A System-Level Simulation Model for a Protocol Processor

Diplomityö
Turun yliopisto
Informaatioteknologian laitos
Sulautetut järjestelmät
2008
Pasi Yliuntinen

Tarkastajat:
Seppo Virtanen
Jani Paakkulainen

TURUN YLIOPISTO
Informaatioteknologian laitos

PASI YLIUNTINEN: A System-Level Simulation Model for a Protocol Processor

Diplomityö, 62 s., 37 liites.
Sulautetut järjestelmät
Huhtikuu 2008

Piirien kompleksisuus kasvaa eksponentiaalisesti puolijohdeteknologian kehityksen noudattaessa Mooren lakina tunnettua trendiä. Perinteisten laitteistokuvauskielten kuten VHDL:n ja Verilogin ilmaisuvoima alkaa käydä riittämättömäksi, eikä kyseisistä kielistä löydy suoraa tukea mm. laitteiston ja ohjelmiston yhteissuunnittelulle. SystemC:n kaltaiset kielet on suunniteltu ratkaisemaan nämä ongelmat yhdistämällä korkean tason ohjelmointikielten ilmaisuvoiman laitteistokuvauskielten laitteistoläheisiin rakenteisiin. Jotta nämä korkeamman abstraktiotason kielet pystyisivät korvaamaan vanhat kielet digitaalijärjestelmien suunnitteluvuossa tulisi niiden olla myös tehokkaasti syntetisoitavissa laitteistoksi.

Nykyaikaiset nopeat verkkotekniikat asettavat verkkolaitteille usein erittäin kireitä reaaliaikaisuuteen ja palvelun laatuun liittyvä vaatimuksia. Samaan aikaan vaaditaan usein matalaa hintaa, pientä kokoa, sekä vähäistä energiankulutusta. Usein laite pitää myös saada nopeasti markkinoille. Näitä vaatimuksia on yhä vaikeampi täyttää perinteisellä tavalla yleiskäyttöisellä prosessorilla. Yksi tapa yhdistää energiatehokkuus ja suuri suorituskyky mutta silti säilyttää joustavuus ja nopea suunnitteluvuo on käyttää ASIP-prosessoreita. Koska eri verkkoprotokollien prosessoinnista voidaan löytää samanlaisia tehtäviä, on mahdollista kehittää protokollaprosessointiin optimoituja arkkitehtuureita. Yksi tällainen on TTA-pohjainen TACO, jonka etuja ovat mm. tehokas rinnakkaisuus, modulaarisuus sekä tehokas ja yksinkertainen käskyjen purkaminen.

Tätä tutkielmaa varten kehitettiin SystemC 2.2 -pohjainen laitteistosimulointiympäristö TACO-arkkitehtuurille käyttäen pohjana edellisellä SystemC-versiolla tehtyä ympäristöä. Ympäristö mahdollistaa simulointimallien nopean konstruktoinnin laitteistolohkoista koostuvan kirjaston ja lohkojen automaattisen konfiguroinnin ja kytkeytymisen ansiosta. Malli mahdollistaa laitteiston ja ohjelmiston samanaikaisen simuloinnin ja verifioinnin. Lisäksi tutkittiin SystemC 1.0:n ja 2.2:n eroja laitteistomallinnuksen kannalta, ja SystemC:llä kirjoitettujen simulointimallien muuntamista synteesikelpoiseksi VHDL:ksi Celoxica Agility SystemC Compiler -työkalulla. Testikäyttöä varten ympäristön avulla kehitettiin simulointimalli TCP/IP-paketteja validoivalle prosessorille.

Asiasanat: SystemC, TTA, protokollaprosessori, systeemitason mallinnus

UNIVERSITY OF TURKU
Department of Information Technology

PASI YLIUNTINEN: A System-Level Simulation Model for a Protocol Processor

Master's Thesis, 62 p., 37 app. p.
Embedded Systems
April 2008

As the development of integrated circuit technology continues to follow Moore's law the complexity of circuits increases exponentially. Traditional hardware description languages such as VHDL and Verilog are no longer powerful enough to cope with this level of complexity and do not provide facilities for hardware/software codesign. Languages such as SystemC are intended to solve these problems by combining the powerful expression of high level programming languages and hardware oriented facilities of hardware description languages. To fully replace older languages in the desing flow of digital systems SystemC should also be synthesizable.

The devices required by modern high speed networks often share the same tight constraints for e.g. size, power consumption and price with embedded systems but have also very demanding real time and quality of service requirements that are difficult to satisfy with general purpose processors. Dedicated hardware blocks of an application specific instruction set processor are one way to combine fast processing speed, energy efficiency, flexibility and relatively low time-to-market. Common features can be identified in the network processing domain making it possible to develop specialized but configurable processor architectures. One such architecture is the TACO which is based on transport triggered architecture. The architecture offers a high degree of parallelism and modularity and greatly simplified instruction decoding.

For this M.Sc.(Tech) thesis, a simulation environment for the TACO architecture was developed with SystemC 2.2 using an old version written with SystemC 1.0 as a starting point. The environment enables rapid design space exploration by providing facilities for hw/sw codesign and simulation and an extendable library of automatically configured reusable hardware blocks. Other topics that are covered are the differences between SystemC 1.0 and 2.2 from the viewpoint of hardware modeling, and compilation of a SystemC model into synthesizable VHDL with Celoxica Agility SystemC Compiler. A simulation model for a processor for TCP/IP packet validation was designed and tested as a test case for the environment.

Keywords: SystemC, TTA, protocol processor, system level modeling

Contents

List of Figures

Chapter 1

Introduction

As the integrated circuit technology continues to follow the trend known as Moore's law the complexity of the circuits increases exponentially. Due to this it is now possible to integrate whole systems on one chip. These kinds of circuits are called System-on-a-Chip or SoC. SoC desing often involves designing both hardware and software portions together.

Traditional hardware description languages such as VHDL and Verilog are no longer powerful enough to cope with this level of complexity and do not provide facilities for hardware/software codesign. Languages such as SystemC are intended to close this gap between desing capacity and integrated circuit capacity and to solve problems with hardware/software codesign and integration by combining the powerful expression of high level programming languages and hardware oriented facilities of hardware description languages. SystemC includes a C++ class library and a simulation kernel that can be used for digital system modeling and verification.

Hardware description languages have an important role in electronic design automation. Models written with VHDL and Verilog are used in hardware synthesis where the model is transformed e.g. into a programming file for a programmable logic circuit or a floorplan for a more or less custom manufactured application specific circuit using a software tool. To fully replace these older languages in the design flow of digital systems

SystemC should also be synthesizable. Tools for this have lately begun to appear.

Computer networks have become a crucial backbone for modern information society. The devices required by these networks often share the same requirements for low price, energy consumption and small size with embedded systems but have also very demanding real time and quality of service requirements that are difficult to satisfy with general purpose processors. As a result dedicated hardware blocks are needed to combine fast processing speed and energy efficiency. Common features can be indentified in the network processing domain, making it possible to develop specialized but configurable processor architectures for network protocol processing. With this kind of platforms it is possible to keep some of the flexibility of general purpose processors while keeping time-to-market reasonable, and also achieve better performance and lower power consumption. One such architecture is the transport triggered TACO architecture.

In this M.Sc.(Tech) thesis a SystemC based simulation environment was designed, implemented and extended using an old version as a starting point. The environment enables simulation and verification of configurable processor instances by means of easy creation of custom hardware blocks and software development with a device specific assembly language. As a test case a simulation model for a processor for TCP/IP packet validation was developed and tested. Synthesizability of this model was also investigated.

Chapter two gives an overview of the TCP/IP protocols and their basic features. Chapter three discusses processor design in general and discusses the TACO architecture in detail. Chapter four gives an overview of SystemC and its features from the standpoint of this thesis. Chapter five describes the goal and the structure of the old and new simulation models, and in chapter six results and observations made during the process of updating the model and testing its synthesizability are presented. Chapter seven provides concluding remarks for the work presented in this thesis.

Chapter 2

Protocols of the Internet

2.1 Layered network architectures

Due to the complexity of modern networks and the need to interconnect different types of computer networks, a monolithic protocol structure is no longer feasible. Instead, most network protocols are organized as a stack of layers that each provide services to the layer above and utilize services provided by the layer below. With each layer the level of abstraction is increased by hiding away technical details. As a result, for example network programming for a high level protocol is possible without taking low level details such as voltage levels or packet routing into consideration. Connecting two peers using a high level protocol is also possible even if the actual physical network topography is complex and consists of a number of different kinds of networks. [1]

Each layer adds at least a header field before the actual data to be transmitted. The header contains control information needed by the protocol, such as sequence numbering, checksums for error detection and possibly correction, timestamps etc. Some protocols add also a trailer to the end of the data.

2.1.1 ISO/OSI reference model

In the beginning of the 1980's an effort was made by the International Standards Organization (ISO) to standardize the functions of protocols used in the various layers [2]. The result was a model that is called the Open Systems Interconnection (OSI) reference model. In this model the protocol stack is divided into seven layers. ISO does not specify the exact services and protocols used in each layer, but it defines what each layer should do [1].

On the bottom in the model is the physical layer that is concerned with sending bits over a physical medium. The second layer is the data link layer whose responsibility is to send data frames with reasonable reliability between two physically connected peers. The third layer, the network layer, routes packets of data in a subnet taking also requirements for quality of service into consideration. Above the network layer is the transport layer that creates an end-to-end connection between source and destination and delivers the data between them reliably. The fifth layer is the session layer that allows different machines to establish sessions between them and handles the needed synchronization. The second last layer is the presentation layer that allows definition and exchange of abstract high level data structures between machines that might have different internal data representations. The highest level is the application layer that contains the protocols that are used by user applications for networking. [1]

2.1.2 TCP/IP reference model

Development of the TCP/IP reference model [3] begun when problems emerged with the ARPANET, a research network sponsored by the U.S. Department of Defense. Interconnected networks that formed the ARPANET were originally connected using telephone lines and existing protocols faced problems when satellite and radio networks were added later. Because of this one of the major design goals was to enable seamless interconnection of multiple types of networks. Another important goal was the ability to maintain

functionality and keep connections intact even in situations where some of the network devices on the route are lost, for example due to malfunctions or especially enemy activity during war [1].

The TCP/IP model divides the protocol stack into four layers. The lowest layer is the host-to-network layer. The model does not however give an exact definition of it other than that it must be able to deliver the packets coming from the internet layer that lays above it. The internet layer is responsible for delivering packets to the destination. Major issues are efficient and robust routing and congestion avoidance and therefore it is similar to the network layer found in OSI model. Above the internet layer is the transport layer that forms end-to-end connections between two machines and is similar to the layer with the same name in OSI model. Two protocols are defined: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP is a connection-oriented protocol that enables sender to reliably form a connection to the receiver and to send a stream of bytes that arrive in correct order to the receiver. UDP is a much simpler connectionless protocol that can be used when features of the TCP are not needed or wanted. TCP/IP model does not have the session and presentation layers found in the OSI-model. The topmost layer, application layer, is on top of the transport layer. It contains a wide variety of different protocols for hypertext and file transfer (HTTP, FTP), electronic mail (SMTP) etc. [1]

2.1.3 Hybrid model

The OSI and TCP/IP models are not suitable for describing modern computer networks alone. The OSI model and its protocols appeared when TCP/IP protocols were already in widespread use. Also the number of layers is somewhat excessive resulting in unnecessary complexity. The TCP/IP reference model on the other hand is based almost completely on existing protocols and does not make distinction between specification and implementation. Therefore it is of little use for designing new network architectures. It also does not define layers below the internet layer and thus makes no distinction between

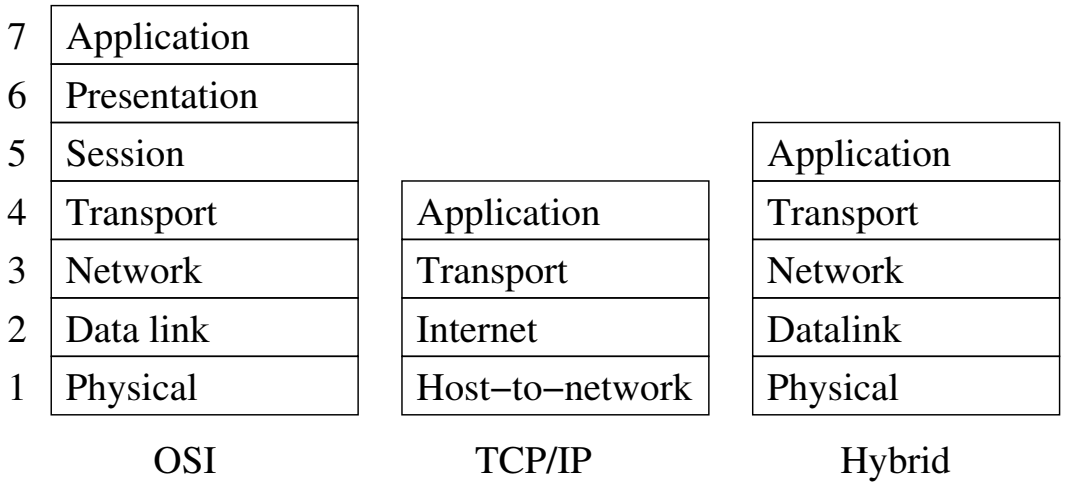


Figure 2.1: Reference models.

the data link and physical layers. As a result a so called hybrid model containing layers 1-4 and 7 from the OSI model suits better for describing existing networks. If IEEE local area network protocols are used as often is the case, the data link layer can be further divided into two parts: Logical Link Control (LLC) and Medium Access Control (MAC). [1]

2.2 Internet Protocol

The Internet is essentially a collection of different types of networks that are interconnected, and the protocol that connects these networks and holds them together is the Internet Protocol. It was developed by Cerf and Kahn during the 1970's and version 4 that is currently used got its RFC specification in 1981. [1][4]

A stream of data that needs to be sent is broken into datagrams with a size of normally 1500 bytes by the transport layer and then these datagrams are given to the IP protocol. The IP protocol's responsibility is to deliver the datagram through the Internet to the destination host. Usually this route consist of numerous intermediate nodes. Some of the nodes might require data to be fragmented further into smaller units that must be

reconstructed later when they reach the destination. Every datagram is routed dynamically as it travels trough the Internet, and because of this, sent datagrams may often travel different routes and arrive out of order. It is up to the IP protocol to reorder and defragment received datagrams before delivering the data to the upper layer. [1]

2.2.1 IPv4 packet

The first field of the packet is the version field which is used to determine the version of the IP protocol. For IPv4 it will always be 4. [1]

The next field, IHL or IP Header Length, tells how many 32 bit words the header contains. It is needed since the length of the IPv4 header is not fixed. [1]

The type of service field can be used for indicating special combinations of speed and reliability requirements. In practice this field is often ignored though. [1]

The last 16 bits of the first word are used by Total length field, which indicates the total length of the packet in bytes (octets), that is the length of the header and data. [1]

Next field is Identification, which is used in determining in which packet arriving fragments belong to. When a packet is fragmented, each fragment of that particular packet has the same identification value. [1]

After this comes one unused bit and two control bits. First one is DF or Do Not Fragment that means the packet must not be fragmented. The second bit is MF or More Fragments which indicates that there are more fragments of this packet still coming after this fragment. [1]

The Fragment offset field tells the location of this fragment in the packet. Elementary fragment unit is 8 bytes, and offset is given as a multiple of it. [1]

The Time to live field is used to limit the lifetime of the packet in the network. The value of the field was originally intended to be decremented with the interval of one second, but in practice it is done on every hop. [1]

