]

Once these files have been generated, they have to be grouped with the VHDL files listed in the previous Appendix and the event-based routers to form the NoC system.

11.2 Matlab code

The user has to execute the main function with the parameters described previously and will obtain the VHDL files in the Matlab working directory.

11.2.1 Main function

```
function writeVHDL(netlist,topVHDL,sizex,sizey,multiplicity,mode)
%% VHDL instantation templates for input splitter
nameSplit='Splitter : INPUTsplitter';
portmapSplit='Port Map(';
clkLine='clk => clkHS,';
resetLine='reset => reset,';
inAERLine='AERin => AERin,';
reqInLine='reqIN => reqIN,';
ackInLine='ackIN => ackIN,';
AERoutLine='AERocut';
AssignLine='=>';
reqOUTLine='reqOUT';
ackOUTLine='ackOUT';
ADDline='ADD';

%%VHDL instantation templates for convolvers
AERLine='AER';
reqLine='req';
ackLine='ack';

%%open netlist file generated for the simulator
fidNetlist=fopen(netlist,'r');
fidOUT=fopen(topVHDL,'w');
if(fidNetlist ~= -1 && fidOUT ~= -1)
    flag1=1;
    while(flag1 == 1)
        NetlistLine=fgets(fidNetlist);
        %%look for the splitter line and extracts channels connected to it
        if(length(NetlistLine) > 13)
            if(strcmp(NetlistLine(1:13),'splitterRoute') == 1)
                flag1=0;
                count=0;
                flag2=1;
                k=14;
                m=1;
                while(flag2 == 1)
                    if(strcmp('{',NetlistLine(k))==1)
                        count=count + 1;
                        if(count == 3)
                            channelsSplit(m)=sscanf(NetlistLine((k+1):end),'%d');
                            m=m+1;
                        end
                    elseif(strcmp(',',NetlistLine(k))==1)
                            channelsSplit(m)=sscanf(NetlistLine((k+1):end),'%d');
                            m=m+1;
                    elseif(strcmp('}',NetlistLine(k))==1)
                        count=count+1;
                        if(count == 4)
                            flag2 = 0;
                        end
                    end
                    k=k+1;
                end
            end
        end
    end


    NetlistLine=fgets(fidNetlist);
    Nelem=sizex * sizey;
    indChip=1;
    ChannelsIN=zeros(Nelem,4);
    ChannelsOUT=zeros(Nelem,4);
    %%now analyze the convolver lines to extract connectivity
    while(indChip <= Nelem)
        if(length(NetlistLine) >= 18)
            if(strcmp(NetlistLine(1:18),'convolutionRoute {') == 1)
                %%analyzeLine to extract input and output channels
                A=sscanf(NetlistLine,'convolutionRoute {%d,%d,%d,%d} {%d,%d,%d,%d}');
                ChannelsIN(indChip,:)=A(1:4);
                ChannelsOUT(indChip,:)=A(5:8);
                indChip = indChip + 1;
            end
        end
        NetlistLine=fgets(fidNetlist);
    end


    %%Parameters needed to write the VHDL: xsize,ysize,number of replica
    %%events for each splitter channel,number of total channels
    nChannels=max(max([ChannelsIN ChannelsOUT]));

    %%extract channels in the borders to route them to the multiplexer
    SelectedChannels=zeros(2*sizex+sizey);
    TiedPorts=zeros(2*sizex+sizey);
    ind=1;
    chip=1;
    %run all the columms
    for k=1:sizex
        %%take UP-channel from NoC border
        SelectedChannels(ind) = ChannelsOUT(chip,1);
        %%take DOWN-channel from NoC border
        SelectedChannels(ind+1) = ChannelsOUT(chip+sizey-1,3);
        %%take UP-channels to tie-up
        TiedPorts(ind) = ChannelsIN(chip,1);
        %%take DOWN-channels to tie-up
        TiedPorts(ind+1) = ChannelsIN(chip+sizey-1,3);
        ind=ind+2;
        chip=chip+sizey;
    end
    chip=(sizex-1)*sizey+1; %%go to the last column
    %%run all the rows
    for k=1:sizey
        %%take RIGHT-channel from NoC border
        SelectedChannels(ind) = ChannelsOUT(chip,2);
        %%take RIGHT-channel to tie-up
        TiedPorts(ind) = ChannelsIN(chip,2);
        ind=ind+1;
        chip=chip+1;
    end

    [bitsNelem,bitsSplit,bitsOUT,bitsADD]=genTOPentity(fidOUT,sizex,sizey,multiplicity);

    writeSignals(fidOUT,nChannels,bitsADD,sizex,sizey);

    tiePorts(fidOUT,TiedPorts);

    writeClocking(fidOUT);

    instantiateSplitter(fidOUT,channelsSplit);
    for k=1:sizex*sizey
       writeConvolver(fidOUT,k,sizex*sizey,ChannelsIN(k,:),ChannelsOUT(k,:));
    end

    writeProg(fidOUT,sizey);

    writeOUTMUX(fidOUT,2*sizey+sizex,SelectedChannels);

    fprintf(fidOUT,'end Behavioral;\n');

    %%%%create auxiliar vhdl blocks needed for configuration,splitting and
    %%%%merging
    if(strcmp(mode,'source') == 1)
        writeProgNetworkSour(sizex,sizey,multiplicity);
    else
        writeProgNetwork(sizex,sizey,multiplicity);
    end
    writeINPUTsplitter(sizey,multiplicity);

    writeMUX(2*sizey+sizex);

    %close files
    fclose(fidNetlist);
    fclose(fidOUT);

else
    if(fidNetlist == -1)
        fprintf('Error opening the netlist file\n');
    elseif(fidOUT == -1)
        fprintf('Error opening the output file\n');
    else
        fprintf('Error opening files\n')
    end
end
```

11.2.2 Top entity generation

```
function [bits,bitsSplit,bitsOUT,bitsADD]=genTOPentity(fidOUT,xsize,ysize,Nsplit)
%%xsize: number of elements in X coordinate
%%ysize: number of elements in Y coordinate
%%Nsplit: maximum splitter output multiplicity. Maximum number of events
%%that can be generated for each input event

TotalSize=xsize * ysize;
flag=1;
bits=1;
while(flag == 1)
    if(2^bits < TotalSize)
        bits = bits + 1;
    else
        flag=0;
    end
end

bitsSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsSplit < Nsplit+1)
        bitsSplit = bitsSplit + 1;
    else
        flag=0;
    end
end

NoutChannels=xsize*2+ysize;
bitsOUT=1;
```

```
flag=1;
while(flag == 1)
    if(2^bitsOUT < NoutChannels)
       bitsOUT = bitsOUT + 1;
    else
        flag=0;
    end
end


fprintf(fidOUT,'library IEEE;\n');
fprintf(fidOUT,'use IEEE.STD_LOGIC_1164.ALL;\n\n');

fprintf(fidOUT,'library UNISIM;\n');
fprintf(fidOUT,'use UNISIM.VComponents.all;\n\n');

fprintf(fidOUT,'entity TOP is\n');
fprintf(fidOUT,'Port ( clkIN : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t reset : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERin : in STD_LOGIC_VECTOR(22 downto 0);\n');
fprintf(fidOUT,'\t reqIN : in STD_LOGIC;\n');
fprintf(fidOUT,'\t ackIN : out STD_LOGIC;\n');
fprintf(fidOUT,'\t sel : in STD_LOGIC_VECTOR(%d downto 0);\n',bitsOUT-1);
fprintf(fidOUT,'\t AERout : out STD_LOGIC_VECTOR(31 downto 0);\n');
fprintf(fidOUT,'\t reqOUT1 : out STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUT1 : in STD_LOGIC;\n');
fprintf(fidOUT,'\t Rx: IN STD_LOGIC;\n');
fprintf(fidOUT,'\t Tx: OUT STD_LOGIC;\n');
fprintf(fidOUT,'\t RTS: OUT STD_LOGIC);\n');
fprintf(fidOUT,'end TOP;\n\n\n');

fprintf(fidOUT,'architecture Behavioral of TOP is\n\n');

fprintf(fidOUT,'component TopConvRouter\n');
fprintf(fidOUT,'Port ( clk : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t reset : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERinL : in  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqINl : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackINl : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERinR : in  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqINr : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackINr : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERinU : in  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqINu : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackINu : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERinD : in  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqINd : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackINd : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERoutL : out  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqOUTl : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUTl : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERoutR : out  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqOUTr : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUTr : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERoutU : out  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqOUTu : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUTu : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERoutD : out  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqOUTd : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUTd : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t NSS : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t SCLK : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t MOSI : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t MISO : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t PD: in std_logic);\n');
fprintf(fidOUT,'end component;\n\n\n');
fprintf(fidOUT,'component progNETWORK\n');
fprintf(fidOUT,'Port (clk : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t reset : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t RX : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t TX : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t RTS : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t DATA : out STD_LOGIC;\n');
fprintf(fidOUT,'\t SCLK : out STD_LOGIC;\n');
fprintf(fidOUT,'\t PD : out STD_LOGIC;\n');
fprintf(fidOUT,'\t NSS : out STD_LOGIC_VECTOR(%d downto 0);\n',TotalSize-1);
for k=1:ysize
    if(k==ysize)
        fprintf(fidOUT,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0));\n'
                                                ,k,8*Nsplit+bitsSplit -1);
    else
        fprintf(fidOUT,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0);\n',
                                                k,8*Nsplit+bitsSplit -1);
    end
end
fprintf(fidOUT,'end component;\n\n\n');

bitsADD=8*Nsplit+bitsSplit;

fprintf(fidOUT,'component INPUTsplitter\n');
fprintf(fidOUT,'Port ( clk : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t reset : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t AERin : in STD_LOGIC_VECTOR(22 downto 0);\n');
fprintf(fidOUT,'\t reqIN : in  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackIN : out  STD_LOGIC;\n');
for k=1:ysize
    fprintf(fidOUT,'\t AERout%d : out  STD_LOGIC_VECTOR (31 downto 0);\n',k);
    fprintf(fidOUT,'\t reqOUT%d : out  STD_LOGIC;\n',k);
    fprintf(fidOUT,'\t ackOUT%d : in  STD_LOGIC;\n',k);
end
for k=1:ysize
    if(k==ysize)
        fprintf(fidOUT,'\t ADD%d : in  STD_LOGIC_VECTOR (%d downto 0));\n',
                                                    k,8*Nsplit+bitsSplit -1);
    else
        fprintf(fidOUT,'\t ADD%d : in  STD_LOGIC_VECTOR (%d downto 0);\n',
                                                    k,8*Nsplit+bitsSplit -1);
    end
end
fprintf(fidOUT,'end component;\n');



fprintf(fidOUT,'component OUTmux\n');
fprintf(fidOUT,'Port ( sel : in  STD_LOGIC_VECTOR (%d downto 0);\n',bitsOUT-1);
for k=1:NoutChannels
    fprintf(fidOUT,'\t AERin%d : in  STD_LOGIC_VECTOR (31 downto 0);\n',k);
    fprintf(fidOUT,'\t reqIN%d : in  STD_LOGIC;\n',k);
    fprintf(fidOUT,'\t ackIN%d : out  STD_LOGIC;\n',k);
end

fprintf(fidOUT,'\t AERout : out  STD_LOGIC_VECTOR (31 downto 0);\n');
fprintf(fidOUT,'\t reqOUT1 : out  STD_LOGIC;\n');
fprintf(fidOUT,'\t ackOUT1 : in  STD_LOGIC);\n');
fprintf(fidOUT,'end component;\n\n\n');


fprintf(fidOUT,'component clk_wiz_v1_6\n');
fprintf(fidOUT,'port\n');
fprintf(fidOUT,'(\n');
fprintf(fidOUT,'\t CLK_IN1            : in     std_logic;\n');
fprintf(fidOUT,'\t CLK_OUT1           : out    std_logic;\n');
fprintf(fidOUT,'\t RESET              : in     std_logic;\n');
fprintf(fidOUT,'\t LOCKED             : out    std_logic\n');
fprintf(fidOUT,'\t );\n');
fprintf(fidOUT,'end component;\n\n\n');
```