The protocol field tells which transport layer protocol handler the payload should be

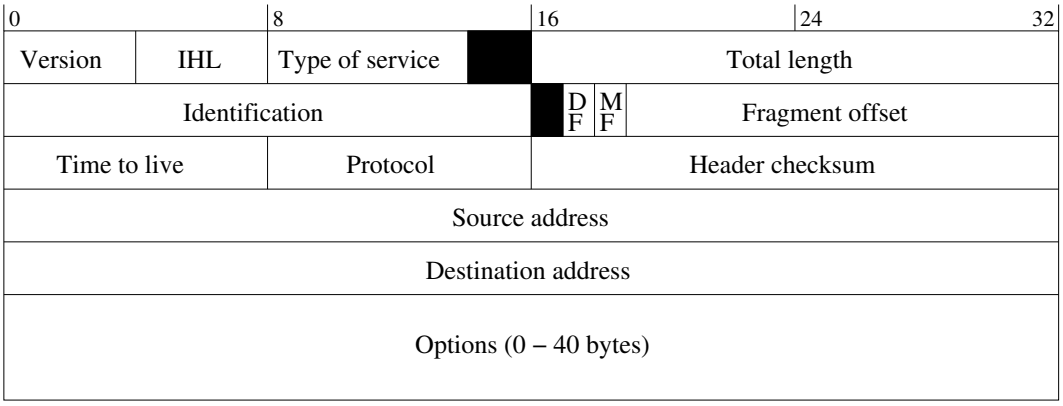


Figure 2.2: The IP version 4 header.

delivered to (for example TCP or UDP). [1]

The Header checksum field contains a checksum for header validation. The algorithm for this is the Internet checksum. Every 16-bit word is added using one's complement arithmetic, and then the one's complement is taken from the sum. The checksum field is zeroed before doing the calculation. [1]

The Options field is a variable length field used for various options concerning security, routing etc. Length of this field is limited to 40 bytes due to 4 bits used for IHL field. [1]

The Source address and Destination address fields contain 32-bit addresses of the source and destination. Every network interface has an unique address. Addresses contain two parts: a network identifier and a host identifier. Lengths of these parts depend on the class of the address. The three first classes are A, B,and C, and the lengths are 8 and 24, 16 and 16, and 24 and 8 bits respectively. Class D is used for multicasting and class E is reserved for future use. Addresses are usually written in dotted decimal notation. The 32-bit address is divided into four 8-bit bytes that are written in decimal and separated with dots. For example address 82E8CA8C is written as 130.232.202.140. [1]

2.2.2 IPv4 versus IPv6

The biggest flaw of the IPv4 protocol is its 32-bit addressing. As the number of computers connected to the Internet is growing at fast pace the 32 bit address space has proven to be too small. This problem has been avoided at short term by introducing technologies such as Classless Inter-Domain Routing (CIDR) [5] and Network Address Translation (NAT) [6]. Work for a new version of the protocol was started in the early 1990's. Other goals were among others to reduce the size of routing tables, simplify the protocol for faster packet processing in high speed networks, improve security features, and improve quality of service especially for real-time data. [7]

The biggest change in IPv6 is the 16 byte, or 128 bit, address space. This enables in theory to for example give $5 * 10^8$ addresses for each of the 6.5 billion people living today.

Another big improvement is a simplified fixed sized header that contains seven fields compared with 14 found from IPv4 header. Optional extension headers with more options can be added after the required header but they can be skipped easily by routers that do not need them. Checksum calculation is not part of the protocol either anymore. Checksums at this level were deemed unnecessary since current networks are fairly reliable and both lower data link layer and upper transport layer have their own checksums. Also a field concerning fragmentation has been left out since a different approach to fragmentation has been taken. Hosts are required to dynamically determine the datagram size in order to avoid need for fragmentation in the first place. If a too large packet is still received it will not be fragmented at the spot but an error message is sent to the sender. This way the sender knows to send smaller packets to that destination in the future. These features simplify and thus speed up processing.

IPv6 also has improved support for authentication and payload encryption which is a major security improvement. [1]

2.2.3 IPv6 packet

The first field of the header is the version field which will always have the value six. Since transition from IPv4 to IPv6 will take a decade or more this field is required for network hardware to recognice the type of the packet. [1]

The traffic class field can be used to give higher priority for real-time transmissions. Field specifies value 0-15 with higher value indicating higher priority. [1]

The flow label field is used when a connection with special properties such as increased bandwidth requirements is set up between two hosts. The field contains an identifier for this connection. The identifier acts as an index to flow tables in routers that contain the information what kind of special service is needed. [1]

The payload length field tells how many bytes will follow the 40 bytes long header field. [1]

The next header field indicates the type of the following extension header if one exists. Extension headers are located between the header and payload. If an extension header does not exist, the field will tell which transport layer protocol is used in the payload. [1]

The hop limit field indicates the life time of the packet. Each node in the network reduces this value by one and when it reaches zero the packet is discarded. This is to prevent packets from travelling in the network forever. [1]

The last two fields are the source and destination address fields. They contain two 128-bit addresses. These addresses are written in eight groups of four hexadecimal digits with colons between the groups. Groups containing zeros may be replaced with two colons and leading zeros in groups can be left out. For example, address

1234:5678:0000:0000:0000:0000:ABCD:00EF

may be written as

1234:5678::ABCD:EF. [1]

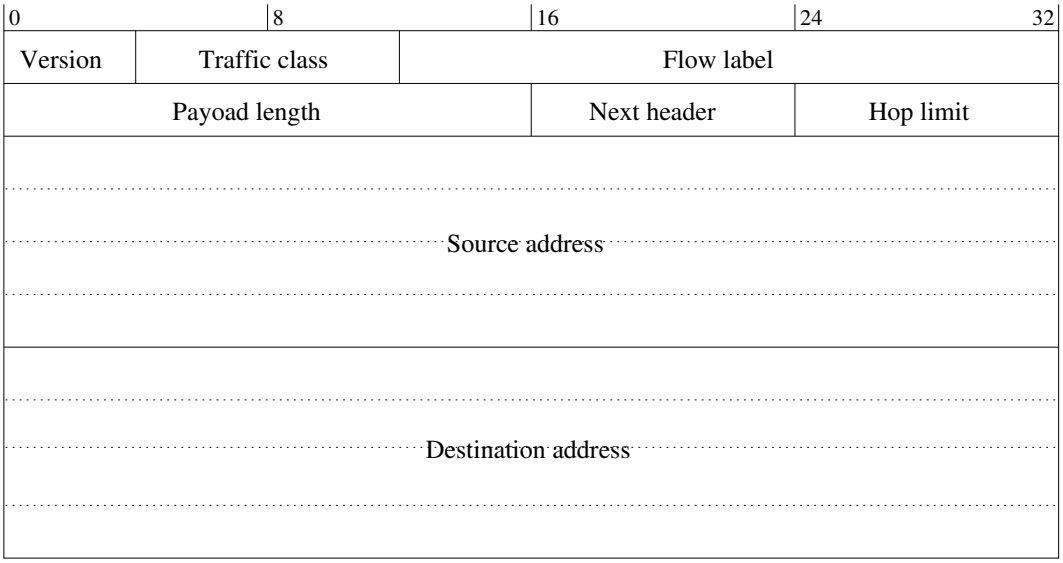


Figure 2.3: The IP version 6 header.

2.3 Transmission Control Protocol

2.3.1 Introduction

Network layer protocols such as IP do not provide reliable end-to-end communication. This is left for transport layer protocols such as the Transmission Control Protocol (TCP) [8]. TCP provides a standardized way to transmit a reliable byte stream over an internetwork that is unreliable and may have different kinds of topologies, delays, bandwidths, packet sizes etc. TCP is used through a TCP entity that can be for example a library procedure, user process or part of the kernel of the operating system. The TCP entity accepts a stream of data from the user, breaks it up into pieces with suitable size (often 1460 bytes in order to fit in an Ethernet frame) and gives them to the IP protocol for delivery. In the receiving end the data stream is reconstructed and can be used by the destination process [1].

A TCP connection is formed between points which are called sockets. Every socket has an identifier consisting of the IP address of the host and a 16-bit number called port. Connections are identified by pairs of socket identifiers and a socket can be an end point

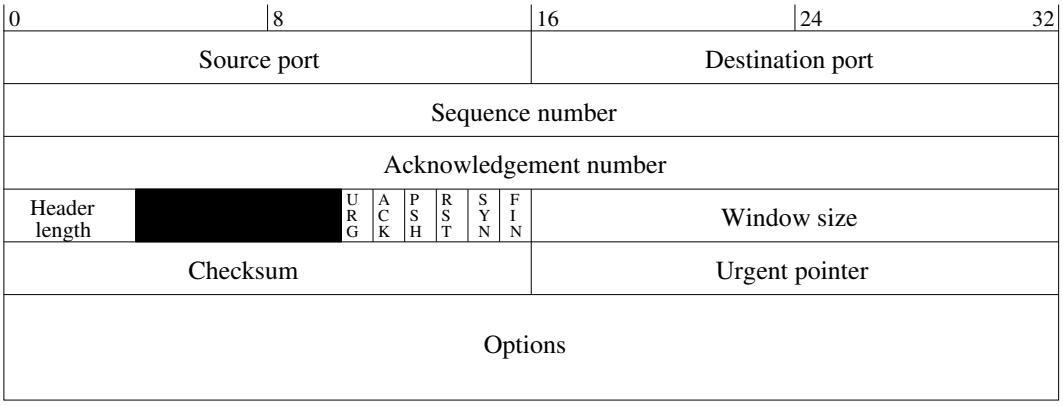


Figure 2.4: The TCP header.

for multiple connections. Port numbers 0-1024 are called well-known ports and are used for certain standard services (such as FTP, HTTP etc.). Other ports can be freely used. [1]

2.3.2 TCP segment

The protocol data unit of the TCP protocol is called segment. A TCP segment consists of a 20-byte header, an optional part and a varying number of data bytes. A size of the segment is decided dynamically by the protocol. The maximum size is limited either by the IP protocol that has a maximum payload size of 65515 bytes or by the maximum payload size of the used by data link layer protocol (often 1500 bytes of Ethernet). [1]

The first 32-bit word of the header consists of source and destination port numbers. Next two words are the sequence number and the acknowledgement number. Conceptually each byte of transmitted data has its own 32 bit sequence number. The sequence number found in the header tells the sequence number of the first byte in the payload. The Acknowledgement number field is used by the receiver to tell which bytes have been received correctly, and contains the next sequence number the receiver expects to receive. [1]

The TCP header length field tells the length of the header in 32-bit words. This field is needed because of the variable length options field. [1]

After six unused bits come six 1-bit flags. When the URG bit is set the Urgent pointer field points to urgent data in the payload. The ACK bit indicates that the Acknowledgement field contains a valid acknowledgement number. The PSH bit indicates that the data has been pushed by the sender and should be delivered to the application immediately without buffering. The RST bit is used to reset a connection in errorneus situation. The SYN bit is used when establishing connection. A connection request has SYN = 1 and ACK = 0, and the reply to this has SYN = 1 and ACK = 1. The FIN bit is used to close connection. [1]

TCP uses a variable-sized sliding window algorithm for flow control. The Window size field indicates how many bytes can be sent starting at the byte acknowledged. [1]

The Options field can be used for defining various extra options that can be used for example to optimize performance. [1]

The Checksum field contains the Internet checksum of the TCP header, payload and parts of IP header. [1]

2.3.3 TCP checksum calculation

TCP does not calculate the checksum only for its own header and payload but also for parts of the IP header. A special pseudoheader containing some fields from the IP header is constructed for this purpose. Since there are two different versions of the IP protocol, and their headers are different, they also require different kinds of pseudoheaders. [1]

The checksum algorithm sums up all 16-bit words of the pseudoheader, the TCP header and the data in one's complement, and finally takes one's complement of the sum. The checksum field in the TCP header is zeroed before calculation and if the length of the data is an odd number of bytes it is padded with zeroes. [1]

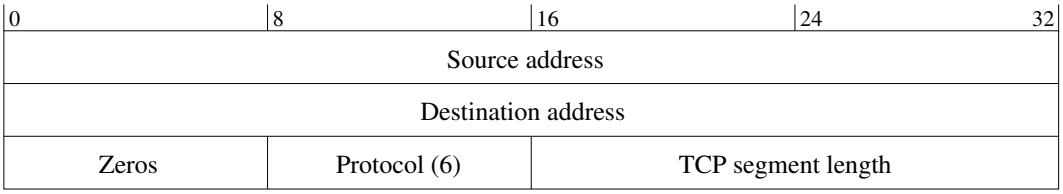


Figure 2.5: The TCP pseudoheader for IP version 4.

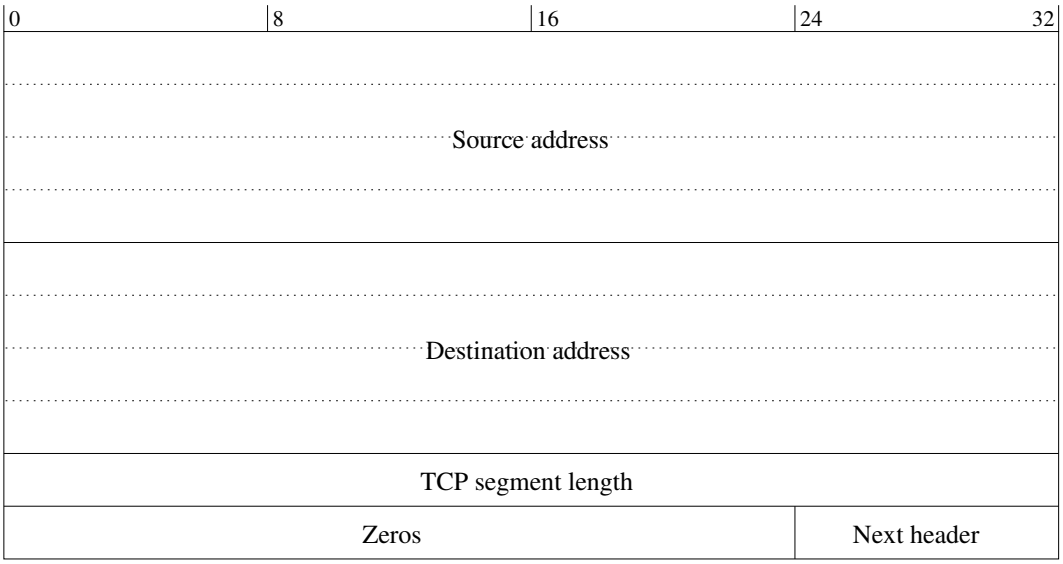


Figure 2.6: The TCP pseudoheader for IP version 6.