11.2.3 Signals defintion

```
%%function that writes all the signals needed for the internal architecture

function writeSignals(fidOUT,nChannels,bitsSplit,xsize,ysize)


ind=1;
while(ind <= nChannels)
    if(ind+3 <= nChannels)
            fprintf(fidOUT,'signal AER%d,AER%d,AER%d,AER%d : std_logic_vector
                                        (31 downto 0);\n',ind,ind+1,ind+2,ind+3);
            ind = ind + 4;
    elseif(ind+2 == nChannels)
            fprintf(fidOUT,'signal AER%d,AER%d,AER%d : std_logic_vector
                                        (31 downto 0);\n',ind,ind+1,ind+2);
            ind = ind + 3;
    elseif(ind+1 == nChannels)
            fprintf(fidOUT,'signal AER%d,AER%d : std_logic_vector(31 downto 0);\n',
                                                                    ind,ind+1);
            ind = ind + 2;
    elseif(ind == nChannels)
            fprintf(fidOUT,'signal AER%d : std_logic_vector(31 downto 0);\n',ind);
            ind = ind + 1;
    end
end


ind=1;
while(ind <= nChannels)
    if(ind+3 <= nChannels)
            fprintf(fidOUT,'signal req%d,req%d,req%d,req%d : std_logic;\n',
                                                        ind,ind+1,ind+2,ind+3);
            ind = ind + 4;
    elseif(ind+2 == nChannels)
            fprintf(fidOUT,'signal req%d,req%d,req%d : std_logic;\n',
                                                        ind,ind+1,ind+2);
            ind = ind + 3;
    elseif(ind+1 == nChannels)
            fprintf(fidOUT,'signal req%d,req%d : std_logic;\n',ind,ind+1);
            ind = ind + 2;
    elseif(ind == nChannels)
            fprintf(fidOUT,'signal req%d : std_logic;\n',ind);
            ind = ind + 1;
    end
end


ind=1;
while(ind <= nChannels)
    if(ind+3 <= nChannels)
            fprintf(fidOUT,'signal ack%d,ack%d,ack%d,ack%d : std_logic;\n',
                                                      ind,ind+1,ind+2,ind+3);
            ind = ind + 4;
    elseif(ind+2 == nChannels)
            fprintf(fidOUT,'signal ack%d,ack%d,ack%d : std_logic;\n',
                                                      ind,ind+1,ind+2);
            ind = ind + 3;
    elseif(ind+1 == nChannels)
            fprintf(fidOUT,'signal ack%d,ack%d : std_logic;\n',ind,ind+1);
            ind = ind + 2;
    elseif(ind == nChannels)
            fprintf(fidOUT,'signal ack%d : std_logic;\n',ind);
            ind = ind + 1;
    end
end

fprintf(fidOUT,'signal NSS : std_logic_vector(%d downto 0);\n',xsize*ysize-1);
fprintf(fidOUT,'signal SCLK,MOSI,PD : std_logic;\n');
for k=1:ysize
    fprintf(fidOUT,'signal ADD%d : std_logic_vector(%d downto 0);\n',k,bitsSplit-1);
end
fprintf(fidOUT,'signal clkHS,clk,lock : std_logic;\n');
fprintf(fidOUT,'begin\n\n\n');
```

11.2.4 Tie handshaking signals of unused channels

```
function tiePorts(fidOUT,TiedPorts)

for k=1:length(TiedPorts)
    fprintf(fidOUT,'AER%d <= (others => ''1'');\n',TiedPorts(k));
    fprintf(fidOUT,'req%d <= ''1'';\n',TiedPorts(k));
end

fprintf(fidOUT,'\n\n');
```

11.2.5 Clock generation block instantation

```
function writeClocking(fidOUT)



fprintf(fidOUT,'CM : clk_wiz_v1_6\n');
fprintf(fidOUT,'port Map\n');
fprintf(fidOUT,'(\n');
fprintf(fidOUT,'\t CLK_IN1 => clkIN,\n');
fprintf(fidOUT,'\t CLK_OUT1 => clkHS,\n');
fprintf(fidOUT,'\t RESET  => reset,\n');
fprintf(fidOUT,'\t LOCKED  => lock );\n\n\n');

fprintf(fidOUT,'process(clkHS,reset)\n');
fprintf(fidOUT,'begin\n');
fprintf(fidOUT,'if(reset = ''1'') then\n');
fprintf(fidOUT,'\t clk <= ''0'';\n');
fprintf(fidOUT,'elsif(clkHS = ''1'' and clkHS''event) then\n');
fprintf(fidOUT,'\t clk <= not clk;\n');
fprintf(fidOUT,'end if;\n');
fprintf(fidOUT,'end process;\n\n\n');
```

11.2.6 Input splitter instantiation

```
function instantiateSplitter(fidOUT,channelsSplit)

fprintf(fidOUT,'Splitter : INPUTsplitter\n');
fprintf(fidOUT,'Port Map(\n');
fprintf(fidOUT,'\t clk => clkHS,\n');
fprintf(fidOUT,'\t reset => reset,\n');
fprintf(fidOUT,'\t AERin => AERin,\n');
fprintf(fidOUT,'\t reqIN => reqIN,\n');
fprintf(fidOUT,'\t ackIN => ackIN,\n');

for k=1:length(channelsSplit)
    fprintf(fidOUT,'\t AERout%d => AER%d,\n',k,channelsSplit(k));
    fprintf(fidOUT,'\t reqOUT%d => req%d,\n',k,channelsSplit(k));
    fprintf(fidOUT,'\t ackOUT%d => ack%d,\n',k,channelsSplit(k));
end

for k=1:length(channelsSplit)
    if(k==length(channelsSplit))
```

```
        fprintf(fidOUT,'\t ADD%d => ADD%d);\n',k,k);
    else
        fprintf(fidOUT,'\t ADD%d => ADD%d,\n',k,k);
    end
end

fprintf(fidOUT,'\n\n');
```

11.2.7 Convolution module

```
function writeConvolver(fidOUT,N,Nconv,inChannels,outChannels)

fprintf(fidOUT,'conv%d : TopConvRouter\n',N);
fprintf(fidOUT,'Port Map ( clk => clkHS,\n');
fprintf(fidOUT,'\t reset => reset,\n');
fprintf(fidOUT,'\t AERinL => AER%d,\n',inChannels(4));
fprintf(fidOUT,'\t reqINl => req%d,\n',inChannels(4));
fprintf(fidOUT,'\t ackINl => ack%d,\n',inChannels(4));
fprintf(fidOUT,'\t AERinR => AER%d,\n',inChannels(2));
fprintf(fidOUT,'\t reqINr => req%d,\n',inChannels(2));
fprintf(fidOUT,'\t ackINr => ack%d,\n',inChannels(2));
fprintf(fidOUT,'\t AERinU => AER%d,\n',inChannels(1));
fprintf(fidOUT,'\t reqINu => req%d,\n',inChannels(1));
fprintf(fidOUT,'\t ackINu => ack%d,\n',inChannels(1));
fprintf(fidOUT,'\t AERinD => AER%d,\n',inChannels(3));
fprintf(fidOUT,'\t reqINd => req%d,\n',inChannels(3));
fprintf(fidOUT,'\t ackINd => ack%d,\n',inChannels(3));


fprintf(fidOUT,'\t AERoutL => AER%d,\n',outChannels(4));
fprintf(fidOUT,'\t reqOUTl => req%d,\n',outChannels(4));
fprintf(fidOUT,'\t ackOUTl => ack%d,\n',outChannels(4));
fprintf(fidOUT,'\t AERoutR => AER%d,\n',outChannels(2));
fprintf(fidOUT,'\t reqOUTr => req%d,\n',outChannels(2));
fprintf(fidOUT,'\t ackOUTr => ack%d,\n',outChannels(2));
fprintf(fidOUT,'\t AERoutD => AER%d,\n',outChannels(3));
fprintf(fidOUT,'\t reqOUTd => req%d,\n',outChannels(3));
fprintf(fidOUT,'\t ackOUTd => ack%d,\n',outChannels(3));
fprintf(fidOUT,'\t AERoutU => AER%d,\n',outChannels(1));
fprintf(fidOUT,'\t reqOUTu => req%d,\n',outChannels(1));
fprintf(fidOUT,'\t ackOUTu => ack%d,\n',outChannels(1));


fprintf(fidOUT,'\t NSS => NSS(%d),\n',N-1);
fprintf(fidOUT,'\t SCLK => SCLK,\n');
fprintf(fidOUT,'\t MOSI => MOSI,\n');
fprintf(fidOUT,'\t MISO => open,\n');
fprintf(fidOUT,'\t PD => PD);\n\n\n');
```

11.2.8 Configuration module instantiation

```
function writeProg(fidOUT,ysize)

fprintf(fidOUT,'progBlock : progNETWORK\n');
fprintf(fidOUT,'Port Map( clk => clkHS,\n');
fprintf(fidOUT,'\t reset => reset,\n');
fprintf(fidOUT,'\t RX => RX,\n');
fprintf(fidOUT,'\t TX => TX,\n');
fprintf(fidOUT,'\t RTS => RTS,\n');
fprintf(fidOUT,'\t DATA => MOSI,\n');
fprintf(fidOUT,'\t SCLK => SCLK,\n');
fprintf(fidOUT,'\t PD => PD,\n');
fprintf(fidOUT,'\t NSS => NSS,\n');
for k=1:ysize
    if(k==ysize)
        fprintf(fidOUT,'\t ADD%d => ADD%d);\n',k,k);
    else
        fprintf(fidOUT,'\t ADD%d => ADD%d,\n',k,k);
    end
end

fprintf(fidOUT,'\n\n');
```

11.2.9 Output multiplexer instantiation

```
function writeOUTMUX(fidOUT,nChannels,SelectedChannels)


fprintf(fidOUT,'MUX : OUTmux\n');
fprintf(fidOUT,'Port Map ( sel => sel,\n');

for k=1:nChannels
    fprintf(fidOUT,'\t AERin%d => AER%d,\n',k,SelectedChannels(k));
    fprintf(fidOUT,'\t reqIN%d => req%d,\n',k,SelectedChannels(k));
    fprintf(fidOUT,'\t ackIN%d => ack%d,\n',k,SelectedChannels(k));
end

fprintf(fidOUT,'\t AERout => AERout,\n');
fprintf(fidOUT,'\t reqOUT1 => reqOUT1,\n');
fprintf(fidOUT,'\t ackOUT1 => ackOUT1);\n');


fprintf(fidOUT,'\n\n');
```

11.2.10 Input splitter VHDL description

```
function writeINPUTsplitter(ysize,Mult)

bitsSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsSplit < Mult+1)
        bitsSplit = bitsSplit + 1;
    else
        flag=0;
    end
end

fid=fopen('INPUTsplitter.vhd','w');
if(fid == -1)
    fprintf('Error creating auxiliar INOUTsplitter file\n');
else
    fprintf(fid,'library IEEE;\n');
    fprintf(fid,'use IEEE.STD_LOGIC_1164.ALL;\n');
    fprintf(fid,'USE IEEE.std_logic_arith.all;\n');
    fprintf(fid,'USE IEEE.std_logic_unsigned.all;\n\n');

    fprintf(fid,'entity INPUTsplitter is\n');
    fprintf(fid,'Port ( clk : in STD_LOGIC;\n');
    fprintf(fid,'\t reset : in STD_LOGIC;\n');
    fprintf(fid,'\t AERin : in STD_LOGIC_VECTOR(22 downto 0);\n');
    fprintf(fid,'\t reqIN : in STD_LOGIC;\n');
    fprintf(fid,'\t ackIN : out STD_LOGIC;\n');

    for k=1:ysize
        fprintf(fid,'\t AERout%d : out STD_LOGIC_VECTOR (31 downto 0);\n',k);
        fprintf(fid,'\t reqOUT%d : out STD_LOGIC;\n',k);
        fprintf(fid,'\t ackOUT%d : in STD_LOGIC;\n',k);
    end

    for k=1:ysize
        if(k==ysize)
```

```
            fprintf(fid,'\t ADD%d : in  STD_LOGIC_VECTOR (%d downto 0));\n',
                                                        k,8*Mult+bitsSplit -1);
        else
            fprintf(fid,'\t ADD%d : in  STD_LOGIC_VECTOR (%d downto 0);\n',
                                                        k,8*Mult+bitsSplit -1);
        end
    end

    fprintf(fid,'end INPUTsplitter;\n\n');

    fprintf(fid,'architecture Behavioral of INPUTsplitter is\n');

    for k=1:ysize
        fprintf(fid,'TYPE estate%d is (IDLE%d,GEN_REQ_OUT%d,WAIT_HANDSHAKE%d);\n',
                                                                  k,k,k,k);
        fprintf(fid,'SIGNAL current_state%d,next_state%d: estate%d;\n',k,k,k);
    end
    for k=1:ysize
        fprintf(fid,'SIGNAL count%d,n_count%d : std_logic_vector(%d downto 0);\n',
                                                           k,k,bitsSplit -1);
    end
    for k=1:ysize
        fprintf(fid,'SIGNAL busy%d : std_logic;\n',k);
    end
    fprintf(fid,'signal reqINl,nreqINl : std_logic;\n');
    fprintf(fid,'begin\n\n\n');

    fprintf(fid,'process(');
    for k=1:ysize-1
         fprintf(fid,'busy%d,',k);
    end
    fprintf(fid,'busy%d)\n',ysize);
    fprintf(fid,'begin\n');
    fprintf(fid,'if(');
    for k=1:ysize-1
            fprintf(fid,'busy%d=''0'' and ',k);
    end
    fprintf(fid,'busy%d = ''0'') then\n',ysize);
    fprintf(fid,'\t ackIN <= ''1'';\n');
    fprintf(fid,'else\n');
    fprintf(fid,'\t ackIN <= ''0'';\n');
    fprintf(fid,'end if;\n');
    fprintf(fid,'end process;\n\n\n');

    for k=1:ysize
        fprintf(fid,'AERout%d(31) <= ''0'';\n',k);
    end

    for k=1:ysize
        fprintf(fid,'AERout%d(22 downto 0) <= AERin;\n',k);
    end

    fprintf(fid,'nreqINl <= reqIN;\n');

    for k=1:ysize
        fprintf(fid,'process(count%d,ADD%d)\n',k,k);
        fprintf(fid,'begin\n');
        fprintf(fid,'\t case count%d is\n',k);
        ind=8;
        for m=1:(Mult-1)
            fprintf(fid,'\t\t when "%s" =>\n',dec2bin(m,bitsSplit));
            fprintf(fid,'\t\t\t AERout%d(30 downto 23) <= ADD%d(%d downto %d);\n',
                                                            k,k,ind+7,ind);
            ind=ind+8;
        end
        fprintf(fid,'\t\t when others => \n');
        fprintf(fid,'\t\t\t AERout%d(30 downto 23) <= ADD%d(7 downto 0);\n',k,k);
        fprintf(fid,'\t end case;\n');
        fprintf(fid,'end process;\n\n');
    end


    for k=1:ysize
        fprintf(fid,'process(current_state%d,reqINl,ackOUT%d,ADD%d,count%d)\n'
                                                                ,k,k,k,k);
        fprintf(fid,'begin\n');
        fprintf(fid,'next_state%d <= current_state%d;\n',k,k);
        fprintf(fid,'reqOUT%d <= ''1'';\n',k);
        fprintf(fid,'n_count%d <= count%d;\n',k,k);
        fprintf(fid,'busy%d <= ''0'';\n\n',k);
        fprintf(fid,'case current_state%d is\n',k);
        fprintf(fid,'\t when IDLE%d =>\n',k);
        fprintf(fid,'\t\t if(reqINl = ''0'' and ADD%d(%d downto %d) /= "%s")
                          then\n',k,8*Mult+bitsSplit-1,8*Mult,dec2bin(0,bitsSplit));
                fprintf(fid,'\t\t\t next_state%d <= GEN_REQ_OUT%d;\n',k,k);
        fprintf(fid,'\t\t end if;\n');
                fprintf(fid,'\t when GEN_REQ_OUT%d =>\n',k);
        fprintf(fid,'\t\t busy%d <= ''1'';\n',k);
        fprintf(fid,'\t\t if(count%d < ADD%d(%d downto %d)) then\n',
                                                k,k,8*Mult+bitsSplit-1,8*Mult);
        fprintf(fid,'\t\t\t reqOUT%d <= ''0'';\n',k);
        fprintf(fid,'\t\t\t if(ackOUT%d = ''0'') then\n',k);
        fprintf(fid,'\t\t\t\t next_state%d <= WAIT_HANDSHAKE%d;\n',k,k);
        fprintf(fid,'\t\t\t end if;\n');
        fprintf(fid,'\t\t else\n');
        fprintf(fid,'\t\t\t if(reqINl = ''1'') then\n');
                fprintf(fid,'\t\t\t\t next_state%d <= IDLE%d;\n',k,k);
                fprintf(fid,'\t\t\t\t n_count%d <= (others => ''0'');\n',k);
        fprintf(fid,'\t\t\t end if;\n');
        fprintf(fid,'\t\t end if;\n');
                fprintf(fid,'\t when WAIT_HANDSHAKE%d =>\n',k);
        fprintf(fid,'\t busy%d <= ''1'';\n',k);
        fprintf(fid,'\t if(ackOUT%d = ''1'') then\n',k);
        fprintf(fid,'\t\t next_state%d <= GEN_REQ_OUT%d;\n',k,k);
        fprintf(fid,'\t\t n_count%d <= count%d + 1;\n',k,k);
        fprintf(fid,'\t end if;\n');
        fprintf(fid,'\t when others =>\n');
        fprintf(fid,'\t\t next_state%d <= IDLE%d;\n',k,k);
        fprintf(fid,'\t end case;\n');
        fprintf(fid,'end process;\n\n\n');
    end

    fprintf(fid,'process(clk,reset)\n');
    fprintf(fid,'begin\n');
    fprintf(fid,'\t if(reset = ''1'') then\n');
    for k=1:ysize
        fprintf(fid,'\t\t current_state%d <= IDLE%d;\n',k,k);
    end
    for k=1:ysize
        fprintf(fid,'\t\t count%d <= (others=> ''0'');\n',k);
    end
    fprintf(fid,'\t\t reqINl <= ''1'';\n');
        fprintf(fid,'\t elsif(clk = ''1'' and clk''event) then\n');
    for k=1:ysize
        fprintf(fid,'\t\t current_state%d <= next_state%d;\n',k,k);
    end
    for k=1:ysize
        fprintf(fid,'\t\t count%d <= n_count%d;\n',k,k);
    end
    fprintf(fid,'\t\t reqINl <= nreqINl;\n');
    fprintf(fid,'\t end if;\n');
    fprintf(fid,'end process;\n\n');

    fprintf(fid,'end Behavioral;\n');
    fclose(fid);
end
```

11.2.11 Output multiplexer VHDL description

```
function writeMUX(nChannels)

bitsSel=1;
flag=1;
while(flag == 1)
    if(2^bitsSel < nChannels)
        bitsSel = bitsSel + 1;
    else
        flag=0;
    end
end

fid=fopen('OUTmux.vhd','w');
if(fid == -1)
    fprintf('Error creating auxiliar OUTmux file\n');
else
    fprintf(fid,'library IEEE;\n');
    fprintf(fid,'use IEEE.STD_LOGIC_1164.ALL;\n');
    fprintf(fid,'entity OUTmux is\n');
    fprintf(fid,'Port ( sel : in STD_LOGIC_VECTOR (%d downto 0);\n',bitsSel-1);
    for k=1:nChannels
        fprintf(fid,'\t AERin%d : in STD_LOGIC_VECTOR (31 downto 0);\n',k);
        fprintf(fid,'\t reqIN%d : in STD_LOGIC;\n',k);
        fprintf(fid,'\t ackIN%d : out STD_LOGIC;\n',k);
    end
    fprintf(fid,'\t AERout : out STD_LOGIC_VECTOR (31 downto 0);\n');
    fprintf(fid,'\t reqOUT1 : out STD_LOGIC;\n');
    fprintf(fid,'\t ackOUT1 : in STD_LOGIC);\n');
    fprintf(fid,'end OUTmux;\n\n\n');

    fprintf(fid,'architecture Behavioral of OUTmux is\n');
    fprintf(fid,'begin\n\n');

    fprintf(fid,'process(sel,');
    for k=1:nChannels
        fprintf(fid,'AERin%d,reqIN%d,',k,k);
    end
    fprintf(fid,' ackOUT1)\n\n');

    fprintf(fid,'begin\n');
    for k=1:nChannels
        fprintf(fid,'ackIN%d <= reqIN%d;\n',k,k);
    end

    fprintf(fid,'\t case sel is\n');
    for k=1:nChannels-1
        fprintf(fid,'\t\t when "%s" =>\n',dec2bin(k-1,bitsSel));
                fprintf(fid,'\t\t\t AERout <= AERin%d;\n',k);
        fprintf(fid,'\t\t\t reqOUT1 <= reqIN%d;\n',k);
        fprintf(fid,'\t\t\t ackIN%d <= ackOUT1;\n',k);
    end
    fprintf(fid,'\t\t when others =>\n');
    fprintf(fid,'\t\t\t AERout <= AERin%d;\n',nChannels);
    fprintf(fid,'\t\t\t reqOUT1 <= reqIN%d;\n',nChannels);
    fprintf(fid,'\t\t\t ackIN%d <= ackOUT1;\n',nChannels);
        fprintf(fid,'\t end case;\n');
    fprintf(fid,'end process;\n\n');

    fprintf(fid,'end Behavioral;\n');

    fclose(fid);
end
```