Chapter 3

Protocol processors

3.1 Processors

The heart of every digital computer is the processor, often called the central processing unit (CPU). In general processor is attached to some amount of memory (or memories) that is used for storing program code and data, and input/output devices that are used for interfacing with peripheral devices and ultimately the surrounding world. A processor manipulates data stored in the memory according to instructions that are given to it in the form of program code. The interface for the programmer is the Instruction Set Architecture (ISA) which defines a specific set of instructions for controlling the processor, including data types, addressing modes, registers and memory spaces. [9]

Internals of the processor can be divided into datapath and control logic. The datapath (or datapaths) handles the manipulation of data including string manipulation and arithmetic and logical operations etc. The heart of the datapath is the Arithmetic Logical Unit (ALU) where the actual processing takes place. [9]

Control logic fetches the instructions, decodes them, controls the operation of the datapath, and handles transfers in the flow of control such as jumps, branches and subroutine calls. [9]

3.1.1 Processor technologies

There are several types of processors that all have their weaknesses and strengths. General-purpose processors are fully programmable processors that are designed for a wide variety of applications. They usually have a large register file and one or more ALUs that can perform different types of operations. They are highly flexible since they are fully programmable and not designed only for some specific function. Their shortcoming is that when a general-purpose processor is used exclusively for some specific application in for example an embedded system, performance is rarely optimal and power consumption and size can be excessive. [10]

Single purpose processing blocks on the contrary are not programmable at all but perform some specific algorithm entirely on hardware. As a result the architecture is easy to optimize and minimize. Reusability and flexibility of the design is however low since even small alterations to the algorithm require a slow and costly redesign process. [10]

Application specific instruction set processors (ASIP) are a compromise between these two extremities. They offer programmability but can also have a highly optimized datapath for computations needed in some specific application area. This is often achieved by coupling a programmable general purpose processor core with single purpose co-processors. [10]

3.1.2 Parallelism

The most simple processors can be called subscalar. Subscalar processors execute only one instruction at the time in its entirety before starting to process the next one. Needed control logic is very simple to implement but as a serious shortcoming most parts of the processor are idle at any given time leading to very inefficient use of hardware resources.

One relatively simple and widely used way to improve parallelism is pipelining. Instructions are divided into several short subinstructions that can be executed sequentially. As a result multiple instructions can be processed at the same time at different stages of

the pipeline and hardware is used more efficiently. The problem is that consecutive instructions may have dependencies, for example the next instructions may use data that is manipulated by the previous one. This requires additional control circuitry to avoid inconsistencies. [9]

Another more complex way to improve parallelism is to make the processor superscalar. A superscalar processor has many pipelines enabling parallel execution of multiple instructions not only in different stages of the pipeline but at the same stages. This requires a dispatcher unit to be added which fetches many instructions at the time and evaluates whether or not they can be executed in parallel or in some cases out of order. In order to be efficient the dispatcher must employ complex techniques such as branch prediction and speculative execution that may require a substantial amount of extra circuitry. [9]

Other ways to improve parallelism at instruction level are used in architectures such as VLIW and TTA which will be described in detail in the next section. These methods fall into the category of Instruction Level Parallelism (ILP). Parallelism can be improved also at higher levels. ILP does not require programmers' attention but with Thread and Process Level Parallelism (TLP and PLP) programmers must organize their code so that it has multiple threads of execution. On a single core processor these threads can be run virtually parallel by switching the thread periodically, for example on fixed time intervals or when a thread gets blocked by an I/O operation. This creates an illusion of real parallelism. Thread level parallelism can be improved with hardware e.g. by increasing the amount of processor cores [9]. Cores can be identical, each executing a thread, or they may be optimized for different types of tasks. The first approach is utilized e.g. in current (as of 2008) x86 PC processors by Intel and AMD while the latter is found e.g. in the Cell processor by IBM.

3.1.3 Architectures

Many early computer architectures can today be characterized as Complex Instruction Set Computers (CISC). As computers were starting to be used for increasingly complex duties starting from the 1960's it was felt that there is a need for more expressive instructions that could almost be described being high-level. This made programming with assembly language easier and also simplified the development of efficient compilers for high-level programming languages. Since a lot of work could be done with a single instruction this also reduced the size of program code and the number of memory accesses. This was especially beneficial during the 1960's and 1970's when memory was both slow and expensive. [9]

The problem with CISC is that it requires relatively complex circuitry for instruction decoding. A low cost solution is to first translate CISC instructions to simpler microcode and execute that on hardware, but this has a negative impact on performance. It was also observed that complex instructions that were originally intended to speed up execution could sometimes be replaced by a few simple instructions to improve performance. These observations led to development of Reduced Instruction Set Computers starting from the 1980's. A RISC has a set of simple fixed sized instructions that can be efficiently executed on hardware in one machine cycle and are also easy to pipeline. Today the difference between CISC and RISC is in practice blurred, for example Intel's widely used x86 architecture is CISC by definition but the latest compatible processors function internally a lot like a RISC processor would. [9]

As said before, superscalar processors require complex circuitry to evaluate and reorder instructions at run time. VLIW (Very Long Instruction Word) architecture on the other hand takes a different approach. In VLIW architecture the instructions are statically scheduled already at compile time and thus the processor design is simplified. VLIW instruction consist of several operations that are executed in parallel on relatively simple execution units. As a result instruction decoding is easier and more efficient than on

superscalar processors. [11]

TTA (Transport Triggered Architecture) is yet another way to achieve parallelism. In conventional architectures the processor is programmed by specifying operations that as a side effect cause data transfers during which the data is processed. A TTA processor is programmed by specifying data transfers between functional units that as a side effect process the data. To achieve this a TTA processor has only one instruction: data move between two registers. Therefore TTA can be seen as the ultimate RISC processor since the instruction set is reduced to the minimum. The instruction structure is somewhat similar with VLIW but contains move instructions instead of operations. Since data transfers in TTA are visible to the programmer or compiler, there is more room for optimization in comparison to VLIW [11]. The requirement however is that all dependencies must be resolved at compile time [9].

3.1.4 Memory

In modern computers there are many physically different kinds of memories in use. Memory types have different access times, area requirements, power consumption and volatility [9]. There are two fundamental memory architectures that are used: Princeton architecture (also known as von Neumann architecture) and Harvard architecture. In the Princeton architecture data and program code share the same memory space. This simplifies the needed circuitry since the processor has only one connection to the memory. The Harvard architecture separates data and program memory. This requires two connections but may also speed up execution since both instruction and data fetches can be done simultaneously [10].

The fastest available memory type is Synchronous Random Access Memory (SRAM). 1-bit SRAM is usually constructed of six transistors and can be accessed very fast. It is also costly to manufacture due to the area it requires, and because of this it is used only for processors' internal registers and caches. [9]

Read-only Memory (ROM) is significantly more dense than SRAM requiring only one transistor per bit and is still relatively fast but cannot be rewritten. Therefore ROM can be only used for constant data that is fixed on design-time. Various Electronically Erasable and Programmable ROM types also exist today ((E)EPROM, Flash) that introduce some amount of rewritability but are generally slow to rewrite. [9]

A storage element of Dynamic Random Access Memory (DRAM) consists of one capacitor and transistor and can be packed densely. Large DRAM memories suffer however from long access delays of several clock cycles despite the efforts to improve the performance (SDRAM, DDR-SDRAM, Rambus) [9]. DRAM is generally used as data and program memory due to its low price and high capacity.

Various mass storage devices such as hard disc drives and large Flash memories are used for storing large amount of data and programs that are not currently used. They can also be used to extend available memory address space with techniques such as virtual memory. Mass storage devices offer huge storage space but access times are many decades worse than with DRAM.

Due to the physical characteristics of memories it is clear that memory cannot be both fast and large at the same time. Illusion of this can however be created by constructing a hierarchy of memories. This is possible because of temporal and spatial locality that are present in programs. Temporal locality means that the referenced data item or instruction is likely to be referenced soon again. Spatial locality means that after reading an instruction or data word it is probable that the next item after it will be needed soon also. Therefore keeping some selected data in small but fast memories - caches - can speed up execution significantly. Cache policies, algorithms and sizes need to be carefully optimized though in order to be efficient. [9]

3.1.5 IC Technology

Processors are ultimately implemented on an integrated circuit (IC). Integrated circuits are devices that consist of numerous semiconductor components packet densely on one chip. Processor design can be mapped to a physical level IC device in a number of ways.

In application specific integrated circuits (ASICs) the whole circuit is optimized for the design. Transistors are placed to minimize interconnection lengths and size of transistors and wire routing is optimized for signaling. ASICs can be very expensive to design but yield excellent performance, size and power consumption. Unit price is often low making them suitable for high volume products. [10]

Programmable Logic Devices (PLD) have all the circuitry already built but logic blocks can be connected with programmable switches. Simple PLDs consist of arrays of logic gates such as AND and OR. Field Programmable Gate Array (FPGA) devices that can have blocks with complex combinatorial functions have become popular. PLDs are fast and cheap to design and implement but are usually slower and bigger than full and semi-custom chips and have a higher unit cost. Therefore they are often used for prototyping or when low time-to-market is required. [10]

3.2 Protocol processors

3.2.1 Introduction

General purpose processors have been popular in embedded systems since they offer short time-to-market and cost-efficient development cycles. However for networking hardware they are a suboptimal solution since they lack optimized execution units for network processing. All functionality must be implemented on software level which leads to very high clock rate requirements in modern high speed networking. General purpose processors that are fast enough are often expensive, consume too much power and occupy too much space. They also have features such as floating point units that are dead weight

considering network applications. [9]

These problems have often been solved with complex single purpose processors designed for certain network protocols. These processors often offer high performance, reduced power consumption and smaller area, but their design process is demanding and expensive. They also lack flexibility that is often needed in dynamic market segments. [9]

Another solution are application specific instruction-set processors that have a programmable general purpose core along with optimized single purpose co-processors for protocol processing. They offer significantly improved performance and better cost efficiency compared to general purpose processors, and have better flexibility and reusability than single purpose processors. [9]

3.2.2 Characteristics

There are some typical characteristics that have been identified as typical for protocol processors. Pattern matching and replacement in bit strings is often needed especially when analyzing protocol headers. Protocols have also control dominated operation with large finite state machines and nested branch structures. Memory access is irregular due to need for managing tables and buffers of various sizes [12]. Other typical characteristics are high bitrate, need for buffering, boolean evaluations and bitwise manipulation, and often also counter and timer functions and checksum calculations. Some protocols may also need random numbers for for example CSMA/CD. Protocol processing can be done also using only unsigned integer arithmetic resulting in considerably simpler hardware implementations [11]. Protocols at OSI layers 1-3 that require intermediate stations between source and destination devices suit best for application-specific hardware implementations while higher level protocols are often implemented in software [9].

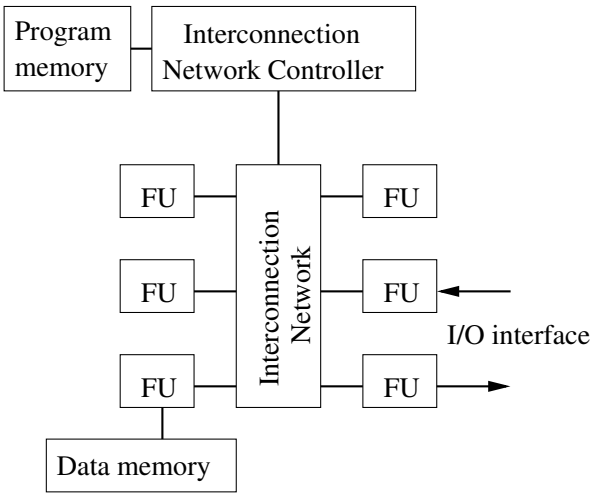


Figure 3.1: General structure of the processor.

3.3 TACO architecture

The TACO architecture is a TTA-based protocol processor architecture developed at the Turku Center for Computer Science. One important goal of the architecture is to take as much advantage of design automation as possible. Modularity and scalability were also pursued. These requirements led to choosing transport triggered architecture as basis for the processor. Functionality of the processor is implemented in functional units (FUs) that perform application specific operations. FUs with suitable functionality can be easily added to the architecture, and because of this they are reusable and a library of them can be constructed. Data transfers between FUs are programmed using assembly language consisting of only move instructions. FUs are connected by interconnection network that can contain configurable number of buses. Every bus can handle one data transfer operation per cycle leading to high level of parallelism. [11]

3.3.1 Functional units

Functional units perform operations to the data. A FU has a set of input and output registers. Input registers can be divided into two subtypes: operand and trigger registers.

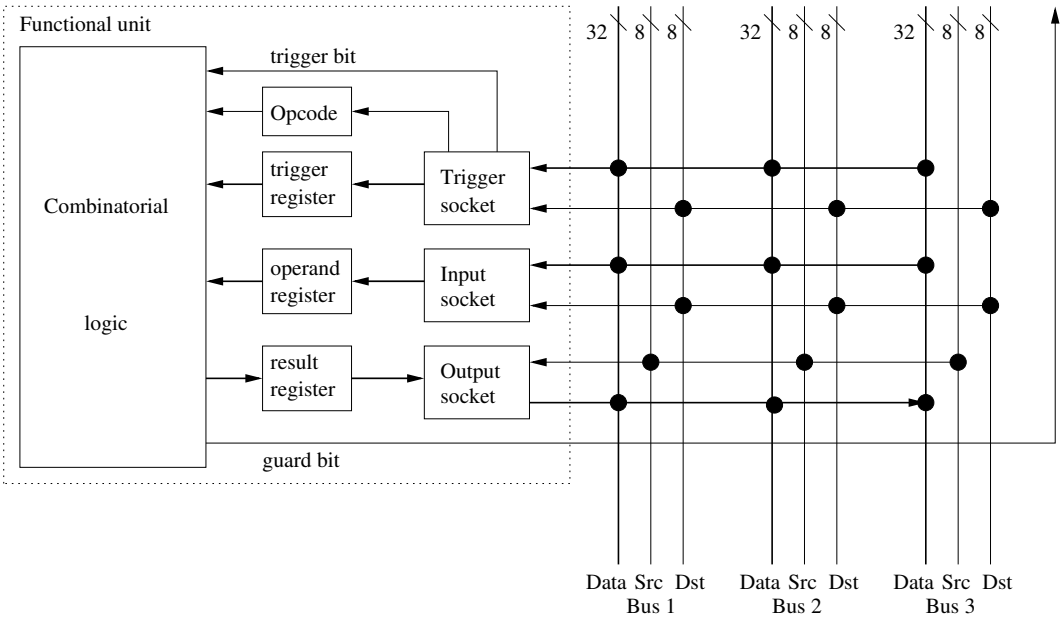


Figure 3.2: General structure of a functional unit with one socket of each type and a guard bit in a processor with three 32/8-bit buses.

Output registers are called result registers. A FU can have multiple operand and result registers but only one trigger register. A data transfer to a trigger register triggers the operation of the FU. A FU can have many operations that are selected with operation codes (details in Socket section). If a FU has operand registers, data has to be transferred to those before triggering the FU. When triggered, a FU performs the selected operation using data found from its trigger register and possible operand registers, and places the result into its result register. A FU can also have a guard bit for signaling some special situations to the network controller. [11]

A FU can perform general purpose functions such as arithmetic or logical operations, but in the TACO architecture FUs perform solely specific operations needed in protocol processing. Operations performed by single FUs include checksum calculation, masking, comparison, counting and memory management among others. [11]

3.3.2 Interconnection network

The interconnection network connects FUs to each other and to the Network controller. Network consists of various number of buses. Each bus has a line for data and source and destination addresses. The data line carries the data while source and destination addresses indicate the source and destination registers. [11]

3.3.3 Sockets

Sockets are modules that connect FUs and the Interconnection network. A socket can be connected to all or some of the buses. Like FU registers, sockets are also divided to three categories: input, output and trigger sockets. Each socket is connected to one register of an FU and has an unique identifier. [11]

Input sockets are connected to the data and destination address lines. The socket compares the value of the destination address line, and if it matches its own identifier content of the data line is read and passed to the operand register of the FU. [11]

Output sockets read the value of the source address line, and if it matches the identifier contents of the result register is written to the data bus. [11]

A trigger socket functions like an input socket except that it also has a one bit signal line to the FU that is set when identifiers match. Setting the bit triggers the operation of the FU. If the FU has more than one operations, its trigger socket has the same amount of identifiers. These extra identifiers are used to separate operation from each others. Information about which operation is triggered is stored in the operation code register. [11]

3.3.4 Interconnection network controller

The Interconnection network controller controls the operation of the processor. It has an own program memory where the program code is stored. [11]

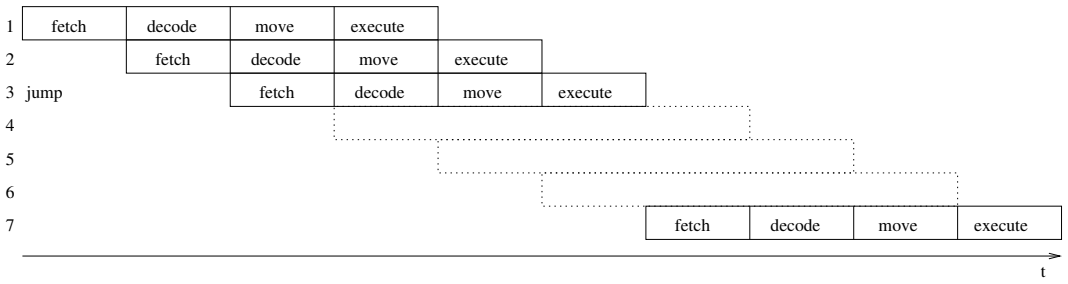


Figure 3.3: TACO processor pipeline and its operation during jumps.