11.2.12 Configuration block VHDL description for the destination-driven routing

```
function writeProgNetwork(xsize,ysize,Mult)

bitsSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsSplit < Mult+1)
        bitsSplit = bitsSplit + 1;
    else
        flag=0;
    end
end

BitsAdd=ysize*(Mult*8+bitsSplit);
oct=floor(BitsAdd/8);
if(rem(BitsAdd,8) ~= 0)
    oct=oct + 1;
end

bitsCSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsCSplit < oct)
        bitsCSplit = bitsCSplit + 1;
    else
        flag=0;
    end
end

bitsChip=1;
flag=1;
while(flag == 1)
    if(2^bitsChip< xsize*ysize)
        bitsChip = bitsChip + 1;
    else
        flag=0;
    end
end


fid=fopen('progNETWORK.vhd','w');
if(fid == -1)
    fprintf('Error creating auxiliar progNETWORK file\n');
else
    fprintf(fid,'library IEEE;\n');
    fprintf(fid,'use IEEE.STD_LOGIC_1164.ALL;\n');
    fprintf(fid,'USE IEEE.std_logic_arith.all;\n');
    fprintf(fid,'USE IEEE.std_logic_unsigned.all;\n\n\n');
    fprintf(fid,'entity progNETWORK is\n');
    fprintf(fid,'Port ( clk : in  STD_LOGIC;\n');
    fprintf(fid,'\t reset : in  STD_LOGIC;\n');
    fprintf(fid,'\t RX : in  STD_LOGIC;\n');
    fprintf(fid,'\t TX : out  STD_LOGIC;\n');
    fprintf(fid,'\t RTS : out  STD_LOGIC;\n');
    fprintf(fid,'\t DATA : out STD_LOGIC;\n');
    fprintf(fid,'\t SCLK : out STD_LOGIC;\n');
    fprintf(fid,'\t PD : out STD_LOGIC;\n');
    fprintf(fid,'\t NSS : out STD_LOGIC_VECTOR(%d downto 0);\n',xsize*ysize-1);
    for k=1:ysize
        if(k == ysize)
            fprintf(fid,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0));\n',k,Mult*8+bitsSplit-1);
        else
            fprintf(fid,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0);\n',k,Mult*8+bitsSplit-1);
        end
    end
```

```
fprintf(fid,'end progNETWORK;\n\n\n');

fprintf(fid,'architecture Behavioral of progNETWORK is\n\n\n');

fprintf(fid,'component RS232\n');
fprintf(fid,'Port(\n');
fprintf(fid,'\t clk: IN  STD_LOGIC;\n');
fprintf(fid,'\t reset: IN STD_LOGIC;\n');
fprintf(fid,'\t Rx: IN STD_LOGIC;\n');
fprintf(fid,'\t Tx: OUT STD_LOGIC;\n');
fprintf(fid,'\t RTS: OUT STD_LOGIC;\n');
fprintf(fid,'\t dataReady: OUT STD_LOGIC;\n');
fprintf(fid,'\t dataOUT: OUT STD_LOGIC_VECTOR(7 downto 0));\n');
fprintf(fid,'end component;\n\n');

fprintf(fid,'component SPIgen is\n');
fprintf(fid,'Port ( clk : in  STD_LOGIC;\n');
fprintf(fid,'\t reset : in  STD_LOGIC;\n');
fprintf(fid,'\t SCLK : out  STD_LOGIC;\n');
fprintf(fid,'\t MOSI : out  STD_LOGIC;\n');
fprintf(fid,'\t ready : out STD_LOGIC;\n');
fprintf(fid,'\t data : in  STD_LOGIC_VECTOR (7 downto 0);\n');
fprintf(fid,'\t enTX : in  STD_LOGIC);\n');
fprintf(fid,'end component;\n');

fprintf(fid,'TYPE estate is (IDLE,CONF_UNIT,WAIT_SEND_OCT,SEND_OCT,WAIT_TX);\n');
fprintf(fid,'SIGNAL current_state,next_state: estate;\n');
fprintf(fid,'signal dataR,enTX,ready : std_logic;\n');
fprintf(fid,'signal dataSer,TXdata,n_TXdata : std_logic_vector(7 downto 0);\n');
fprintf(fid,'signal Bytes,n_Bytes : std_logic_vector(7 downto 0);\n');
fprintf(fid,'signal countOCT,n_countOCT : std_logic_vector(7 downto 0);\n');
fprintf(fid,'signal cCHIP,n_cCHIP : std_logic_vector(%d downto 0);\n',bitsChip-1);
fprintf(fid,'signal spd,n_spd : std_logic;\n');
fprintf(fid,'signal sel,n_sel : std_logic_vector(%d downto 0);\n',xsize*ysize-1);
fprintf(fid,'signal n_Split,Split : std_logic_vector(%d downto 0);\n',BitsAdd-1);
fprintf(fid,'signal n_cSplit,cSplit : std_logic_vector(%d downto 0);\n',bitsCSplit-1);
fprintf(fid,'signal endSplit : std_logic;\n');
fprintf(fid,'begin\n\n');

fprintf(fid,'uart : RS232\n');
fprintf(fid,'Port Map(\n');
fprintf(fid,'\t      clk => clk,\n');
fprintf(fid,'\t reset => reset,\n');
fprintf(fid,'\t Rx => RX,\n');
fprintf(fid,'\t Tx => TX,\n');
fprintf(fid,'\t RTS => RTS,\n');
fprintf(fid,'\t dataReady => dataR,\n');
fprintf(fid,'\t dataOUT => dataSer);\n');

fprintf(fid,'spiINT : SPIgen\n');
fprintf(fid,'Port Map ( clk => clk,\n');
fprintf(fid,'\t reset => reset,\n');
fprintf(fid,'\t SCLK => SCLK,\n');
fprintf(fid,'\t MOSI => DATA,\n');
fprintf(fid,'\t ready => ready,\n');
fprintf(fid,'\t data => TXdata,\n');
fprintf(fid,'\t enTX => enTX);\n\n');

cCHIP=ones(xsize*ysize,xsize*ysize);
ind=xsize*ysize;
for k=1:xsize*ysize
    cCHIP(k,ind)=0;
    ind=ind-1;
end

fprintf(fid,'NSS <= sel;\n');
fprintf(fid,'decNSS : process(cCHIP,ready,sel)\n');
fprintf(fid,'begin\n');
fprintf(fid,'\t if(ready = ''1'') then\n');
fprintf(fid,'\t\t case cCHIP is\n');
for k=1:xsize*ysize
    fprintf(fid,'\t\t\t      when "%s" =>\n',dec2bin(k-1,bitsChip));
    fprintf(fid,'\t\t\t\t n_sel <= "');
    for m=1:xsize*ysize
        fprintf(fid,'%d',cCHIP(k,m));
    end
    fprintf(fid,'";\n');
end
fprintf(fid,'\t\t\t when others =>\n');
fprintf(fid,'\t\t\t\t n_sel <= (others => ''1'');\n');
    fprintf(fid,'\t\t end case;\n');
fprintf(fid,'\t else\n');
fprintf(fid,'\t\t n_sel <= sel;\n');
fprintf(fid,'\t end if;\n');
fprintf(fid,'end process;\n\n');

fprintf(fid,'process(countOCT,ready,spd)\n');
fprintf(fid,'begin\n');
fprintf(fid,'\t if(ready = ''0'') then\n');
fprintf(fid,'\t\t n_spd <= spd;\n');
fprintf(fid,'\t else\n');
fprintf(fid,'\t\t if(countOCT >= "00001011") then\n');
fprintf(fid,'\t\t\t n_spd <= ''1'';\n');
fprintf(fid,'\t\t else\n');
fprintf(fid,'\t\t\t n_spd <= ''0'';\n');
fprintf(fid,'\t\t end if;\n');
fprintf(fid,'\t end if;\n');
fprintf(fid,'end process;\n\n');
fprintf(fid,'PD <= spd;\n');

ind=0;
for k=1:ysize
    if(ind+Mult*8+bitsSplit-1 > BitsAdd)
        fprintf(fid,'ADD%d <= Split(%d downto %d);\n',k,BitsAdd-1,ind);
    else
        fprintf(fid,'ADD%d <= Split(%d downto %d);\n',k,ind+Mult*8+bitsSplit-1,ind);
    end
    ind=ind + Mult*8 + bitsSplit;
end

fprintf(fid,'\n\n');

fprintf(fid,'splitter : process(cSplit,Split,dataR,dataSer)\n');
fprintf(fid,'begin\n');
fprintf(fid,'n_cSplit <= cSplit;\n');
fprintf(fid,'endSplit <= ''0'';\n');
fprintf(fid,'n_Split <= Split;\n');
fprintf(fid,'\t case cSplit is\n');

ind=0;
for k=1:oct
    fprintf(fid,'\t\t when "%s" =>\n',dec2bin(k-1,bitsCSplit));
    fprintf(fid,'\t\t\t if(dataR = ''1'') then\n');
    if(ind + 7 > BitsAdd-1)
        ex=BitsAdd-ind-1;
        fprintf(fid,'\t\t\t\t n_Split(%d downto %d) <= dataSer(%d downto 0);\n',ind+ex,ind,ex);
    else
        fprintf(fid,'\t\t\t\t n_Split(%d downto %d) <= dataSer;\n',ind+7,ind);
    end
    ind=ind+8;
    fprintf(fid,'\t\t\t\t n_cSplit <= cSplit + 1;\n');
    fprintf(fid,'\t\t\t end if;\n');
end
    fprintf(fid,'\t\t when others =>\n');
fprintf(fid,'\t\t\t endSplit <= ''1'';\n');
fprintf(fid,'\t      end case;\n');
fprintf(fid,'end process;\n\n\n');

fprintf(fid,'FSM:process(current_state,dataR,dataSer,Bytes,countOCT,ready,cCHIP,TXdata,endSpl
fprintf(fid,'begin\n');
fprintf(fid,'\t next_state <= current_state;\n');
fprintf(fid,'\t n_Bytes <= Bytes;\n');
fprintf(fid,'\t enTX <= ''0'';\n');
fprintf(fid,'\t n_countOCT <= countOCT;\n');
fprintf(fid,'\t n_cCHIP <= cCHIP;\n');
fprintf(fid,'\t n_TXdata <= TXdata;\n\n');

fprintf(fid,'\t case current_state is\n');
fprintf(fid,'\t\t when IDLE=>\n');
fprintf(fid,'\t\t\t if(dataR = ''1'' and endSplit = ''1'') then\n');
fprintf(fid,'\t\t\t\t next_state <= CONF_UNIT;\n');
fprintf(fid,'\t\t\t\t n_Bytes <= dataSer;\n');
fprintf(fid,'\t\t\t end if;\n');
    fprintf(fid,'\t\t when CONF_UNIT=>\n');
fprintf(fid,'\t\t\t if(dataR = ''1'') then\n');
fprintf(fid,'\t\t\t\t n_TXdata <= dataSer;\n');
fprintf(fid,'\t\t\t\t if(ready = ''1'') then\n');
fprintf(fid,'\t\t\t\t\t     next_state <= SEND_OCT;\n');
    fprintf(fid,'\t\t\t\t else\n');
fprintf(fid,'\t\t\t\t\t next_state <= WAIT_SEND_OCT;\n');
    fprintf(fid,'\t\t\t\t end if;\n');
fprintf(fid,'\t\t\t end if;\n');
fprintf(fid,'\t\t when WAIT_SEND_OCT =>\n');
    fprintf(fid,'\t\t\t if(ready = ''1'') then\n');
fprintf(fid,'\t\t\t\t next_state <= SEND_OCT;\n');
fprintf(fid,'\t\t\t end if;\n');
fprintf(fid,'\t\t when SEND_OCT =>\n');
fprintf(fid,'\t\t\t enTX <= ''1'';\n');
fprintf(fid,'\t\t\t next_state <= WAIT_TX;\n');
fprintf(fid,'\t\t\t n_countOCT <= countOCT + 1;\n');
fprintf(fid,'\t\t when WAIT_TX =>\n');
fprintf(fid,'\t\t\t if(countOCT < Bytes) then\n');
fprintf(fid,'\t\t\t\t next_state <= CONF_UNIT;\n');
fprintf(fid,'\t\t\t else\n');
fprintf(fid,'\t\t\t\t n_cCHIP <= cCHIP + 1;\n');
fprintf(fid,'\t\t\t\t next_state <= IDLE;\n');
fprintf(fid,'\t\t\t\t n_Bytes <= (others => ''0'');\n');
fprintf(fid,'\t\t\t\t n_countOCT <= (others => ''0'');\n');
fprintf(fid,'\t\t\t end if;\n');
fprintf(fid,'\t\t when others =>\n');
fprintf(fid,'\t\t\t next_state <= IDLE;\n');
fprintf(fid,'\t\t end case;\n');
fprintf(fid,'end process;\n\n\n');

fprintf(fid,'process(clk,reset)\n');
fprintf(fid,'begin\n');
fprintf(fid,'\t if(reset = ''1'') then\n');
fprintf(fid,'\t\t current_state <= IDLE;\n');
fprintf(fid,'\t\t Bytes <= (others => ''0'');\n');
    fprintf(fid,'\t\t TXdata <= (others => ''0'');\n');
    fprintf(fid,'\t\t countOCT <= (others => ''0'');\n');
    fprintf(fid,'\t\t cCHIP <= (others => ''0'');\n');
    fprintf(fid,'\t\t spd <= ''1'';\n');
    fprintf(fid,'\t\t sel <= (others => ''0'');\n');
    fprintf(fid,'\t\t Split <= (others => ''0'');\n');
    fprintf(fid,'\t\t cSplit <= (others => ''0'');\n');
    fprintf(fid,'\t elsif(clk = ''1'' and clk''event) then\n');
    fprintf(fid,'\t\t current_state <= next_state;\n');
    fprintf(fid,'\t\t Bytes <= n_Bytes;\n');
    fprintf(fid,'\t\t TXdata <= n_TXdata;\n');
    fprintf(fid,'\t\t countOCT <= n_countOCT;\n');
    fprintf(fid,'\t\t cCHIP <= n_cCHIP;\n');
    fprintf(fid,'\t\t spd <= n_spd;\n');
    fprintf(fid,'\t\t sel <= n_sel;\n');
```

```
        fprintf(fid,'\t\t Split <= n_Split;\n');
        fprintf(fid,'\t\t cSplit <= n_cSplit;\n');
        fprintf(fid,'\t end if;\n');
    fprintf(fid,'end process;\n\n');
    fprintf(fid,'end Behavioral;\n');
    fclose(fid);
end
```