One instruction word consists of one subinstruction for every bus in the processor. Subinstruction has two address fields, source and destination, that are placed on corresponding lines of the bus. The addresses trigger a data transfer between two registers. A subinstruction also has a guard field, that contains identification for guard expressions that are used for conditional execution. In addition to subinstructions, the instruction word has also a field that enables dispatching of immediate values. This field specifies the subinstruction(s) that contains an immediate value. When for example the first bit of the immediate field is set, the source address field in the first subinstruction contains an immediate value that will be placed on the data line of the bus instead of source address. [11]

The Network controller fetches instructions, maintains the program counter, evaluates guard expressions and dispatches subinstructions and immediates onto buses. The operation of the processor is pipelined and consists of four stages. [11]

Fetch In the fetch stage Network controller fetches instruction from program memory. [11]

Decode The decode stage divides into to substages. At first the Network controller places source and destination identifiers on the address bus. After this sockets read the contents of the bus, and if there is a match between a socket's own identifier and the identifier on the bus a data move on this bus is scheduled. [11]

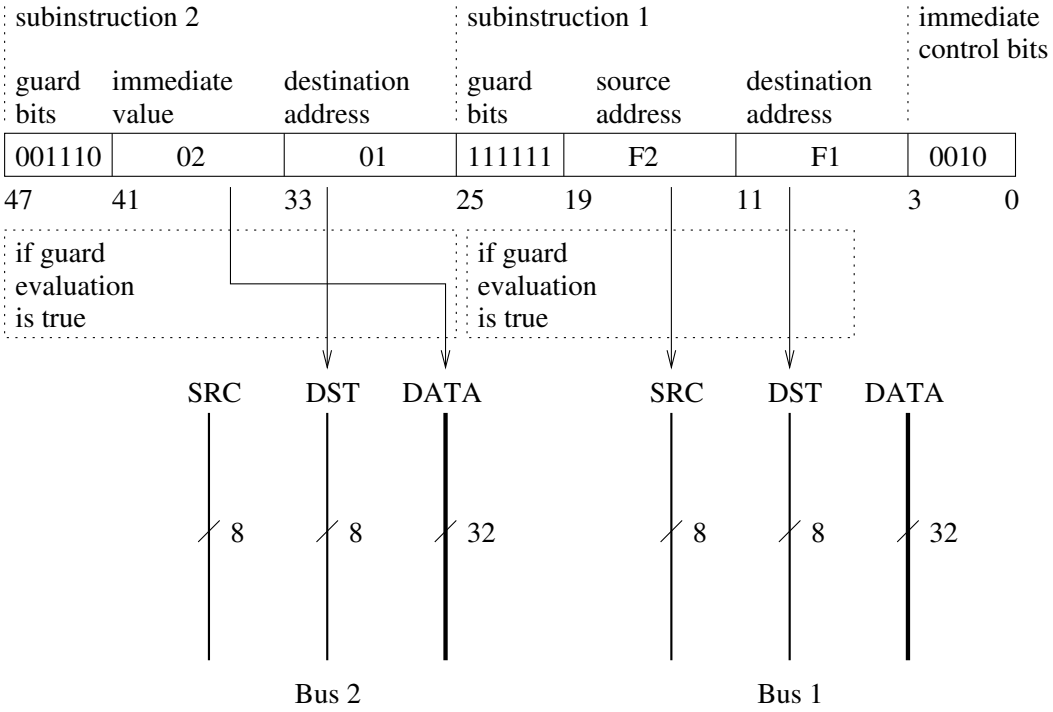


Figure 3.4: Simplified example of instruction decoding for a TACO processor with two buses, 8-bit addresses and 32-bit data. If guard evaluations return true, data is dispatched on buses. Subinstruction 2 contains immediate value that is written on the data bus instead of source address.

Move In the move stage the actual data move between the output socket and the input or trigger socket of two functional units takes place on the data bus. [11]

Execute In the execute stage functional units execute their operations and write the result in the result register. [11]

All instruction scheduling is done by the programmer or the compiler resulting in very simple instruction decoding. As a result the structure of the Network controller is relatively simple. [11]

The program counter is implemented as a socket that is connected to the Network controller. It has three operation codes that make jumps possible. With these operations the program counter can be loaded with the given value or a value can be added or subtracted

from it. When the program counter is updated the Network controller must wait a period of three cycles to allow pipelined instructions to complete. [11]

Chapter 4

SystemC

4.1 Introduction

SystemC is a C++ based hardware modeling language. The ever increasing level of complexity if digital systems has created the need for higher abstraction level tools than traditional hardware description languages (HDL) such as VHDL and Verilog for system specification, modelling, and verification. High level programming languages such as C++ offer a suitable level of abstraction but lack structures that are needed for accurate modeling of hardware. SystemC is intended to close this gap by introducing HDL structures and concurrency modeling to C++. This enables the use of one language at various levels of abstraction in projects that require hardware/software integration and codesign [13]. SystemC is not only a language but it also includes a simulation kernel on which the code is executed. This enables accurate modeling of concurrency that is typical for hardware instances, as opposed to software that is inherently sequential [14].

4.2 Features

SystemC introduces several hardware oriented data types in addition to those found in standard C/C++ and structural components found from traditional HDLs such as modules,

```
sc_bit binary = '0';
sc_logic high_impedance = 'Z';
sc_bv<8> bitvector = "11110000"; // 8-bit vector
sc_uint<32> integer = 12345; // 32-bit unsigned integer
sc_bigint<128> = -234124; // 128-bit signed integer
```

Figure 4.1: Examples of SystemC data types.

ports and signals.

4.2.1 Data types

SystemC provides several data types in addition to native C/C++ data types. For binary representation there are two types: sc_bit and sc_logic. Sc_bit is a two valued logic type which can have values true ('1') and false ('0'). Sc_logic adds two additional values, high impedance ('Z') and unknown ('X'), that are needed e.g. for accurate modeling of buses with multiple drivers. There are also vector types with adjustable width available of both types, sc_bv and sc_lv. [14]

In addition to binary types, SystemC has signed and unsigned integer types and fixed point types. The length of the standard C++ integer is not fixed but depends on the computer architecture on which the code is compiled. When describing hardware it is however very important to be able to specify the length exactly. Therefore SystemC integer types have fixed precision that is defined explicitly. The length of integers is given as a template parameter when variable is declared. Types sc_int (signed) and sc_uint (unsigned) can have width of 1 to 64 bits. Sc_bigint and sc_biguint are used for larger integers. SystemC integer types also include several operators and functions that enable hardware oriented operations. For example several binary operations and bit ranges selections can be performed easily. [14]

```
SC_MODULE(ExampleModule1){
  ...
};

class ExampleModule2:  public sc_module {
  ...
};
```

Figure 4.2: Two ways to declare a SystemC module.

4.2.2 Module

Modules are the basic building blocks of a system. In SystemC modules correspond largely to classes in object oriented software design. In fact a SystemC module is a C++ class that inherits library class sc_module. A module's internal implementation can be encapsulated and a public interface offered through which the module is connected to the surrounding system. The nature of the interface depends on the abstraction level of the model. SystemC allows register transfer level (RTL) modeling using ports and signals but also transaction level modeling (TLM) which operates on a higher level of abstraction, is possible. [14]

Modules can be declared two ways: using the SystemC keyword SC_MODULE or explicitly inheriting class sc_module which is the parent class of every module, in normal C++ style. [14]

A constructor is used to create and initialize the module. There are two different ways to declare the constructor. The SC_CTOR keyword may be used, or C++ style can be used in which case the keyword SC_HAS_PROCESS(modulename) must be present in the code. SC_CTOR allows only one constructor parameter to be passed which is also obligatory for every module constructor: module name. In SystemC every module must have a unique name. If the user does not want or cannot provide unique names SystemC offers a name generator function. [14]

```
SC_CTOR(Adder){
  // constructor body
}

SC_HAS_PROCESS(Adder);
Adder(const sc_module_name& name):sc_module(name){
  // constructor body
}
```

Figure 4.3: Two ways for declaring a module constructor.

4.2.3 Processes

Processes are the part of the module that provide the functionality. Processes are sensitive to certain signals and are executed when an event happens on one of these signals. A typical event type is a change of value. The process body contains statements that are executed sequentially until the end is reached or process is suspended (e.g. by calling the wait() function) [14]. Processes in a module can run virtually in parallel thereby enabling the modeling of parallelism.

In simplicity, processes are C++ functions that are registered with the SystemC kernel, and can therefore be invoked when necessary and be executed concurrently with other processes. There is no hierarchy between processes but they are all equal and cannot invoke each other. Modules can however have normal C++ functions that are not processes and that can be called by processes. [14]

There are three types of processes in SystemC. Method processes execute their body sequentially from start to end and when complete, control is returned to the simulation kernel. Therefore a method process cannot have e.g. infinite loops but the execution must always terminate. [14]

Thread processes execute until they are suspended by calling wait() within the process body. Execution is resumed from that point the next time process is triggered. [14]

Clocked thread processes are like thread processes but they are sensitive to exactly one signal, the clock signal that is given to it when the process is declared. The clock signal triggers the process on every clock edge, and execution is suspended by calling either wait() or wait_until() function. Sensitivity to the positive or the negative clock edge can be defined. The wait_until() function can be used to halt the execution of the process until a certain condition becomes true. These features make clocked threads especially suitable for describing finite state machines in a hardware oriented way [14]. Unlike methods, threads and clocked threads require an own execution stack in the simulation kernel. This makes context switching slightly heavier and introduces some overhead in comparison to method processes [15].

4.2.4 Ports and channels

In SystemC ports are the interface between a module and the rest of the system. Ports are connected to other ports via channels.

There are three types of ports. Input and output ports are one directional. Input ports carry data to the module and can only be read and output port transfers data out and can therefore be only written. Inout ports allow two-directional transfer. Ports are accessed with two self-explanatory functions, read() and write() [14]. SystemC allows creation of ports only within modules [15].

Channels are the means to interconnect ports. The simplest channel type is signal that models a wire in the physical circuit. Other more advanced channel types include buffers, FIFOs and semaphores. In SystemC there are many ways to bind the ports to a channel. Earlier versions supported positional binding, where a channels could be bound to ports in the order the ports were declared in the module using operator $<<$. This feature has been deprecated since SystemC 2.x however. Currently the two best ways are to either use operator() or in a more explicit manner use function bind(). The SystemC simulation kernel requires every port to be bound to exactly one channel (or in some cases directly

```
SC_MODULE(Example){
  void exampleProc1(){
    cout << "This is a method process" << endl;
  }
  void exampleProc2(){
    while(true){
      cout << "This is a thread process" << endl;
      wait();
    }
  }
  void exampleProc3(){
    while(true){
      cout << "This is a clocked thread process" << endl;
      wait();
    }
  }
  SC_HAS_PROCESS(Example);
  Example(const sc_module_name& name, sc_clock& clk):  sc_module(name){
    SC_METHOD(exampleProc1);
    sensitive << clk.pos();

    SC_THREAD(exampleProc2);
    sensitive << clk.pos();

    SC_CTHREAD(exampleProc3, clk.pos());
  }

};
```

Figure 4.4: Example of three different process types.

```
SC_MODULE(PortExample){
  public:
  sc_in<sc_bit> input; // input port
  sc_out<sc_bit> output; // output port
  // constructor, variables, processes, functions etc.
};

// code in higher level module constructor or main function:
// creation of two instances of PortExample module:
PortExample one("Ex1"), two("Ex2");
// creation of two signals for interconnecting these modules
sc_signal<sc_bit> sig1, sig2;
// signal binding using operator() and function bind():
one.input(sig1); one.output(sig2);
two.input.bind(sig2); two.output.bind(sig1);
```

Figure 4.5: Example of port binding.

to another port) [14]. Channels can be created only within modules or the main function [15]. This means that port binding must happen either within module constructors or in the main function.

The port type must match the type of the channel to which it is bound, for example a port of type sc_uint cannot be bound to a channel of type sc_bv. When a port is read, the value of the channel connected to it is returned. When a port is written to, value is assigned to the channel. To solve timing problems that might occur during simulation, new channel values are assigned only after the writing process has stopped executing [14]. As a result, when the writing and reading processes are synchronized by the same clock and are therefore executing concurrently, a new channel value cannot be read until the next clock cycle.

Signals can have multiple drivers (i.e. output ports connected to it). To model collisions accurately, SystemC 2.x requires resolved four-valued logic to be used as a data

type for the ports and signals instead of higher level data types. [14]

4.2.5 Simulation

When a SystemC description of the system is ready, it can be simulated. Prior to the simulation every module must be constructed properly and ports must be bound. Simulation is started with the sc_start() function that takes the simulation time as a parameter or runs indefinitely if none is given. Simulation can be stopped with function sc_stop() and during the simulation the elapsed simulation time can be queried with the function sc_simulation_time(). Waveforms of selected signals can be traced during the simulation, and results are stored in a file in VCD, WIF or ISDB format and are viewable with any waveform viewer supporting some of these file formats. [14]

The execution of SystemC code consists of two parts: elaboration and simulation. During the elaboration module hierarchy is created and integrity of modules and connections of ports is checked [16]. This includes creation of modules and their ports and signals. After the elaboration this structure is fixed, meaning that additional modules, ports or signals cannot be created dynamically during the simulation as this would make little sense from hardware point of view. If the system passes elaboration, the actual simulation is started under the control of the scheduler of the simulation kernel. SystemC has a non-preemptive scheduler that schedules the execution of processes based on events created by e.g. changes in the signal values. After the simulation module hierarchy is automatically destructed [15].

4.3 Hardware synthesis

Efforts have been made to standardize a synthesizable subset of SystemC to ensure compatibility of different synthesis tools. The Subset is currently however at draft state and much of the details are dependent on the tool that is used.

Improved expressiveness of SystemC compared with traditional hardware description languages could speed up hardware design significantly by simplifying the design flow provided that efficient synthesis is possible. The synthesizable subset however has many severe restrictions in comparison to normal SystemC/C++.

One of the biggest limitations is that the member variables and functions cannot be referenced from outside the module [17]. Intermodule communication must be implemented exclusively using ports and signals in a similar manner as with traditional hardware description languages. In effect this means that the modules must be fully initialized in their constructors and must communicate strictly at register transfer level with other modules.