11.2.13 Configuration block VHDL description for the source-driven routing

```
function writeProgNetworkSour(xsize,ysize,Mult)
bitsSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsSplit < Mult+1)
        bitsSplit = bitsSplit + 1;
    else
        flag=0;
    end
end

BitsAdd=ysize*(Mult*8+bitsSplit);
oct=floor(BitsAdd/8);
if(rem(BitsAdd,8) ~= 0)
    oct=oct + 1;
end

bitsCSplit=1;
flag=1;
while(flag == 1)
    if(2^bitsCSplit < oct)
        bitsCSplit = bitsCSplit + 1;
    else
        flag=0;
    end
end

bitsChip=1;
flag=1;
while(flag == 1)
    if(2^bitsChip< xsize*ysize)
        bitsChip = bitsChip + 1;
    else
        flag=0;
    end
end


fid=fopen('progNETWORK.vhd','w');
if(fid == -1)
    fprintf('Error creating auxiliar progNETWORK file\n');
else
    fprintf(fid,'library IEEE;\n');
    fprintf(fid,'use IEEE.STD_LOGIC_1164.ALL;\n');
    fprintf(fid,'USE IEEE.std_logic_arith.all;\n');
    fprintf(fid,'USE IEEE.std_logic_unsigned.all;\n\n\n');
    fprintf(fid,'entity progNETWORK is\n');
    fprintf(fid,'Port ( clk : in STD_LOGIC;\n');
    fprintf(fid,'\t reset : in STD_LOGIC;\n');
    fprintf(fid,'\t RX : in STD_LOGIC;\n');
    fprintf(fid,'\t TX : out STD_LOGIC;\n');
    fprintf(fid,'\t RTS : out STD_LOGIC;\n');
    fprintf(fid,'\t DATA : out STD_LOGIC;\n');
    fprintf(fid,'\t SCLK : out STD_LOGIC;\n');
    fprintf(fid,'\t PD : out STD_LOGIC;\n');
    fprintf(fid,'\t NSS : out STD_LOGIC_VECTOR(%d downto 0);\n',xsize*ysize-1);
    for k=1:ysize
        if(k == ysize)
            fprintf(fid,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0));\n',k,Mult*8+bitsSplit-1);
        else
            fprintf(fid,'\t ADD%d : out STD_LOGIC_VECTOR(%d downto 0);\n',k,Mult*8+bitsSplit-1);
        end
    end
    fprintf(fid,'end progNETWORK;\n\n\n');
        fprintf(fid,'architecture Behavioral of progNETWORK is\n\n\n');

    fprintf(fid,'component RS232\n');
    fprintf(fid,'Port(\n');
    fprintf(fid,'\t clk: IN  STD_LOGIC;\n');
    fprintf(fid,'\t reset: IN STD_LOGIC;\n');
    fprintf(fid,'\t Rx: IN STD_LOGIC;\n');
    fprintf(fid,'\t Tx: OUT STD_LOGIC;\n');
    fprintf(fid,'\t RTS: OUT STD_LOGIC;\n');
    fprintf(fid,'\t dataReady: OUT STD_LOGIC;\n');
    fprintf(fid,'\t dataOUT: OUT STD_LOGIC_VECTOR(7 downto 0));\n');
    fprintf(fid,'end component;\n\n');

    fprintf(fid,'component SPIgen is\n');
    fprintf(fid,'Port ( clk : in  STD_LOGIC;\n');
    fprintf(fid,'\t reset : in  STD_LOGIC;\n');
    fprintf(fid,'\t SCLK : out  STD_LOGIC;\n');
    fprintf(fid,'\t MOSI : out  STD_LOGIC;\n');
    fprintf(fid,'\t ready : out STD_LOGIC;\n');
    fprintf(fid,'\t data : in  STD_LOGIC_VECTOR (7 downto 0);\n');
    fprintf(fid,'\t enTX : in  STD_LOGIC);\n');
    fprintf(fid,'end component;\n');

    fprintf(fid,'TYPE estate is (IDLE,PREV_CONF_UNIT,CONF_UNIT,WAIT_SEND_OCT,SEND_OCT,WAIT_TX);\n'
    fprintf(fid,'SIGNAL current_state,next_state: estate;\n');
    fprintf(fid,'signal dataR,enTX,ready : std_logic;\n');
    fprintf(fid,'signal dataSer,TXdata,n_TXdata : std_logic_vector(7 downto 0);\n');
    fprintf(fid,'signal Bytes,n_Bytes : std_logic_vector(15 downto 0);\n');
    fprintf(fid,'signal countOCT,n_countOCT : std_logic_vector(15 downto 0);\n');
    fprintf(fid,'signal cCHIP,n_cCHIP : std_logic_vector(%d downto 0);\n',bitsChip-1);
    fprintf(fid,'signal spd,n_spd : std_logic;\n');
    fprintf(fid,'signal sel,n_sel : std_logic_vector(%d downto 0);\n',xsize*ysize-1);
    fprintf(fid,'signal n_Split,Split : std_logic_vector(%d downto 0);\n',BitsAdd-1);
    fprintf(fid,'signal n_cSplit,cSplit : std_logic_vector(%d downto 0);\n',bitsCSplit-1);
    fprintf(fid,'signal endSplit : std_logic;\n');
    fprintf(fid,'begin\n\n');

    fprintf(fid,'uart : RS232\n');
    fprintf(fid,'Port Map(\n');
    fprintf(fid,'\t     clk => clk,\n');
    fprintf(fid,'\t reset => reset,\n');
    fprintf(fid,'\t Rx => RX,\n');
    fprintf(fid,'\t Tx => TX,\n');
    fprintf(fid,'\t RTS => RTS,\n');
    fprintf(fid,'\t dataReady => dataR,\n');
    fprintf(fid,'\t dataOUT => dataSer);\n');

    fprintf(fid,'spiINT : SPIgen\n');
    fprintf(fid,'Port Map ( clk => clk,\n');
    fprintf(fid,'\t reset => reset,\n');
    fprintf(fid,'\t SCLK => SCLK,\n');
    fprintf(fid,'\t MOSI => DATA,\n');
    fprintf(fid,'\t ready => ready,\n');
    fprintf(fid,'\t data => TXdata,\n');
    fprintf(fid,'\t enTX => enTX);\n\n');

    cCHIP=ones(xsize*ysize,xsize*ysize);
    ind=xsize*ysize;
    for k=1:xsize*ysize
```

```
        cCHIP(k,ind)=0;
        ind=ind-1;
    end

    fprintf(fid,'NSS <= sel;\n');
    fprintf(fid,'decNSS : process(cCHIP,ready,sel)\n');
    fprintf(fid,'begin\n');
    fprintf(fid,'\t if(ready = ''1'') then\n');
    fprintf(fid,'\t\t case cCHIP is\n');
    for k=1:xsize*ysize
        fprintf(fid,'\t\t\t      when "%s" =>\n',dec2bin(k-1,bitsChip));
        fprintf(fid,'\t\t\t\t n_sel <= "');
        for m=1:xsize*ysize
            fprintf(fid,'%d',cCHIP(k,m));
        end
        fprintf(fid,'";\n');
    end
    fprintf(fid,'\t\t\t when others =>\n');
    fprintf(fid,'\t\t\t\t n_sel <= (others => ''1'');\n');
        fprintf(fid,'\t\t end case;\n');
    fprintf(fid,'\t else\n');
    fprintf(fid,'\t\t n_sel <= sel;\n');
    fprintf(fid,'\t end if;\n');
    fprintf(fid,'end process;\n\n');

    fprintf(fid,'process(countOCT,ready,spd)\n');
    fprintf(fid,'begin\n');
    fprintf(fid,'\t if(ready = ''0'') then\n');
    fprintf(fid,'\t\t n_spd <= spd;\n');
    fprintf(fid,'\t else\n');
    fprintf(fid,'\t\t if(countOCT >= "0000000000001001") then\n');
    fprintf(fid,'\t\t\t n_spd <= ''1'';\n');
    fprintf(fid,'\t\t else\n');
    fprintf(fid,'\t\t\t n_spd <= ''0'';\n');
    fprintf(fid,'\t\t end if;\n');
    fprintf(fid,'\t end if;\n');
    fprintf(fid,'end process;\n\n');
    fprintf(fid,'PD <= spd;\n');

    ind=0;
    for k=1:ysize
        if(ind+Mult*8+bitsSplit-1 > BitsAdd)
            fprintf(fid,'ADD%d <= Split(%d downto %d);\n',k,BitsAdd-1,ind);
        else
            fprintf(fid,'ADD%d <= Split(%d downto %d);\n',k,ind+Mult*8+bitsSplit-1,ind);
        end
        ind=ind + Mult*8 + bitsSplit;
    end

    fprintf(fid,'\n\n');

    fprintf(fid,'splitter : process(cSplit,Split,dataR,dataSer)\n');
    fprintf(fid,'begin\n');
    fprintf(fid,'n_cSplit <= cSplit;\n');
    fprintf(fid,'endSplit <= ''0'';\n');
    fprintf(fid,'n_Split <= Split;\n');
    fprintf(fid,'\t case cSplit is\n');

    ind=0;
    for k=1:oct
        fprintf(fid,'\t\t when "%s" =>\n',dec2bin(k-1,bitsCSplit));
        fprintf(fid,'\t\t\t if(dataR = ''1'') then\n');
        if(ind + 7 > BitsAdd-1)
            ex=BitsAdd-ind-1;
            fprintf(fid,'\t\t\t\t n_Split(%d downto %d) <= dataSer(%d downto 0);\n',ind+ex,ind,ex);
        else
            fprintf(fid,'\t\t\t\t n_Split(%d downto %d) <= dataSer;\n',ind+7,ind);
        end
        ind=ind+8;
        fprintf(fid,'\t\t\t\t n_cSplit <= cSplit + 1;\n');
        fprintf(fid,'\t\t\t end if;\n');
    end
        fprintf(fid,'\t\t when others =>\n');
    fprintf(fid,'\t\t\t endSplit <= ''1'';\n');
    fprintf(fid,'\t      end case;\n');
    fprintf(fid,'end process;\n\n\n');

    fprintf(fid,'FSM:process(current_state,dataR,dataSer,Bytes,countOCT,ready,cCHIP,TXdata,endSpli
    fprintf(fid,'begin\n');
    fprintf(fid,'\t next_state <= current_state;\n');
    fprintf(fid,'\t n_Bytes <= Bytes;\n');
    fprintf(fid,'\t enTX <= ''0'';\n');
    fprintf(fid,'\t n_countOCT <= countOCT;\n');
    fprintf(fid,'\t n_cCHIP <= cCHIP;\n');
    fprintf(fid,'\t n_TXdata <= TXdata;\n\n');

    fprintf(fid,'\t case current_state is\n');
    fprintf(fid,'\t\t when IDLE=>\n');
    fprintf(fid,'\t\t\t if(dataR = ''1'' and endSplit = ''1'') then\n');
    fprintf(fid,'\t\t\t\t next_state <= PREV_CONF_UNIT;\n');
    fprintf(fid,'\t\t\t\t n_Bytes(7 downto 0) <= dataSer;\n');
    fprintf(fid,'\t\t\t end if;\n');
    fprintf(fid,'\t\t when PREV_CONF_UNIT=>\n');
    fprintf(fid,'\t\t\t if(dataR = ''1'' and endSplit = ''1'') then\n');
    fprintf(fid,'\t\t\t\t next_state <= CONF_UNIT;\n');
    fprintf(fid,'\t\t\t\t n_Bytes(15 downto 8) <= dataSer;\n');
    fprintf(fid,'\t\t\t end if;\n');
        fprintf(fid,'\t\t when CONF_UNIT=>\n');
    fprintf(fid,'\t\t\t if(dataR = ''1'') then\n');
    fprintf(fid,'\t\t\t\t n_TXdata <= dataSer;\n');
    fprintf(fid,'\t\t\t\t if(ready = ''1'') then\n');
    fprintf(fid,'\t\t\t\t\t      next_state <= SEND_OCT;\n');
        fprintf(fid,'\t\t\t\t else\n');
    fprintf(fid,'\t\t\t\t\t next_state <= WAIT_SEND_OCT;\n');
        fprintf(fid,'\t\t\t\t end if;\n');
    fprintf(fid,'\t\t\t end if;\n');
    fprintf(fid,'\t\t when WAIT_SEND_OCT =>\n');
        fprintf(fid,'\t\t\t if(ready = ''1'') then\n');
    fprintf(fid,'\t\t\t\t next_state <= SEND_OCT;\n');
    fprintf(fid,'\t\t\t end if;\n');
    fprintf(fid,'\t\t when SEND_OCT =>\n');
    fprintf(fid,'\t\t\t enTX <= ''1'';\n');
    fprintf(fid,'\t\t\t next_state <= WAIT_TX;\n');
    fprintf(fid,'\t\t\t n_countOCT <= countOCT + 1;\n');
    fprintf(fid,'\t\t when WAIT_TX =>\n');
    fprintf(fid,'\t\t\t if(countOCT < Bytes) then\n');
    fprintf(fid,'\t\t\t\t next_state <= CONF_UNIT;\n');
    fprintf(fid,'\t\t\t else\n');
    fprintf(fid,'\t\t\t\t n_cCHIP <= cCHIP + 1;\n');
    fprintf(fid,'\t\t\t\t next_state <= IDLE;\n');
    fprintf(fid,'\t\t\t\t n_Bytes <= (others => ''0'');\n');
    fprintf(fid,'\t\t\t\t n_countOCT <= (others => ''0'');\n');
    fprintf(fid,'\t\t\t end if;\n');
    fprintf(fid,'\t\t when others =>\n');
    fprintf(fid,'\t\t\t next_state <= IDLE;\n');
    fprintf(fid,'\t\t end case;\n');
    fprintf(fid,'end process;\n\n\n');

    fprintf(fid,'process(clk,reset)\n');
    fprintf(fid,'begin\n');
    fprintf(fid,'\t if(reset = ''1'') then\n');
    fprintf(fid,'\t\t current_state <= IDLE;\n');
    fprintf(fid,'\t\t Bytes <= (others => ''0'');\n');
        fprintf(fid,'\t\t TXdata <= (others => ''0'');\n');
        fprintf(fid,'\t\t countOCT <= (others => ''0'');\n');
        fprintf(fid,'\t\t cCHIP <= (others => ''0'');\n');
```

```
        fprintf(fid,'\t\t spd <= ''1'';\n');
        fprintf(fid,'\t\t sel <= (others => ''0'');\n');
        fprintf(fid,'\t\t Split <= (others => ''0'');\n');
        fprintf(fid,'\t\t cSplit <= (others => ''0'');\n');
        fprintf(fid,'\t elsif(clk = ''1'' and clk''event) then\n');
        fprintf(fid,'\t\t current_state <= next_state;\n');
        fprintf(fid,'\t\t Bytes <= n_Bytes;\n');
        fprintf(fid,'\t\t TXdata <= n_TXdata;\n');
        fprintf(fid,'\t\t countOCT <= n_countOCT;\n');
        fprintf(fid,'\t\t cCHIP <= n_cCHIP;\n');
        fprintf(fid,'\t\t spd <= n_spd;\n');
        fprintf(fid,'\t\t sel <= n_sel;\n');
        fprintf(fid,'\t\t Split <= n_Split;\n');
        fprintf(fid,'\t\t cSplit <= n_cSplit;\n');
        fprintf(fid,'\t end if;\n');
    fprintf(fid,'end process;\n\n');
    fprintf(fid,'end Behavioral;\n');
    fclose(fid);
end
```

References

[1] S. Grossberg, E. Mingolla, and J. Williamson, "Synthetic aperture radar processing by a multiple scale neural system for boundary and surface representation," *Neural Networks*, vol. 8, pp. 1005–1028, 1995. 3, 19

[2] T. Serre, L. Wolf, S. Bileschi, M. Reisenhuber, and T. Poggio, "Robust object recognition with cortex-like mechanism," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, pp. 411–426, March 2007. 3, 12, 19, 21, 65

[3] E. Culurciello, R. Etienne-Cummings, and K. Boahen, "A biomorphic digital image sensor," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 281– 294, February 2003. 3, 5

[4] J. Costas-Santos, T. Serrano-Gotarredona, R. Serrano-Gotarredona, and B. Linares-Barranco, "A contrast retina with on-chip calibration for neuromporhic spike based AER vision systems," *IEEE Transactions on Circuits and Systems I*, vol. 54, pp. 1444–1458, July 2007. 3, 5

[5] J. P. Carrasco, T. Serrano-Gotarredona, C. Serrano-Gotarredona, B. Acha, and B. Linares-Barranco, "On the computational power of AER vision processing hardware," *Proceedings of the XXII International Conference Design Circuits Integrated Systems*, 2007. 4

[6] M. Silvilotti, "Wiring considerations in analog VLSI systems with application to field-programmable networks." PhD. Thesis, California Institute of Technology, Pasadena, CA, 1991. 4

[7] A. Ruedi, P. Heim, F. Kaess, E. Grenet, F. Heitger, P. Burgi, S. Gyger, and P. Nussbaum, "A 128x128 pixel 120-dB dynamic-range vision-sensor chip for image contrast and orientation extraction," *IEEE Journal of Solid-State Circuits*, vol. 38, pp. 2325– 2333, February 2003. 5

[8] C. Shoushun and A. Bermak, "Arbitrated time-to-first spike CMOS image sensor with on-chip histogram equalization," *IEEE Transactions on VLSI systems*, vol. 15, pp. 346–357, March 2007. 5

[9] M. Azadmehr, J. Abrahamsen, and P. Hafliger, "A foveated AER imager chip," *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 2751–2754, Kobe Japan 2005. 5

[10] K. Zaghloul and K. Boahen, “Optic nerve signals in a neuromorphic chip,” *IEEE Transactions on Biomedical Engineering*, vol. 51, pp. 657–675, April 2004. 5

[11] P. Lichtsteiner, C. Posch, and T. Delbrück, “A 128x128 120dB 30mW asynchronous vision sensor that responds to relative intensity change,” *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 566–576, February 2008. 5, 21, 60

[12] J. Lazzaro, “Silicon auditory processors as computer peripherals,” *IEEE Transactions on Neural Networks*, vol. 4, pp. 523–528, May 1993. 5

[13] G. Cauwenberghs, N. Kumar, W. Himmelbauer, and A. Andreou, “An analog VLSI chip with asynchronous interface for auditory feature extraction,” *IEEE Transactions on Circuits and Systems I*, vol. 45, pp. 600–606, May 1998.

[14] V. Chan, S. Liu, and A. van Schaik, “AER EAR: a matched silicon cochlea pair with address event representation interface,” *IEEE Transactions on Circuits and Systems I*, vol. 54, pp. 48–59, January 2007. 5

[15] E. Chicca, A. Whatley, P. Lichtsteiner, V. Dante, T. Delbrück, P. D. Giudice, R. Douglas, and G. Indiveri, “A multi-chip pulse-based neuromorphic infrastructure and its application to a model of orientation selectivity,” *IEEE Transactions on Circuits and Systems I*, vol. 54, pp. 981–993, May 2007. 5, 17, 30, 32

[16] M. Oster, Y. Wang, R. Douglas, and S.-C. Liu, “Quantification of a spiked-based winner-take-all VLSI network,” *IEEE Transactions on Circuits and Systems I*, vol. 55, pp. 3160–3169, 2008. 5

[17] P. Vernier, A. Mortara, X. Arreguit, and E. Vittoz, “An integrated cortical layer for orientation enhancement,” *IEEE Journal of Solid-State Circuits*, vol. 32, pp. 177–186, February 1997. 5, 24, 32

[18] T. Choi, P. Merolla, J. Arthur, K. Boahen, and B. Shi, “Neuromorphic implementation of orientation hypercolumns,” *IEEE Transactions on Circuits and Systems I*, vol. 52, pp. 1049–1060, 2005. 24, 32

[19] T. Serrano-Gotarredona, A. Andreou, and B. Linares-Barranco, “AER image filtering architecture for vision processing systems,” *IEEE Transactions on Circuits and Systems II*, vol. 46, pp. 1064–1071, September 1999. 24

[20] R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jiménez, and B. Linares-Barranco, “A neuromorphic cortical-layer microchip for spike-based event processing vision systems,” *IEEE Transactions on Circuits and Systems I*, vol. 53, pp. 2548–2566, December 2006. 21, 24, 32, 151, 157

[21] R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jiménez, C. Serrano-Gotarredona, J. Pérez-Carrasco, B. Linares-Barranco, A. Linares-Barranco, G. Jiménez-Moreno, and A. Civit-Ballcels, “On real-time AER 2-D convolutions hardware for neuromorphic spike based cortical processing,” *IEEE Transactions on Neural Networks*, vol. 19, pp. 1196–1219, July 2008. 21, 24

[22] L. Camuñas-Mesa, A. Acosta-Jiménez, C. Zamarreño-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, “A 32x32 pixel convolution processor chip for address event vision sensors with 155ns event latency and 20Meps throughput,” *IEEE Transactions on Circuits and Systems*, vol. 58, pp. 777–790, April 2011. 24, 32, 60, 71

[23] L. Camuñas-Mesa, C. Zamarreño-Ramos, A. Linares-Barranco, A. Acosta-Jiménez, T. Serrano-Gotarredona, and B. Linares-Barranco, “An event-driven multi-kernel convolution processor module for event-driven vision sensors,” *IEEE Journal of Solid-State Circuits*, in Press. 5, 24, 71, 151, 157

[24] T. Teixeira, A. Andreou, and E. Culurciello, “Event-based imaging with active illumination in sensor networks,” *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 644–647, 2005. 5

[25] A. Delorme, L. Perrinet, and S. Thorpe, “Networks of integrate-and-fire neurons using rank-order coding B: spike timing dependence plasticity and emergence of orientation selectivity,” *Neurocomputing*, pp. 539–545, 2001. 5

[26] C. Shoushun and A. Bermak, “A low power CMOS imager based on time-to-first-spike encoding and fair AER,” *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, vol. 5, pp. 5306– 5309, May 2005. 5

[27] R. Williams and K. Herrup, “The control of neuron number,” *Annual Review on Neuroscience*, vol. 11, pp. 423–453, 1988. 11

[28] B. Pakkenberg and H. Gundersen, “Neocortical neuron number in humans: effect of sex and age,” *Journal of Comparative Neurology*, vol. 384, pp. 312–320, 1997. 11

[29] R. Collobert and J. Weston, “An unified architecture for natural language processing: deep natural networks with multitask learning,” *Proc. Int. Conf. on Machine Learning (ICML 2008)*, pp. 160–167, 2008. 12, 21

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, 1998. 21

[31] C. Garcia and M. Delakis, “Convolutional face finder: A neural architecture for fast and robust face detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 1408–1423, November 2004. 19

[32] P. Simard, D. Steinkraus, and J. Platt, “Best practices for convolutional neural networks applied to visual document analysis,” *Seventh International Conference on Document Analysis and Recognition*, pp. 958– 963, Aug. 2003. 19

[33] K. Chellapilla, M. Shilman, and P. Simard, “Optimally combining a cascade of classifiers,” *Proc. of Document Recognition and Retrieval 13, Electronic Imaging*, p. 6067, 2006.