There are also limitations for data types. Most integer and bit types of C++ and SystemC are supported, but floating point types are not. Also pointers have restrictions that have a significant impact. Pointers are allowed only in cases where they point to a statically determinable object. For example pointer arithmetics and arrays of pointers are not allowed [17]. Because of this most of C/C++ standard libraries are not supported, containers among the most important ones. The only synthesizable data structure is a normal C array of supported data type. Because the C array is a very restricted, unsafe and low level structure this imposes serious limitations that are discussed later.

There are many details which the draft for synthesizable subset does not address. For example inheritance is a significant issue that is largely left open. The tool used for synthesis tests was Celoxica Agility SystemC Compiler, and many additional restrictions were found from it. E.g. inheriting processes does not work and virtual inheritance is not supported. In practice inheritance works only with data members making an object oriented design approach practically impossible.

Chapter 5

TACO simulation model

5.1 Introduction

To enable rapid simulations, evaluation and design space exploration, a SystemC based simulation environment was constructed for the TACO processor. SystemC also makes it easy to construct libraries of functional units. Designers can easily construct models for a certain protocol processing application from a set of readily available functional units and possibly implement some new ones, write program code for the processor instance, simulate it, and evaluate results for both the hardware and software. Design tools used are also free, for example C++ compilers such as GCC (GNU Compiler Collection) are available under open license.

5.2 Previous version

The previous version of the TACO simulation model was developed for SystemC 1.0. It was hoped that object oriented techniques could have been employed, but due to limitations in that version of SystemC this was not fully achieved. [11]

The model contains SystemC descriptions for functional units, sockets, the interconnection network and the interconnection network controller. It also has certain elements

that are used for automating model constructions as much as possible. Program code for the modeled processor can be written with an assembly language which is compiled to hexadecimal form into an ASCII file with an assembler/compiler. The simulation model loads the program code from this file automatically when it is run. [11]

Abstraction level of the model is heterogeneous. Communication in the interconnection network is modeled at RTL level while functionality of the functional units can be given with C++. The functional units have a very similar interface towards the interconnection network. Therefore much of this code can be gathered to a parent class making the code more manageable which simplifies the implementation of new functional units. [11]

During the elaboration phase module hierarchy is created. Sockets are created and addresses are distributed to the sockets automatically, and functional units are connected to the buses automatically. After the elaboration phase module hierarchy is fixed and simulation is started [11]. Since much of the variability of the TACO architecture comes from an undisclosed number of functional units, dynamic creation of FUs and their sockets is an essential part of the simulation model.

A simulation model is constructed by creating the basic elements of the model and desired functional units in the main function, and connecting functional units to buses. When a FU is constructed, the correct amount of sockets and all the needed signals for it are automatically created partially in the constructor and partially in the SocketManager class. Buses have functions that take a pointer to a FU as a parameter and automatically connect a FU's sockets to the right lines of the bus. After this the code is compiled and simulation is automatically started provided that everything was done in the main function correctly. [11]

Due to the immaturity of SystemC 1.0 several problems were encountered during the development of the previous model which required suboptimal solutions to be corrected. Use of inheritance especially in the functional units was required to enable extendability

and reusablity. This however was difficult to achieve, and required a threefold class hierarchy to be created for the functional units including use of class templates. Also common C++ practice of separating declarations and implementations to different .h and .cpp files was not possible. [11]

5.3 Updated version

Due to advancements in SystemC a need for an updated TACO simulation model was felt. The previous simulation model also did not simulate the four stage pipeline that was introduced to the architecture later. Also the possibility to synthesize a simulation model was considered an interesting topic. A synthesizable VHDL model for the TACO architecture was created earlier and comparing it to a synthesizable SystemC model would be a valid research topic.

After thorough examination of the model it was however apparent that due to vast changes in SystemC and certain problematic features in the model it would not be possible to bring it up to date without major overhaul and rewriting large parts of the model.

One big problem were the signals. SystemC 1.0 did not model traffic in the signals as accurately as SystemC 2.2 and did not enforce collisions. A signal in SystemC 2.2 must be of a type sc_logic if the signal has multiple drivers, therefore forcing the developer to consider collisions. In the old model signals were of type sc_uint and essentially used like shared variables where the last written value can be read and overwritten if desired. This simplified the model and enabled higher level programming, but was totally incompatible with SystemC 2.2 and not accurate enough when considering synthesis.

Other big problems were dynamic creation of signals and ports. The sequence of creating and connecting the ports and needed signals of sockets to buses and to FUs was complex, and during it signals and ports were created outside module constructors. Since this is enforced more strictly by SystemC 2.2 it caused errors and some of the ports were

not properly bound. This complexity also hindered creation of new FUs. The base class only included code for creating and connecting one of each type of sockets. However many FUs have more input and output sockets than just one, and the code for creating and connecting these extra sockets had to be placed in the child class. This made the code bloated with functions that conceptually should be in the parent class.

There were also smaller problems. The synthesizable subset of SystemC does not allow e.g. pointer arithmetic and dynamic memory allocation which are used by classes of the C++ standard library. This is especially problematic since large parts of the functionality of the model was implemented using STL container classes that rely heavily on these features. Another problem was that the synthesizable subset allows intermodule communication only via ports. Member functions and variables of a module cannot be accessed outside the scope of that module. Some parts of the model were implemented in a normal high level C++ manner which would have been very difficult to synthesize later.

5.3.1 Design principles

To remedy the problems faced with the old model and to overcome the limitations of SystemC and especially its synthesizable subset, removal of all unnecessary complexity was taken as a design principle. Also recommended SystemC 2.x coding style was used and synthesizable subset was followed whenever feasible rather than higher level software engineering oriented approach. In order to keep the design clear from hardware oriented point of view, the abstraction level was chosen to follow system level block structure and to introduce only few complex software structures to the model. In addition, to make the model even more usable, implementation of new functional units and construction of a new simulation model instance was simplified even further by automating as much of the module initialization as possible.

In effect this meant that all the necessary initializations and submodule constructions had to be done in the module constructors, and necessary information be given as param-

eters. It soon became obvious that choice had to be made between an elegant and user friendly model and a synthesizable model. The first one was chosen and as a result some non-synthesizable features were added to make the model more usable. They were however implemented in a way that their removal and replacement is easy. To improve reuse, certain features such as width of the buses and variables can be parametrized in a global level definition file. Variable widths were also considered when implementing all parts of the model, making transition from e.g. a 32-bit width to 64 bits only a matter of changing the value of one macro and recompiling the code.

Coding style was also changed a little. Due to limitations of SystemC 1.0 typical C/C++ practice of separation of declaration and definition in separate .h and .cpp files was not possible. Since SystemC 2.x no longer suffers from this the practice was taken into use. There is one limitation however: the module constructor still has to be written in the header file. Another decision concerning style was not to use SystemC macros such as SC_MODULE and SC_CTOR but to replace them with standard C++ code. Readability of standard C++ code is better, some limitations can be diverted (e.g. number of constructor parameters), and it also e.g. simplifies the use of C++ compliant code documentation tools. Code documentation was formatted compatible with the Doxygen tool that automatically generates HTML documentation of the code.

5.3.2 Structure

Basic class structure was largely kept the same despite the fact that the actual implementation changed a lot due to application of new design principles and introduction of new features.

The class FunctionalUnit is an abstract parent class of every functional unit. It creates and configures a parametrizable number of different types of sockets, and contains one process. The checkTrigger process is a clocked thread process that executes on every clock cycle and checks if the functional unit is triggered. If it is, it executes the triggerOp-

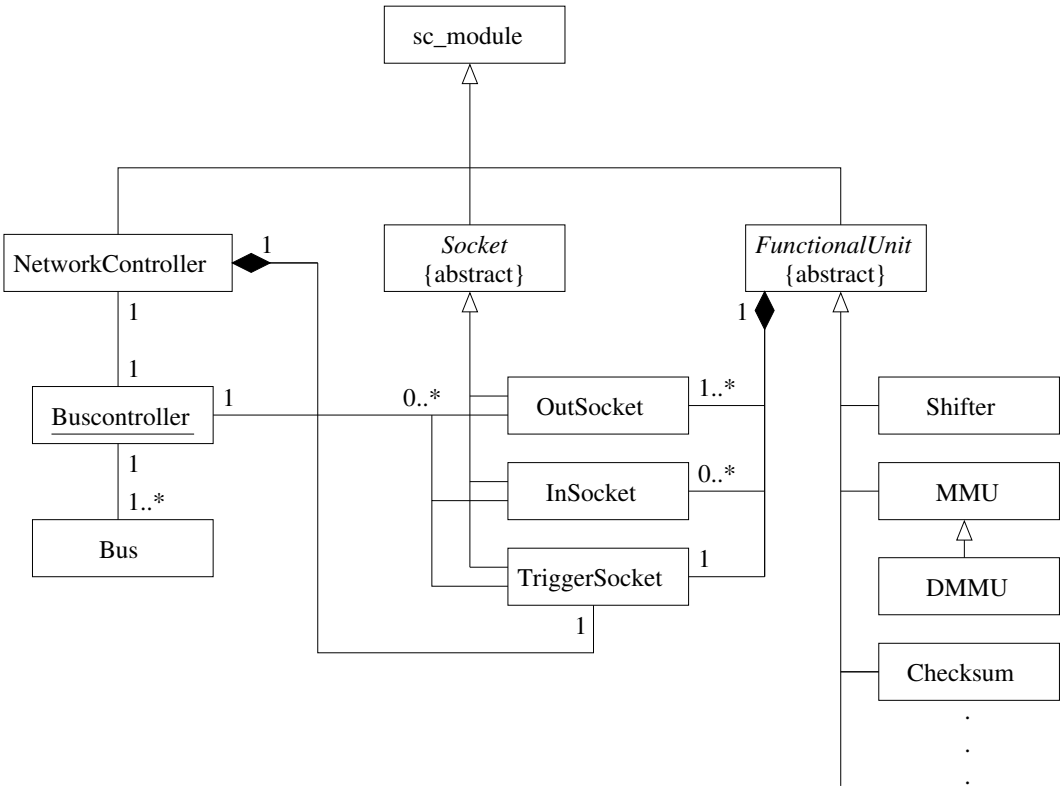


Figure 5.1: UML class diagram of the new version of simulation model.

eration function. This function is a virtual function and the specific implementation for it is given in the inheriting class. Since checkTrigger is a thread process in contrast to a method process, it is possible to add wait-statements into the implementation of triggerOperation if needed (e.g. a FU has a non-trivial execution unit that needs several clock cycles to complete execution). In principle implementing a specific functional unit is very simple task, it only includes giving parameters (mainly the number of input and output sockets) for the constructor of the parent class and giving an implementation for function triggerOperation. Also existing functional units implemented for the old version of the simulation model are relatively easy to convert to the new model, it just requires the functional description to be copied inside the triggerOperation function. Certain functional units may also have guard bits that are connected to the NetworkController. Since they are very implementation specific, the simulation model does not provide any general facilities for them. Implementation of FUs is discussed in detail later.

The Socket class is the parent class of the three types of sockets in the processor architecture; input, output and trigger sockets. Sockets have one process: decodeId. DecodeId is a method process that is sensitive to the clock. A suitable implementation for this is given in child classes. Contents of the address bus (src in output socket, dst in input and trigger socket) are inspected. If a matching address is found the socket reacts, e.g. an output socket writes its contents to the data bus. A socket is connected to the buses in the child classes since they do not have a common interface towards the buses. OutputSocket has output ports that are connected to the data buses, and input ports that are connected to the source address buses. TriggerSocket and InputSocket have input ports that are connected to data buses and destination address buses. In addition, TriggerSocket has a one bit output port that is connected to the functional unit and used to trigger it. Sockets are always a solid part of some FunctionalUnit instance. There is one exception though, NetworkController also has one TriggerSocket that is connected to the program counter and used for programmed jumps.

The NetworkController is the third and last elemental component of the system. The program memory is modeled as an array of instruction words. When an instance is created, contents of the program memory are loaded from an ASCII text file containing program code created by a separate compiler tool (discussed in detail later). NetworkController is connected to all the buses and in addition some of the FUs have guard bits that are also connected to the NetworkController. Bus configurations are automatic but guard signals must be connected manually in the top level file. NetworkController also has one TriggerSocket that is used for updating program counter. It has two method processes that are sensitive to the positive edge of the clock. The updatePc process checks the trigger bit coming from the trigger socket and loads a new program counter value if one is present. The other process if fetch. It fetches a new instruction word from the memory on every clock cycle unless a programmed jump has occurred in which case it waits for three cycles. Instruction words are decoded, guard expressions are evaluated,

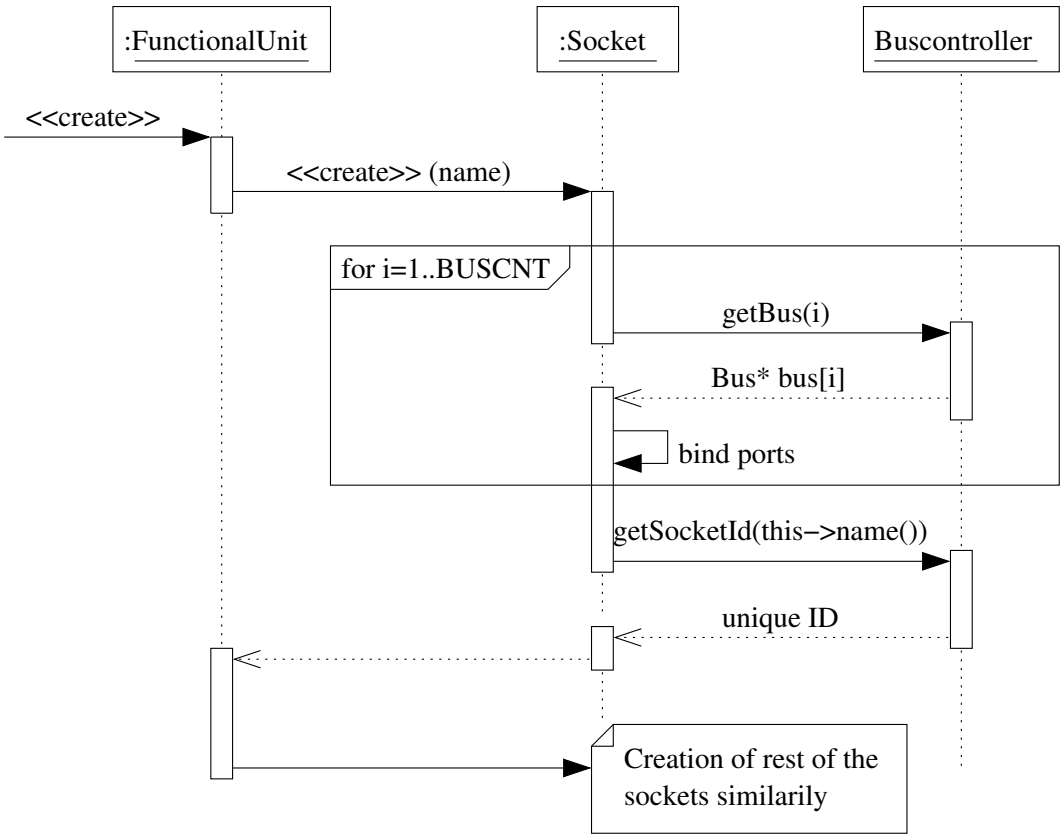


Figure 5.2: UML sequence diagram of the creation and initialization of a functional unit using Buscontroller.

and subinstructions are dispatched to the buses. Finally it is checked whether the end of the program memory is reached in which case execution is halted. Guard evaluation is separated into an own function to clarify the code and since the implementation of different guard expressions may change from processor instance to another.