[34] K. Chellapilla, S. Puri, and P. Simard, “High performance convolutional neural networks for document processing,” *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[35] A. Abdulkader, “A two-tier approach for arabic offline handwriting recognition,” *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[36] K. Chellapilla and P. Simard, “A new radical based approach to offline handwritten east-asian character recognition,” *Tenth International Workshop on Frontiers in Handwriting Recognition*, 2006. 19

[37] A. Ahmed, Y. Gong, and E. Xing, “Training hierarchical feed-forward visual recognition models using transfer learning from pseudo-tasks,” *Proceedings of the 10th European Conference on Computer Vision: Part III*, 2008. 19

[38] J. Weston, F. Ratle, and R.Collobert, “Deep learning via semi-supervised embedding,” *Proceedings of the 25th international conference on Machine learning*, 2008. 12, 19

[39] V. J. et al, “Supervised learning of image restoration with convolutional networks,” *IEEE 11th International Conference on Computer Vision (ICCV 2007)*, pp. 1–8, October 2007. 12, 21

[40] R. Hadsell, P. Sermanet, M. Scoffier, A. Erkan, K. Kavackuoglu, U. Muller, and Y. LeCun, “Learning long-range vision for autonomous off-road driving,” *Journal Field Robotics*, vol. 26(2), pp. 120–144, February 2009. 12, 21

[41] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay, “Large scale FPGA-based convolutional neural networks,” *in R. Bekkerman, M. Bilenko and J. Langford (Eds) Machine Learning on Very Large Data Sets*, vol. Cambridge University Press, 2011. 12, 26

[42] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “NeuFlow: a runtime-reconfigurable dataflow processor for vision,” *Proceedings of Embedded Computer Vision Workshop (ECVW’11)*, 2011. 12, 26

[43] Y. Shi, S. B. Furber, J. Garside, and L. Plana, “Fault tolerant delay insensitive inter-chip communication,” *Proceeding of 15th IEEE International Symposium on Asynchronous Circuits and Systems, Async 2009*, vol. 12, pp. 78–847, May 2009. 12

[44] X. Jin, M. Luján, L. Plana, S. Davies, S. Temple, and S. Furber, “Modeling spiking neural networks on SpiNNaker,” *Computing in Science and Engineering*, vol. 12, pp. 91–97, September-October 2010. 12

[45] J. Schemmel, J. Fieres, and K. Meier, “Wafer-scale integration of analog neural networks,” *IEEE International Joint Conference on Neural Networks (IJCNN 2008)*, pp. 431–438, June 2008. 14, 15

[46] S. Hartmann, S. Schiefer, S. Scholze, J. Partzsch, C. Mayr, S. Henker, and R. Schüffny, “Highly integrated packet-based AER communication infrastructure with 3Gevents/s throughput,” *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pp. 950–953, 2010. 16, 32, 35, 75, 76

[47] R. Vogelstein, U. Mallik, J. Vogelstein, and G. Cauwenberghs, “Dynamically reconfigurable silicon array of spiking neurons with conductance-based synapses,” *IEEE Transactions on Neural Networks*, vol. 18, pp. 253–265, January 2007. 17, 30, 31, 32

[48] S. Joshi, S. Deiss, M. Arnold, J. Park, T. Yu, and G. Cauwenberghs, “Scalable event routing in hierarchical neural array architecture with global synaptic connectivity,” *12th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA)*, pp. 1–6, February 2010. 17, 31, 34

[49] K. Boahen, “A burst-mode word-serial address-event link-I: transmitter design,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, pp. 1269–1280, July 2004. 18

[50] ——, “A burst-mode word-serial address-event link-II: receiver design,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, pp. 1281– 1291, July 2004.

[51] ——, “A burst-mode word-serial address-event link-III: analysis and test results,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, pp. 1292–1300, July 2004. 18

[52] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camuñas-Mesa, R. Berner, M. Rivas-Pérez, T. Delbrück, S.-C. Liu, R. Douglas, P. Hafliger, G. Jimenez-Moreno, A. C. Ballcels, T. Serrano-Gotarredona, A. Acosta-Jimenez, and B. Linares-Barranco, “CAVIAR: a 45K-neuron, 5M-synapse 12G-connects/s AER hardware sensory-processing-learning-actuating system for high speed visual object recognition and tracking,” *IEEE Transactions on Neural Networks*, vol. 20, pp. 1417–1438, September 2009. 18, 21, 31, 33

[53] E. T. Rolls and G. Deco, “Computational neuroscience of vision.” Oxford University Press, 2002. 19

[54] R. DeValois, D. Albrecht, and L. Thorell, “Spatial frequency selectivity of cells in macaque visual cortex,” *Vision Research*, vol. 22, pp. 545–559, 1982. 19

[55] R. DeValois, E. Yund, and N. Hepler, “The orientation and direction selectivity of cells in macaque visual cortex,” *Vision Research*, vol. 22, pp. 531–544, 1982.

[56] P. Schiller, B. Finlay, and S. Volman, “Quantitative studies of single-cell properties in monkey striate cortex. Spatial frequency,” *Journal of Neurophysiology*, vol. 39, pp. 1334–1351, 1976. 19

[57] K. Fukushima, “Visual feature extraction by a multilayered network of analog threshold elements,” *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-5, pp. 322–333, October 1969. 19

[58] K. Fukushima and K. Wake, “Handwritten alphanumeric character recognition by the Neocognitron,” *IEEE Transactions on Neural Networks*, vol. 2, pp. 355–365, May 1991. 19, 65

[59] Y. LeCun and Y. Bengio, “Convolutional networks for images, speech and time series,” *The Handbook of Brain Science and Neural Networks. M. Arbib (Ed.), Cambridge, MA: MIT Press.*, pp. 255–258, 1995. 19

[60] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, November 1998.

[61] C. Neubauer, “Evaluation of convolution neural networks for visual recognition,” *IEEE Transactions on Neural Networks*, vol. 9, pp. 685–696, 1998.

[62] S. Lawrence, C. Giles, A. Tsoi, and A. Back, “Face recognition: a convolutional neural network approach,” *IEEE Transactions on Neural Networks*, vol. 8, pp. 98–113, 1997. 19

[63] S. Thorpe, D. Fize, and C. Marlot, “Speed of processing in the human visual system,” *Nature*, vol. 381, pp. 520–2, 1996. 21

[64] S. Thorpe, R. Guyonneaua, N. Guilbauda, J. M. Allegrauda, and R. VanRullen, “SpikeNet: real-time visual processing with one spike per neuron,” *Neurocomputing*, vol. 58-60, pp. 857–64, 2004. 21

[65] K. Boahen, “Point-to-point connectivity between neuromorphic chips using address events,” *IEEE Transactions on Circuits and Systems Part II*, vol. 47, pp. 416–434, May 2000. 24

[66] F. Nasse, C. Thurau, and G. Fink, “Face detection using GPU-based convolutional neural networks,” *Lectures Notes in Computer Science*, vol. 5702, Computer Analysis of Images and Patterns, pp. 83–90, 2009. 26

[67] M. Sankaradas, V. Jakkul, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. Graff, “A massively parallel coprocessor for convolutional neural networks,” *20th IEEE International Conference on Application-Specific Systems, Architectures and Processors.*, pp. 53–58, 2009. 26

[68] P. Merolla, J. Arthur, B. E. Shi, and K. Boahen, “Programmable connections in neuromorphic grids,” *Proceedings of International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 80–84, August 2006. 31, 33

[69] J. Lin, P. Merolla, J. Arthur, and K. Boahen, "Expandable networks for neuromorphic chips," *IEEE Transactions on Circuits and Systems, Part-I*, vol. 54, pp. 301–311, February 2007. 33

[70] S. Bamford, A. Murray, and D. Willshaw, "Large developing receptive fields using a distributed and locally reprogrammable address-event receiver," *IEEE Transactions on Neural Networks*, vol. 21, pp. 286–304, February 2010. 31, 33

[71] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber, "SpiNNaker: Mapping neural networks onto a massively-parallel chip multiprocessor," *IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 2849–2856, June 2008. 31, 35

[72] J. Fieres, J. Schemmel, and K. Meier, "Realizing biological spiking network models in a configurable wafer-scale hardware system," *IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 969–976, June 2008. 31, 35

[73] C. Mayr, H. Eisenreich, S. Henker, and R. Schüffny, "Pulsed multi-layered image filtering: A VLSI implementation," *International Journal of Applied Mathematics and Computer Sciences)*, vol. 1, pp. 60–65, 2005. 30, 32

[74] A. Cassidy, A. Andreou, and J. Georgiou, "Design of a one million neuron single FPGA neuromorphic system for real-time multimodal scene analysis," *45th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, March 2011. 30, 50

[75] Y. Wang and S.-C. Liu, "Programmable synaptic weights for an aVLSI network of spiking neurons," *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems*, pp. 4531–4534, May 2006. 32

[76] A. Cassidy, T. Murray, A. Andreou, and J. Georgiou, "Evaluating On-Chip Interconnects for Low Operating Frequency Silicon Neuron Arrays," *IEEE Int. Symp. on Circ. and Syst. (ISCAS)*, pp. 2437–2440, May 2011. 32

[77] D. Gross and C. M. Harris, "Fundamentals of queueing theory," *3rd ed. John Wiley & Sons, Inc*, 1998. 32

[78] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computers)*, vol. 35, pp. 70–78, January 2002. 35

[79] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar, "A 2 Tb/s 6×4 mesh network for a single-chip cloud computer with DVFS in 45nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 46, pp. 757–766, April 2011. 35

[80] J. Howard, S. Dighe, S. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, and R. V. D. Wijngaart, "A 48-core IA-32 processor in 45 nm CMOS

using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, pp. 173–183, January 2011.