The two remaining classes, Buscontroller and Bus, are not hardware modules and exist to make the simulation model easier to configure by allowing automatic bus binding. Bus is a simple class that simply encapsulates the “wiring” or signals of one bus, i.e. source and destination address lines and data line. Buscontroller is a static class that manages Bus objects. Buscontroller is called by constructors of the modules. It returns a pointer to a Bus object, and the ports of the module are bound to it in the module constructor. Buscontroller also attaches all signals to a tracer object that creates a VCD trace file of

bus traffic when a simulation is run. In addition, Buscontroller is used to give unique ID:s or addresses to the sockets. Because Buscontroller is not synthesizable, its removal from the system was done simple. When the Buscontroller is missing bus signals need to be manually created in a top level file and connected to the ports of the functional units (or sockets to be exact) and network controller. Also socket addresses need to be given manually as a parameter to constructors of each functional unit. This is a trivial task but makes simulation model construction significantly slower and more prone for errors, and is therefore feasible only when the model is to be synthesized.

5.4 Implementing a new functional unit

When the processor architecture is used on a new application area it may be necessary to implement new functional units. This is a relatively straight-forward task. A trivial case, a new FU must inherit from class FunctionalUnit and give implementation for the triggerOperation function. An example of a simple FU is seen in fig 5.3. The constructor takes five parameters. The first one is the name that has to be a unique string of characters. Reference to the clock signal is also needed. The rest of the parameters are pointers to arrays of integers that are the addresses for each type of sockets. They can be left blank in which case they are initialized with a constant array that contains only a zero. The reason for using this array with zero instead of initializing the pointer to zero is that the synthesizable subset does not allow a pointer to be checked for the value zero.

These parameters are passed to the constructor of the parent class, as well as integer values that define how many input and output sockets need to be created and how many operation codes the trigger socket has. In this example the FU has two input sockets and one output socket and two operation codes. When a FU has more than one operation code it may be useful to give them specific names rather than let the simulation model generate generic names. It makes the virtual assembly code more readable since these names will

```
ExampleFU(sc_module_name name,
          sc_clock &c,
          sc_uint<ADDRESSWIDTH>* opId = zero,
          sc_uint<ADDRESSWIDTH>* resId = zero,
          sc_uint<ADDRESSWIDTH>* trigIds = zero
  ) :  FunctionalUnit(name, c, 2, 1, 2, opId,
          resId, trigIds, opNames)
  { // constructor body }

void triggerOperation(){
  resultReg[0].write(operandReg[1].read());
}
```

Figure 5.3: Example of a constructor and triggerOperation function of a simple functional unit.

be used as mnemonics for these registers. In that case a pointer to an array of character strings can be given as a parameter to the parent class constructor. Care must be taken though that the pointer does not point to a freed address space. This can be achieved by for example introducing a constant and static member variable containing these strings.

Simple FUs rarely need any initializations for themselves and the constructor body can be left empty.

The functionality of the FU is given in the triggerOperation() function. In the example case the FU reads the contents of the second operand register (i.e. contents of the input socket) and writes it to the first and only output. Input and output registers are found in arrays created by the constructor of the parent class. Some care must be taken not to read or write outside their boundaries since C/C++ arrays are not protected against this and doing so leads to errors that may be hard to debug.

In some cases however it might also be feasible to extend some existing FU. In the FUs found from the current library there is one such case. The Data Memory Management

Unit (DMMU) shares the same interface towards the interconnection network with the normal MMU but is also directly connected to two other FUs: InputFU and OutputFU that connect the processor to the surrounding world through e.g. a network interface. However, in general FUs rarely have such features.

5.5 Writing software for the processor

The software for the processor is written with a virtual assembly language just as with the older version of the simulation model. Assembly code for the processor consists only of data transfers between registers (or sockets to be exact) and guard expressions that are used for conditional execution. Names of the registers and the guard expressions are always specific for a certain processor architecture instance. When the model is constructed, compiled and run once it will create a file named socks.txt. This file contains the names of every socket and their corresponding addresses. The names found from this file are used in the assembly code. Names are generated by the model by taking a part of the name of a FU and adding a suitable prefix to it. For example the first output socket of a Counter instance “C1” will get the name “RC1”. The guard expressions can be defined freely and must be manually placed to the file guards.txt. This file will contain the assembly notation for the expression and the index that points to the implementation found from the NetworkController.

Once the code is written with assembly notation that corresponds to mnemonics found from socks.txt and guards.txt in the code is compiled or assembled to binary form. This is done with a separate compiler tool. The compiler uses the same global definition file with the simulation model and can therefore assemble the code to instruction words of right dimensions. The compiler program must be recompiled though every time the definitions are changed. When it is executed it creates a file code.bin which is an ASCII file that contains the program code in binary form. This code file is then placed in the same

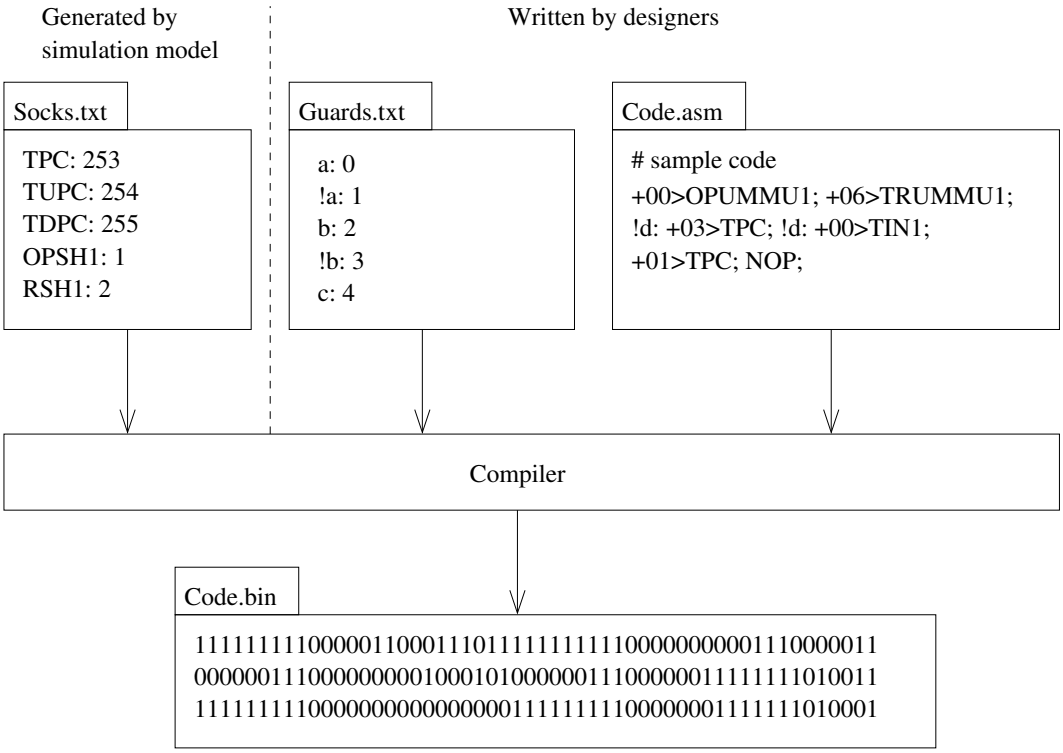


Figure 5.4: Inputs and outputs of compiler. The model creates the socks.txt file while the developers write the assembly code and design the guard expressions.

folder with the simulation model, and it is loaded from there every time the simulation is executed.

The compiler was updated only slightly in comparison to the previous version. Since several features of the new model are easily configurable and some of them, such as the address bus width, affect the width of the instruction word, these parameters were added also to the compiler. The compiler uses the same definition file as the simulation model and has to be recompiled when the definitions change. The old compiler and simulation model exchanged the program code in hexadecimal format. This was changed to binary since then there is no need to first do decimal and binary to hexadecimal conversion in the compiler and hexadecimal to binary conversion in the simulation model. Readability of the compiled program code is also improved since the fields in the instruction words are not usually multiples of four bits making hexadecimal representation next to impossible

to read for a human.

5.6 Instantiating a model instance

Simulation is instantiated by creating instances of wanted modules and support classes inside sc_main function in the top level main.cpp file. An example of this is seen in figure 5.5. The first object to create is the system clock. It is of type sc_clock and takes a few parameters including period, start time etc. Then exactly one NetworkController and Buscontroller instance are needed. In addition, a Testbench that creates stimuli for the model is created. This is discussed in detail in the next chapter. After these objects the needed functional units can be created. The guard bits in the FUs are connected manually to preferred signals going to the network controller. Other initializations are done automatically. When all the FUs are created simulation is started by calling sc_start and giving the run time as a parameter. If no parameters are passed the simulation runs forever or until some severe error is detected.

Constructing a synthesizable model is otherwise similar but Buscontroller cannot be used. This means that the signals of the interconnection network need to be created manually in the main function. Then the ports of the network controller and FUs are bound to these signals manually. The ports are public member variables of the classes and the task in general is trivial but has to be done very carefully. Also the amount of work can be large. For example a processor with three buses and ten functional units needs nine signals for the buses and in addition one signal per guard. The network controller has nine ports for the buses and one for each guard bit, and the functional units may be estimated to have an average of 25 ports per FU. As a result there will be over 250 ports that need to be bound to correct signals.

```
//include needed headers
int sc_main(int argc, char argv[]){
  sc_time simPeriod(20, SC_US); sc_time simStartTime(5, SC_US);
  sc_clock clk("clock", simPeriod, 0.5, simStartTime, true);
  Testbench tb("Testbench", clk);
  Buscontroller bctrl(clk);
  NetworkController ntc("Netc", clk);
  Shifter sh1("SH1", clk);
  sh1.guardBit(bctrl.getGuard(3));
  // creation of rest of the FUs similarily
  sc_time runtime(60000, SC_US);
  sc_start(runtime);
  return 0;
};
```

Figure 5.5: Example of a top level simulation file.

5.7 Simulation model outputs

The primary purpose of the simulation model is to verify and analyze the functionality of the system. To enable this the simulator produces many kinds of outputs.

For every clock cycle the simulation model outputs in text format the number of the cycle, contents of the buses and miscellaneous debugging information about active functional units. Therefore it is possible to easily analyze and verify the behavior of for example new functional units by printing out verbose debugging messages. When this output is printed on the screen it however becomes a major performance bottleneck for the execution speed of the simulation so it is advisable to redirect it to a text file or to null device if it is not needed. It is also possible to calculate accurately how many clock cycles are needed to execute the given algorithm written in assembly.

The simulation model also generates a VCD file containing the waveforms of every bus and guard signal and clock signal. This file can be viewed with any waveform viewer

supporting this format. Contents of the data buses are also dumped to an ASCII file and the utilization percentage can be calculated based on this information with a separate tool.

Chapter 6

Testing and results

In order to test the functionality of the model a simple test case was implemented. A processor was designed for verifying the header structure of an IPv6 packet, calculating and verifying the checksum for its TCP payload and extracting the payload of the TCP packet. For this task, the following functional units were needed for protocol prosessing: shifter, masker, matcher, counter, comparator and checksum calculator. A user memory management unit was needed for storing a few 32-bit values that cannot be dispatched as immediate values on an 8-bit address buses, input and output FUs were needed for connecting the processor to the network interface and a data memory management unit with DMA support and direct connections to input and output FUs was used for storing and retrieving incoming and outgoing data. In addition four general purpose registers were needed for storing temporary results.

To simulate the network interface of the processor the testbench of the old simulation model was utilized. It was connected to the input and output FUs, and it read simulated packed data from text files and fed them to the input FU. Test packets were hand crafted to the file to verify functionality with different types of packets with different types of payloads and errors. The data from the output interface was written to another file to verify that incoherent packets were dealt with and that correct ones remained intact during and after the processing.

After analyzing the task it soon became apparent that two buses is the optimal amount for this task. One bus would severally slow down processing while at least one of three buses would be idle at least 90% of the time. Two buses allow suitable amount of parallelism in calculations though the speed of the task is ultimately limited by the memory. Due to DMA it is however possible to start processing a packet immediately when the first words arrive without going to have to wait until it is entirely stored to the memory.

A slightly simplified flow chart of the software part can be seen in in figure 6.1. First the validity of fields in the IPv6 header is checked, then the pseudo header is constructed and finally the checksum is calculated for the TCP packet. If the checksum is correct, the payload is written out, in other cases the data is discarded. This code fits in 128 instruction words, or 864 bytes (instruction word width for this processor is 54 bits). Approximately 2030 processor cycles are needed to process one 1500 bytes long packet. The program code leaves however much room for optimization since it was designed for testing purposes only.

The execution speed of the simulator was also analyzed by running this algorithm. An almost legacy PC with two 1GHz Pentium III processors, 256 MB of RAM and Linux can simulate some 35 000 processor cycles per minute. The simulation speed however depends heavily on the architecture of the processor design under test (number of buses and FUs) and also the software portion which defines the number of operations per simulated cycle.

6.1 Synthesis

As discussed before the simulation model was constructed following the synthesizable subset where feasible to simplify synthesis later. However the used tool, Celoxica Agility SystemC Compiler, was not available until months after the task had been completed so as a result the synthesizability could not be verified during the implementation. This proved

fatal as the limitations were far more severe than anticipated.

The synthesis tests were done by using the tool to compile the SystemC modules one by one into synthesizable VHDL. The starting point were the simplest modules which in this case were the sockets. The functionality of the VHDL code was verified by constructing a VHDL testbench for each module, simulating the behaviour with Mentor Modelsim, and restructuring the SystemC code until desired behaviour was achieved.

The tool did not support inheritance for other than data members, and since much of the model was constructed with object oriented methodology in mind, the code had to be restructured extensively by removing nearly all the object orientation. Every process declaration had to be moved to the bottom level in the inheritance hierarchy effectively making the top level classes next to useless. The only possible function the parent modules could be used is to act as interface classes with no functionality. In hardware design this is especially of limited value since the modules are always decoupled due to the use of RTL or transaction level ports as an interface mechanism. This very restricted relation between parent and child class resembles that of entity and architecture in VHDL and cannot really be called inheritance since it does not offer almost any of the benefits of object orientation.

Lack of proper synthesizable data structures also proved to be a big problem. In the simulation model the functional units constructed automatically needed a number of sockets for themselves and this rid the designer from this monotonous task. In general this kind of dynamic instantiation of configurable hierarchical module structures could simplify hardware design by making the modules more reusable and removing trivial manual work. While this can be achieved e.g. in VHDL with the generate statement, in synthesizable SystemC it is not possible as there are no data structures where the module references or pointers could be stored. Natural choice would be to use an array of pointers but for some reason they are excluded from the subset. Creating an array of modules themselves does not work since SystemC requires each module to have a unique name

given as a constructor parameter and because of this writing a default constructor for a module is not possible. There exists a library function for generating these names but its use is not supported by the tool. This situation could be though easily remedied in the future when the synthesis tools are developed further.

Because of these limitations the code eventually had to be completely restructured in order to be synthesized. The three socket types that were the simplest modules in the system were successfully synthesized, but even the simplest functional units proved to be problematic. Eventually it became apparent that synthesizing them would require them to be rewritten almost from scratch since first of all as mentioned before the code had to be restructured and second the functionality of the processes did not compile into working VHDL. It would seem that in order to write synthesizable SystemC code the most important considerations are about structuring, but also the functional descriptions should be very carefully implemented. A software oriented approach should be mostly avoided and principles used in traditional hardware description language based should be followed. When the modules are independent and have a very clear block chart structure the synthesis should cause less problems. Generally an algorithm written with SystemC does not necessarily synthesize even if the guidelines given by the manuals are followed, and the abstraction level of synthesizable SystemC appears to be close to that of traditional HDLs. These limitations effectively negate many of the advantages SystemC could offer.

6.2 Criticism

During the project it became apparent that C++ is not the optimal choice for high level hardware modeling in every aspect. Its biggest advantages might be that the same language can be used for hardware and software design in projects concerning HW/SW codesign. However there are some problematic features, mostly because the age of the language and tools are clearly showing and adding unnecessary complexity.

First of all, the exclusive use of the standard C array as the sole data structure in the synthesizable subset is somewhat problematic. The array is a very low level structure and while in software domain it is extremely effective since it compiles efficiently into assembly, this benefit has no value in hardware modeling. Since it lacks basic features such as boundary checks and the size has to be known statically at compile time, it causes some unnecessary awkwardness and may introduce bugs that are hard to detect. Ironically the C array is more low level than the one found from e.g. VHDL even though one of the most important goals of the SystemC is to increase the level of abstraction and expression.

There are also other kind of features in C++ that are dead weight in hardware design. The pointers and references and memory management in general are useless features that offer nothing for hardware design but are a huge source of bugs. The compilation model of the language and the compilers themselves, mostly GCC, could also be better. The biggest problem is that the designer has to use effort for determining dependencies and linking. While in software development better control over the compiler may be useful in some cases, in hardware modeling it is only a hindrance.

Because of these issues one could argue that a modern language with similar reference semantics with e.g. Java, better synthesizable data structures, automatic garbage collection and modern compiler technology could be a better choice as basis for a high level hardware description language. Advanced integrated development environments with SystemC support may offer some improvements in the future though.

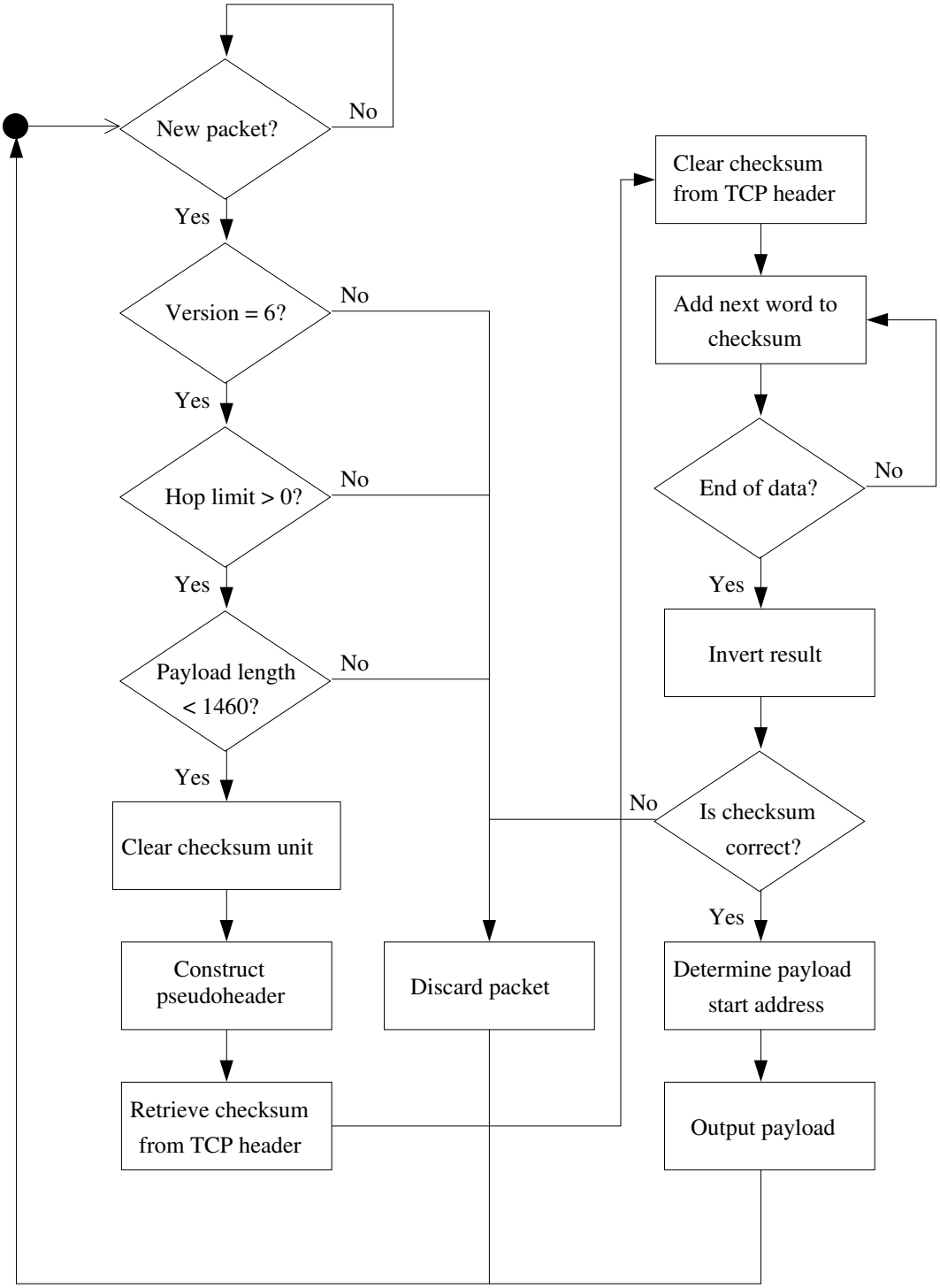


Figure 6.1: A flow chart of the software part of the test case.

Chapter 7

Conclusion

This thesis covered object oriented SystemC 2.x based hardware simulation especially in the protocol processing domain. A SystemC based simulation environment was developed for a TTA based protocol processor architecture using an old version as a starting point. A simulation model for a simple processor for TCP/IP packet validation was developed and tested with the environment for testing purposes. Synthesizability of the environment was also explored.

It was observed that SystemC 2.x has evolved significantly from the version 1.0 and now has good support for an object oriented approach to hardware modeling. This enables the creation of configurable and reusable module descriptions and testbenches. It also makes it easy to construct simulation environments for hardware platforms such as the one used in this thesis. These kind of environments simplify and speed up desing tasks as they enable rapid prototyping and design space exploration.

SystemC synthesis proved problematic. The synthesizable subset of SystemC is very limited. Many if not most of the powerful features that give significant advantage to SystemC compared with traditional HDLs are excluded from this subset. Inheritance is one of the biggest missing features and this severely restricts the level of expression. Also higher level data structures than the standard C array cannot be used, and e.g. advanced channel types such as FIFOs and buffers are not supported. In practice this means that

synthesizable SystemC does not offer as high level of abstraction as could be hoped for. The difference to e.g. VHDL becomes remarkable only when the functional complexity of the modules grows significantly. Although synthesizable SystemC offers increased level of expression compared with VHDL, the latter has the advantage of giving more control over the hardware and therefore giving more room for optimization.

It was also observed that the exact limitations of the synthesizable subset are not well defined, and much of the details depend on the tool that is used since the draft provided by OSCI only gives vague outlines. Since the synthesis tool was not available during the development of the simulation environment, synthesizability of its basic structures could not be tested until months after. Then it was obseved that much of the code had to be once again rewritten or at least extensively restructured even though the draft for the subset was followed. Therefore it would appear that if both simulation and synthesis are done with SystemC this decision should be done already in an early phase, and the model should be tested also with the synthesis tool on every major increment. There also exists a trade-off between utilizing the powerful expression of C++ in its full extent and pursuing synthesizability of the model since it appears they are very difficult to fit into the same model, and refining a high level simulation model towards synthesis requirements requires significant effort.

To conclude, new versions of SystemC offer powerful features for hardware simulation and verification and hardware/software codesign, but current synthesis tools leave room for improvement. In the future, incorporating object orientation and facilities such as higher level data types and structures to the synthesizable subset could offer significant benefits for hardware design.

References

[1] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, Inc., Upper Saddle River, New Jersey, USA, 4th edition, 2003.

[2] John D. Day and Hubert Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71:1334–1340, December 1983.

[3] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22(5):637–648, May 1974.

[4] Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A. *RFC 791: Internet Protocol - DARPA Internet Program Protocol Specification*, September 1981.

[5] V. Fuller, T. Li, and K. Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. *RFC 1519*, Sept. 1993.

[6] P. Srisuresh. Traditional IP network address translator (traditional NAT). *RFC 3022*, January 2001.

[7] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. *RFC 2460*, December 1998.

[8] Information Sciences Institute, University of Southern California, Marina del Rey, CA, U.S.A. *RFC 761: DOD Standard Transmission Control Protocol*, January 1980.

[9] Jari Nurmi, editor. *Processor Design - System-On-Chip Computing for ASICs and FPGAs*. Dordrecht, The Netherlands, 2007.

[10] Frank Vahid and Tony Givargis. *Embedded System Design, A Unified Hardware/-Software Introduction*. John Wiley Sons, Inc, Hoboken, New Jersey, USA, 2002.

[11] Seppo Virtanen. *A Framework for Rapid Design and Evaluation of Protocol Processors*. PhD thesis, Dept. of Information Technology, University of Turku, Finland, 2004.

[12] A. Jantsch, J. Öberg, and A. Hemani. Is there a niche for a general protocol processor core? In *Proceedings of the 16th IEEE Norchip Conference*, pages 93–100, Lund, Sweden, November 1998.

[13] Stuart Swan, Cadence Design Systems, Inc. *An Introduction to System Level Modeling in SystemC 2.0*, 2001.

[14] Open SystemC Initiative. *SystemC Version 2.0 User's Guide*, 2002.

[15] The Institute of Electrical and Electronics Engineers, Inc., New York, NY, U.S.A. *IEEE Std 1666-2005, IEEE Standard SystemC Language Reference Manual*, 2006.

[16] SystemC Language Working Group. *Functional Specification for SystemC 2.0*, 2002.

[17] Synthesis Working Group of Open SystemC Initiative. *SystemC Synthesizable Subset Draft 1.1.18*, 2004.

Appendix A

Simulation model text file inputs

Contents of the file socks.txt

TPC: 253
TUPC: 254
TDPC: 255
OPSH1: 1
RSH1: 2
TLRSH1: 3
TLLSH1: 4
TLSH1: 5
OPCM1: 6
RCM1: 7
TEQCM1: 8
TLZCM1: 9
TGZCM1: 10
TEQZCM1: 11
TLEQCM1: 12
TLTCM1: 13
TGEQCM1: 14
TGTCM1: 15
RC1: 16
TSCC1: 17
TICC1: 18
TDCC1: 19
OPM1: 20
ODM1: 21
RM1: 22
TM1: 23
OPMS1: 24
ODMS1: 25
RMS1: 26
TMS1: 27
OPCH1: 28
ODCH1: 29
RCH1: 30

TRCCH1: 31
TCCCH1: 32
OPRLI: 33
ODRLI: 34
RRLI: 35
TMTURLI: 36
TLLARLI: 37
TUNIRLI: 38
TCSTRLI: 39
TSMTURLI: 40
TSLLARLI: 41
TSUNIRLI: 42
TSCSTRLI: 43
OPIC: 44
ODIC: 45
RIC: 46
TIC: 47
RR1: 48
TR1: 49
RR2: 50
TR2: 51
RR3: 52
TR3: 53
RR4: 54
TR4: 55
OPUMMU1: 56
ODUMMU1: 57
RUMMU1: 58
TRMMUMMU1: 59
TWMMUMMU1: 60
OPDMMU1: 61
ODDMMU1: 62
RDMMU1: 63
TRMMDMMU1: 64
TWMMDMMU1: 65
RIN10: 66
RIN11: 67
RIN12: 68
TIN1: 69
OPOUT1: 70
ODOUT1: 71
TOUT1: 72

Contents of the file guards.txt

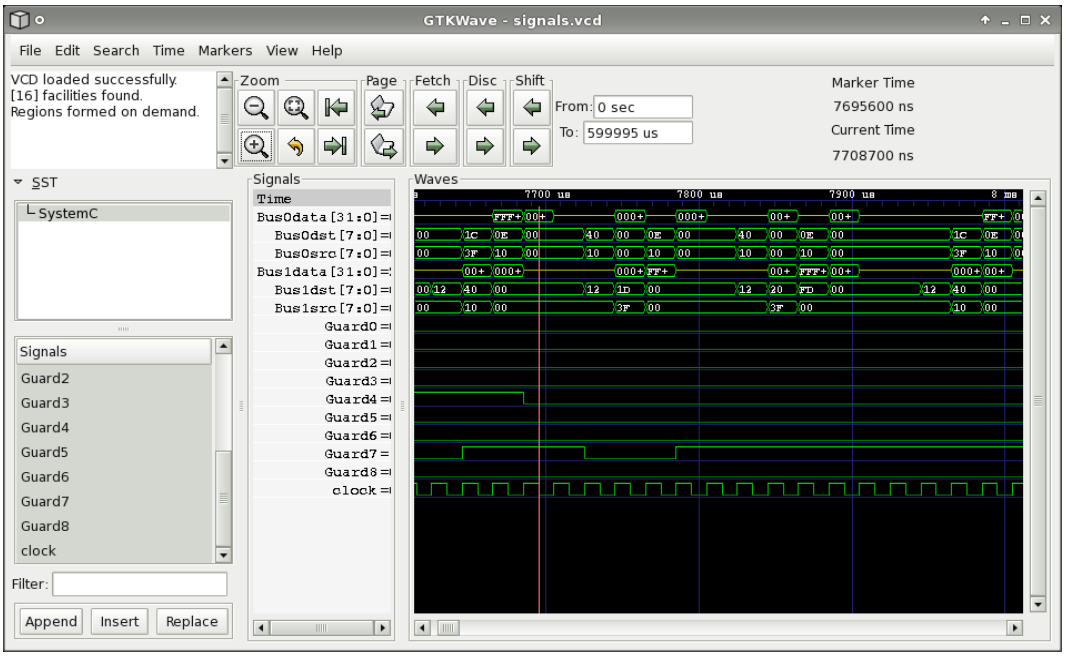
a: 0
!a: 1
b: 2
!b: 3
c: 4
!c: 5
d: 6
!d: 7
e: 8
!e: 9
a.b : 10

```
!a.b : 11
a.!b : 12
!a.!b: 13
```

Appendix B

Signal trace file

An example of a signal trace file viewed with GTKWave.