[81] S. Sarkar, G. Kulkarni, P. Pande, and A. Kalyanaraman, "Network-on-chip hardware accelerators for biological sequence alignment," *IEEE Transactions on Computers*, vol. 59, pp. 29–41, January 2010. 35

[82] A. Linares-Barranco, R. Paz-Vicente, A. J. F. Gómez-Rodriguez, M. Rivas, G. Jiménez, and A. Civit, "On the AER convolution processors for FPGA," *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 4237–4240, May 2010. 45, 60, 65

[83] D. Fasnacht, A. Whatley, and G. Indiveri, "A serial communication infrastructure for multi-chip address event systems," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 648–651, May 2007. 47, 75, 76

[84] U. Ogras, P. Bogdan, and R. Marculescu, "An analytical approach for network-on-chip performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 2001–2013, December 2010. 50, 51, 52

[85] J. Hu, U. Y. Ogras, and R. Marculescu, "System-level buffer allocation for application-specific networks-on-chip router design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 2919–2933, December 2006. 50

[86] A. Cassidy and A. Andreou, "Beyond Amdahl's Law: An objective function that links multiprocessor performance gains to delay and energy," *IEEE Transactions on Computers*, 2011. 50

[87] F. Hillier and G. Lieberman, "Introduction to operations research," *6th Ed., New York, McGraw-Hill*, pp. 631–732, 1995. 52

[88] J. A. Leñero-Bardallo, T. Serrano-Gotarredona, and B. Linares-Barranco, "A 3.6μs asynchronous frame-free event-driven dynamic-vision-sensor," *IEEE Journal of Solid-State Circuits*, vol. 46, pp. 1443–1455, June 2011. 60

[89] F. Gómez-Rodríguez, R. Paz-Vicente, A. Linares-Barranco, M. Rivas, L. Miró, S. Vicente, G. Jiménez, and A. Civit, "AER tools for communications and debugging," *Proceedings of the IEEE International Symposium of Circuits and Systems*, pp. 3253–3256, May 2006. 60, 62, 88

[90] J. Pérez-Carrasco, T. Serrano-Gotarredona, C. Serrano, B. Acha, and B. Linares-Barranco, "High-speed character recognition system based on a complex hierarchical AER architecture," *Proceedings of the IEEE International Symposium on Circuits and Systems*. 65

[91] B. Razavi, “Design of integrated circuits for optical communications.” McGraw Hill, New York, 2003. 72

[92] ——, “Challenges in the design high-speed clock and data recovery circuits,” *IEEE Communications Magazine*, vol. 40, pp. 94–101, August 2002. 72

[93] M.-T. Hsieh and G. Sobelman, “Architectures for multi-gigabit wire-linked clock and data recovery,” *IEEE Circuits and Systems Magazine*, vol. 8, pp. 45–57, 2008. 72

[94] J. Terada, K. Nishimura, S. Kimura, H. Katsurai, N. Yoshimoto, and Y. Ohtomo, “A 10.3 Gb/s burst-mode CDR using a $\Delta\Sigma$ DAC,” *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 2921–2928, December 2008. 72, 73

[95] M. Hossain and A. C. Carusone, “5-10Gb/s 70mW burst mode AC coupled receiver in 90-nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 45, pp. 524–537, March 2010. 72, 73

[96] A. Li, J. Faucher, and D. V. Plant, “Burst-mode clock and data recovery in optical multiaccess networks using broad-band PLLs,” *IEEE Photonics Technology Letters*, vol. 18, pp. 73–75. 73, 74

[97] Q. Du, J. Zhuang, and T. Kwasniewski, “A low-power, fast acquisition, data recovery circuit with digital threshold decision for SFI-5 application,” *IEEE Transactions on VLSI Systems*, vol. 17, pp. 1742–1748. 72, 73

[98] M. van Ierssel, A. Sheikholeslami, H. Tamura, and W. W. Walker, “A 3.2 Gb/s CDR using semi-blind oversampling to achieve high jitter tolerance,” *IEEE Journal of Solid-State Circuits*, vol. 42, pp. 2224–2234, October 2007. 72, 73

[99] C. Zamarreno-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, “An instant-startup jitter-tolerant manchester-encoding serializer/deserializer scheme for event-driven bit-serial LVDS interchip AER links,” *IEEE Transactions on Circuits and Systems-I : Regular Papers*, 2011. 73, 76

[100] J. Kim and D.-K. Jeong, “Multi-gigabit-rate clock and data recovery based on blind oversampling,” *IEEE Communications Magazine*, vol. 41, pp. 68–74. 72

[101] S. Ahmed and T. Kwasniewski, “Overview of oversampling clock and data recovery circuits,” *Canadian Conference on Electrical and Computer Engineering*, pp. 1876–1881, May 2005. 72

[102] K. Yamashita, M. Nakata, N. Kamogawa, O. Yumoto, and H. Kodera, “Compact-same-size 52- and 156 Mb/s SDH optical transceiver modules,” *IEEE/OSA Journal of Lightwave Technology*, vol. 12, pp. 1607–1615, September 1994. 74

[103] I. Radovanovic, W. van Etten, and H. Freriks, “Ethernet-based passive optical local-area networks for fiber-to-the-desk application,” *IEEE/OSA Journal of Lightwave Technology*, vol. 21, pp. 2534–2545, November 2003. 74

[104] L. Miró-Amarante, A. Jiménez-Fernández, A. Linares-Barranco, F. Gómez-Rodríguez, G. J. R. Paz, A. Civit, and R. Serrano-Gotarredona, "LVDS serial AER link performance," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1537–1540, May 2007. 76

[105] H. Berge and P. Häfliger, "High-speed serial AER on FPGA," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 857–860, May 2007. 75, 76

[106] P. Popescu, A. Solheim, and M. Wight, "Experimental monolithic high speed transceiver for Manchester encoded data," *Proceedings of the 1995 Bipolar/C-MOS Circuits and Technology Meeting*, pp. 110–113, October 1995. 76, 77, 81, 86

[107] ANSI/TEIA/EIA-644-1995, "Electrical characteristics of low voltage differential signalling (LVDS) interface circuits," *Telecommunications Industry Association*, November 15, 1995. 76

[108] A. Boni, A. Pierazzi, and D. Vecchi, "LVDS I/O interface for Gbp/s-per-pin operation in 0.35μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 706–711, April 2001. 76, 97, 103, 107, 111, 112, 115, 120, 128

[109] N. Semiconductors, "LVDS owner's manual," 4th edition. 76, 97

[110] D. Muller and W. Bartky, "A theory of asynchronous circuits," *Proceedings International Symposium Theory of Switching*, pp. 204–243, 1959. 79

[111] S. M. Mishra, S. S. Rofail, and K. S. Yeo, "Design of high performance double edge-triggered flip-flops," *IEE Proceedings on Circuits, Devices and Systems*, vol. 147, pp. 283–290, October 2000. 82

[112] T. L. W. Chung and M. Sachdev, "A comparative analysis of low-power low-voltage dual-edge-triggered flip-flops," *IEEE Transactions on VLSI Systems*, vol. 10, pp. 913–918, December 2002. 82

[113] S. Cheng, H. Tong, J. Silva-Martinez, and A. I. Karsilayan, "Design and analysis of an ultrahigh-speed glitch-free fully differential charge pump with minimum output current variation and accurate matching," *IEEE Transactions on Circuits and Systems, Part II*, vol. 53, pp. pp. 843–847, September 2006. 85

[114] J. Craninckx and G. V. der Plas, "A 65fJ/conversion-step 0-to-50Ms/s 0-to-0.7mW 9b charge-sharing SAR ADC in 90nm digital CMOS," *IEEE ISSCC Digest of Technical Papers*, pp. 246–600, February 2007. 87

[115] A. A. Abidi, "Phase noise and jitter in CMOS ring oscillators," *IEEE Journal of Solid-State Circuits*, vol. 41, pp. 1803–1816, August 2006. 88

[116] M. Chen, A. Pierazzi, J. Silva-Martínez, M. Nix, and M. Robinson, "Low-voltage low-power LVDS drivers," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 472–479, February 2005. 97, 105, 115, 120

[117] V. Bratov, J. Binkley, V. Katzman, and J. Choma, "Architecture and implementation of low-power LVDS output buffer for high-speed applications," *IEEE Transactions on Circuits and Systems Part-I*, vol. 53, pp. 2101–2108, October 2005. 97, 115

[118] A. Tajalli and Y. Leblebici, "A slew controlled LVDS output driver circuit in 0.18 μm CMOS technology," *IEEE Journal of Solid-State Circuits*, vol. 44, pp. 538–548, February 2009. 97, 115

[119] K. Abugharbieh, S. Krishnan, J. Mohan, V. Devnath, and I. Duzevik, "An ultralow-power 10-Gbits/s LVDS output driver," *IEEE Transactions on Circuits and Systems Part-I*, vol. 57, pp. 262–269, January 2010. 97, 115

[120] H. Lu, H.-W. Wang, C. Su, and C.-N. Liu, "Design of an all-digital LVDS driver," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, pp. 1635–1644, August 2009. 97, 115, 121, 132

[121] P. Heydari and R. Mohanavelu, "Design of ultrahigh-speed low-voltage CMOS CML buffers and latches," *IEEE Transactions on Very Large Scale of Iintegration (VLSI) systems*, vol. 12, pp. 1081–1093, Oct. 2004. 98

[122] A. Boni, "1.2-Gb/s true PECL 100K compatible I/O interface in 0.35-μm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 979–987, Jun. 2001. 98

[123] C. Zamarreño-Ramos, R. Serrano-Gotarredona, T. Serrano-Gotarredona, and B. Linares-Barranco, "LVDS interface for AER links with burst mode operation capability," *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 644–647, May 2008. 101, 110, 112, 126, 128, 130

[124] J. Poulton, R. Palmer, A. M. Fuller, T. Greer, J. Eyles, W. J. Dally, and M. Horowitz, "A 14-mW 6.25-Gb/s transceiver in 90-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 42, pp. 2745–2757, December 2007. 120, 121, 132

[125] K. L. J. Wong, H. Hatamkhani, M. Mansuri, and C. K. K. Yang, "A 27-mW 3.6-Gb/s I/O transceiver," *IEEE Journal of Solid-State Circuits*, vol. 39, pp. 602–612, April 2004. 121, 132

[126] M. Kossel, C. Menolfi, J. Weiss, P. Buchmann, G. von Bueren, L. Rodoni, T. Morf, T. Toifl, and M. Schmatz, "A T-coil-enhanced 8.5Gb/s high-swing SST transmitter in 65nm Bulk CMOS with $\leq$-16 dB return loss over 10Ghz bandwidth," *IEEE Journal of Solid-State Circuits*, vol. 43, pp. 2905–2920, December 2008. 132

[127] K. Fukuda, H. Yamashita, F. Yuki, M. Yagyu, R. Nemoto, T. Takemoto, T. Saito, N. Chujo, K. Yamamoto, H. Kanai, and A. Hayashi, "An 8Gb/s transceiver with 3x-oversampling 2-threshold eye-tracking CDR circuit for -36.8dB-loss backplane," *IEEE International Solid-State Circuits Conference (ISSCC)*, 2008. 132

[128] J. Pérez-Carrasco, B. Acha, C. Serrano, L. Camuñas-Mesa, T. Serrano-Gotarredona, and B. Linares-Barranco, "Fast vision through frameless event-based sensing and convolutional processing: Application to texture recognition," *IEEE Transactions on Neural Networks*, vol. 21, pp. 609–620, April 2010. 135

[129] J. Pérez-Carrasco, C. Serrano, B. Acha, T. Serrano-Gotarredona, and B. Linares-Barranco, "Spike-based convolutional network for real-time processing," *20th International Conference on Pattern Recognition (ICPR)*, pp. 3085–3088, August 2010. 135

[130] J. A. Pérez-Carrasco, "Simulation tool to build and analyze complex and hierarchically structured AER-based systems which implement visual information processing." PhD. Thesis, University of Seville, 2011. 179