Appendix C

Source code

Source code of the model is listed on the following pages in alphabetical order, one source file per page. To save space source code for most of the trivial functional units in the library is omitted.

```
1 /**
2  * Bus is class modelling a bus in a transport triggered processor. It is a collection ↙
      of needed signals, that are connected elsewhere to network controller and sockets  ↙
      of functional units.
3  */
4
5 #ifndef Bus_H
6 #define Bus_H
7
8 #include "globaldefs.h"
9
10 #include "systemc.h"
11 class Bus: public sc_module {
12
13   public:
14
15   ////////////////////////////////////////////////////
16   // Signals
17   ////////////////////////////////////////////////////
18
19   /**
20    * Data bus.
21    */
22   sc_signal_rv<BUSWIDTH> sigData;
23
24   /**
25    * Source address bus.
26    */
27   sc_signal<sc_uint<ADDRESSWIDTH> > sigSrc;
28
29   /**
30    * Destination address bus.
31    */
32   sc_signal<sc_uint<ADDRESSWIDTH> > sigDst;
33
34   SC_HAS_PROCESS(Bus);
35   /**
36    * Constructor.
37    *
38    * \param name Name of this module.
39    */
40   Bus(const sc_module_name name): sc_module(name){
41   }
42
43 };
44 #endif // Bus_H
45
```

```
 1 #include "bus.h"
 2
 3 int Bus::busCnt = 0;
 4
 5 char* Bus::names[BUSES];
 6
 7 char* Bus::generateName(){
 8   char name[] = "Busx";
 9   name[3] = busCnt;
10   names[busCnt] = name;
11   return names[busCnt++];
12 }
13
```

```
/**
 * Buscontroller is a static class that handles the buses of the system.
 */

#ifndef Buscontroller_H
#define Buscontroller_H

#include "globaldefs.h"
#include "bus.h"
#include "systemc.h"
#include <fstream>

class Buscontroller {

  private:

  /**
   * Tracer for VCD file output.
   */
  static sc_trace_file* bustracer;

  /**
   * Array of bus pointers, initialized in constructor.
   */
  static Bus* buses[];

  /**
   * Number of sockets in the system.
   */
  static int socketCnt;

  /**
   * File writer for socket ID output.
   */
  static ofstream fileWriter;

  /**
   * Array of guard signals.
   */
  static sc_signal<bool> guards[];

  public:

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Returns a pointer to bus i.
   *
   * \pre (0 <= i <= BUSES-1 )
   * \return (0 <= i <= BUSES) ? (pointer to bus i) : 0
   * \param i Number of the bus.
   */
  static Bus* getBus(int i);

  /**
   * Returns a reference to guard i.
   */
  static sc_signal<bool>& getGuard(int i);

  /**
   * Returns an id for a socket. If number of sockets exceeds the range
   * of address bus, error message is displayed in simulation is stopped.
   *
   * \param socketName Name of the socket.
   */
  static sc_uint<ADDRESSWIDTH> getSocketId(const char* socketName);

  /**
   * Constructor.
   */
  Buscontroller(sc_clock &c);

  ~Buscontroller();

};
#endif // Buscontroller_H

```

```
#include "busctrl.h"
#include <math.h>
#include <string>

Bus* Buscontroller::buses[BUSES];

sc_signal<bool> Buscontroller::guards[GUARDCNT];

int Buscontroller::socketCnt = 0;

ofstream Buscontroller::fileWriter;

sc_trace_file* Buscontroller::bustracer = sc_create_vcd_trace_file("signals");

Buscontroller::Buscontroller(sc_clock &c){

  fileWriter.open("socks.txt");

  sc_trace(bustracer, c, "clock");

  for(int i = 0; i < BUSES; i++){
    char bname[] = "Bus ";
    bname[3] = i+48;
    buses[i] = new Bus(bname);
    sc_trace(bustracer, (buses[i]->sigData), bname);
  }

  for(int i = 0; i < GUARDCNT; i++){
    char gname[] = "Guard ";
    gname[5] = i+48;
    sc_trace(bustracer, guards[i], gname);
  }

  // write PC addresses
  int TPCaddr = (int)pow(2, ADDRESSWIDTH) - 3;
  int TUPCaddr = (int)pow(2, ADDRESSWIDTH) - 2;
  int TDPCaddr = (int)pow(2, ADDRESSWIDTH) - 1;
/*
  char str1[3];
  char str2[3];
  char str3[3];
  sprintf(str1, "%02X", TPCaddr);
  sprintf(str2, "%02X", TUPCaddr);
  sprintf(str3, "%02X", TDPCaddr);
*/
  fileWriter << "TPC" << ": " << TPCaddr << endl
             << "TUPC" << ": " << TUPCaddr << endl
             << "TDPC" << ": " << TDPCaddr << endl;
}

Buscontroller::~Buscontroller(){
  cout << "Destructing buscontroller ... ";

  fileWriter.close();

  sc_close_vcd_trace_file(bustracer);

  for(int i = 0; i < BUSES; i++){
    delete buses[i];
  }
  cout << "done" << endl;
}

Bus* Buscontroller::getBus(int i){
    if( (i < BUSES) & (i >= 0) ){
    return buses[i];
    }
    else{
       cout << "ERROR: Bus " << i << " does not exist, there are "
            << BUSES << " buses in the system!" << endl;
       exit(1);
    }
}

sc signal<bool>& Buscontroller::getGuard(int i){
  if( (i < GUARDCNT) & (i >= 0) ){
    return guards[i];
  }
  else{
    cout << "ERROR: Guard " << i << " does not exist, there are "
         << GUARDCNT << " guards in the system!" << endl;
    exit(1);
  }
}

sc_uint<ADDRESSWIDTH> Buscontroller::getSocketId(const char* socketName){
  socketCnt++;
  if(socketCnt == (pow(2, ADDRESSWIDTH) -1) ){
    cout << "ERROR: too many sockets, increase ADDRESSWIDTH." << endl;
    exit(1);
  }
  else{
    /*
    char str[3];
    sprintf(str, "%02X", socketCnt);
    */
    fileWriter << socketName << ": " << socketCnt << endl;
    sc_uint<ADDRESSWIDTH> addr = socketCnt;
    return addr;
  }
}
```

```
 1 #ifndef Checksum_H
 2 #define Checksum_H
 3
 4 #include "globaldefs.h"
 5 #include "fu.h"
 6 #include "systemc.h"
 7
 8 /**
 9  * Checksum calculates the Internet checksum.
10  */
11
12 class Checksum: public FunctionalUnit{
13
14   private:
15
16   /**
17    * Holds temporary result needed in consequtive calculations.
18    */
19   sc_uint<32> result_storage;
20
21   /**
22    * Array of strings that represent the name of operations.
23    */
24   static char* opNames[];
25
26   public:
27
28   /**
29    * Calculate checksum.
30    *
31    * msw = 16 most significant bits
32    * lsw = 16 least significant bits
33    * A = OP_msw + OP_lsw + OD_msw + OD_lsw + TR_msw + R'
34    * B = A_msw + A_lsw
35    * R' = B + B_carry
36    * RESULT = (R')'
37    *
38    */
39   void triggerOperation();
40
41   SC_HAS_PROCESS(Checksum);
42   Checksum(sc_module_name name,
43                sc_clock &c,
44                sc_uint<ADDRESSWIDTH>* opId = zero,
45                sc_uint<ADDRESSWIDTH>* resId = zero,
46                sc_uint<ADDRESSWIDTH>* trigIds = zero) : FunctionalUnit(name, c, 2, 1, ↙
      2, opId, resId, trigIds, opNames){
47     if(BUSWIDTH < 32){
48       cout << name << ": ERROR: this FU functions properly only when bus width is at   ↙
      least 32" << endl;
49       exit(1);
50     }
51     else{
52       if(BUSWIDTH > 32){
53         cout << name << ": Warning: this FU operates internally with 32 bits."
54              << endl;
55       }
56     }
57   }
58
59 };
60 #endif // Checksum_H
61
```

```
 1 #include "chksum.h"
 2
 3 char* Checksum::opNames[] = {"TRC", "TCC"};
 4
 5 void Checksum::triggerOperation(){
 6   int opC = 0;
 7   sc_uint<16> temp[6];
 8   sc_uint<32> temp_result = 0;
 9   sc_uint<32> temp_result_16b = 0;
10   sc_uint<32> op = 0;
11   sc_uint<32> od = 0;
12   sc_uint<32> tr = 0;
13   sc_uint<32> res = 0;
14
15   opC=opCode.read();
16   switch(opC) {
17     case 0:
18       // reset checksum unit
19       resultReg[0].write(0);
20       result_storage = 0;
21     break;
22
23     case 1:
24       // calculate checksum
25       op = operandReg[0].read();
26       od = operandReg[1].read();
27       tr = triggerReg.read();
28
29       op = ~op;
30       od = ~od;
31       tr = ~tr;
32       temp[0] = op.range(31,16);
33       temp[1] = op.range(15,0);
34       temp[2] = od.range(31,16);
35       temp[3] = od.range(15,0);
36       temp[4] = tr.range(31,16);
37       temp[5] = tr.range(15,0);
38
39       temp_result = 0;
40       for(int i = 0; i < 6; i++){
41         temp_result += temp[i];
42       }
43       temp_result += result_storage;
44
45       temp_result_16b = temp_result.range(15,0) + temp_result.range(31,16);
46
47       result_storage = temp_result_16b.range(15,0) + temp_result_16b.range(31,16);
48
49       res = ~result_storage;
50       resultReg[0].write(res.range(15,0));
51     break;
52   }
53
54   cout << name() << " triggered, op:  "<< op << " od : " << od << " trig: "  << tr << "↙
       result: " << res.range(15,0) << endl;
55
56 }
57
```

```
#ifndef COMPILER_H
#define COMPILER_H

#include <fstream>
#include <string>
#include <sstream>
#include <iostream>
#include <map>
#include <vector>
#include <systemc.h>

#include "globaldefs.h"
#define INSTRLENGTH ((GUARDCNT + (2*ADDRESSWIDTH))*BUSES)+IMMCNT
#define SUBINSTRLENGTH GUARDCNT + (2*ADDRESSWIDTH)

using namespace std;

class Compiler{

  private:
  const string GUARDDELIM;
  const string INSTRDELIM;
  const string REGDELIM;
  const string SPACECHAR;
  const string IMMIDENTIFIER;
  const string COMMENTIDENTIFIER;

  sc_bv<GUARDCNT> GTRUE;

  int lineCounter;

  map<string, sc_bv<ADDRESSWIDTH> > addresses;
  map<string, sc_bv<GUARDCNT> > guards;

  ofstream* outfile;
  ifstream* codefile;

  /**
   * Read guard numbers from given file.
   */
  void initGuards(string guardFilename);

  /**
   * Read socket addresses from given file.
   */
  void initAddr(string addrFilename);

  /**
   * Parse binary instruction word from a vector of subinstructions in string format.
   */
  sc_bv<INSTRLENGTH> getInstr(vector<string> subinstructions);

  /**
   * Parse binary destination address from given string.
   */
  sc_bv<ADDRESSWIDTH> extractDst(string* subInstr);

  /**
   * Parse binary source address or short immediate from given string, clear
   * this part from the string, and set bit i from immbits to 1 if immediate
   * was found.
   */
  sc_bv<ADDRESSWIDTH> extractSrc(string* subInstr, int i, sc_bv<IMMCNT>* immbits);

  /**
   * Parse binary guard word from given string, and clear guard expressions from the string.
   */
  sc_bv<GUARDCNT> extractGuards(string* subInstr);

  public:
  /**
   * Constructor.
   *
   * \param codeFilename Name of the source code file.
   * \param addrFilename Name of the file containing  socket addresses.
   * \param guardFilename Name fo the file containing guard numbers.
   */
  Compiler(string codeFilename, string addrFilename, string guardFilename);

  ~Compiler();

  /**
   * Make compilation and produce code.bin file.
   */
  void compile();

};
#endif
```

```
#include "compiler.h"

void Compiler::initGuards(string guardFilename){
  ifstream guardfile( guardFilename.c_str() );
  string line;
  unsigned int location;
  string::size_type posBeginIdx = 0, posEndIdx = 0;

  if (!guardfile) cerr << "ERROR: unable to open input file guards.txt!" << endl;
    else{
      cout << "Succesfully opened " << guardFilename << endl;
      while (getline(guardfile,line)) {
        if (line.empty());  //ignore empty lines
        else {
          location = line.find_first_of( SPACECHAR, 0 );
          while ( location != string::npos ) { // remove spaces
            line.erase(location,1);
            location = line.find_first_of( SPACECHAR, 0);
          }
          posEndIdx = line.find(GUARDDELIM,posBeginIdx);
          int ignumber = 0;
          istringstream ss( line.substr(posEndIdx+1,line.length()) );
          ss >> ignumber;
          sc_uint<GUARDCNT> ugnumber = ignumber;
          sc_bv<GUARDCNT> bgnumber = ugnumber;
          guards[line.substr(0,posEndIdx)] = bgnumber;
      } // else  (not an empty line)
    } // while lines left
    guardfile.close();
  }
  /*
  map<string, sc_bv<GUARDCNT> >::iterator pos = guards.begin();
  do{
    cout << pos->first << " " << pos->second << endl;
    pos++;
  }
  while(pos != guards.end());
  */
};

void Compiler::initAddr(string addrFilename){
  ifstream addressfile( addrFilename.c_str() );
  string line;
  unsigned int location;
  string::size_type posBeginIdx = 0, posEndIdx = 0;

  if (!addressfile) cerr << "ERROR: unable to open input file socks.txt!" << endl;
  else{
    cout << "Succesfully opened " << addrFilename << endl;
    while (getline(addressfile,line)) {
      if (line.empty());  //ignore empty lines
      else {
        location = line.find_first_of( SPACECHAR, 0 );
        while ( location != string::npos ) { // remove spaces
          line.erase(location,1);
          location = line.find_first_of( SPACECHAR, 0);
        }
        posEndIdx = line.find(GUARDDELIM,posBeginIdx);
        int iaddr = 0;
        istringstream ss( line.substr(posEndIdx+1,line.length()) );
        ss >> iaddr;
        sc_uint<ADDRESSWIDTH> uaddr = iaddr;
        sc_bv<ADDRESSWIDTH> baddr = uaddr;
        addresses[line.substr(0,posEndIdx)] = baddr;
    } // else  (not an empty line)
  } // while lines left
  addressfile.close();
  }
  /*
  map<string, sc_bv<ADDRESSWIDTH> >::iterator pos = addresses.begin();
  do{
    cout << pos->first << " " << pos->second << endl;
    pos++;
  }
  while(pos != addresses.end());
  */
```

```
};

void Compiler::compile(){

  string line = "";
  unsigned int location = 0;
  string::size_type posBeginIdx = 0, posEndIdx = 0;

  while(getline(*codefile,line)){
    // process line if it isn't empty or a comment line
    if( (!line.empty()) && !(line.substr(0,1) == COMMENTIDENTIFIER) ){
      // remove space characters
      location = line.find_first_of( SPACECHAR, 0 );
      while ( location != string::npos ) {
        line.erase(location,1);
        location = line.find_first_of( SPACECHAR, 0);
      }

      // exctract subinstructions
      vector<string> subinstructions;
      while (line.length() > 0){
        posBeginIdx = 0;
        posEndIdx = line.find(INSTRDELIM,posBeginIdx);
        subinstructions.push_back(line.substr(posBeginIdx,posEndIdx));
        line.erase(posBeginIdx,posEndIdx+1);
      }

      sc_bv<INSTRLENGTH> instruction = getInstr(subinstructions);
      *outfile << instruction.to_string() << endl;
    }
    lineCounter++;
  }

};

sc_bv<INSTRLENGTH> Compiler::getInstr(vector<string> subinstructions){

  sc_bv<IMMCNT> immediates;
  sc_bv<SUBINSTRLENGTH> subinstr[BUSES];
  sc_bv<INSTRLENGTH> instruction;

  for(int i = 0; i < subinstructions.size(); i++){
    if(!subinstructions.at(i).empty()){
      string sub = subinstructions.at(i);
      string src = "", dst = "";

      // find guards
      subinstr[i].range((SUBINSTRLENGTH)-1, (SUBINSTRLENGTH)-(GUARDCNT)) =
    extractGuards(&sub);

      unsigned int loc = sub.find( ("NOP"), 0 );
      if( loc == string::npos ){
        // not a NOP
        // extract src or imm
        subinstr[i].range((2*(ADDRESSWIDTH))-1, ADDRESSWIDTH) = extractSrc(&sub, i, &
    immediates);

        // extract dst
        subinstr[i].range((ADDRESSWIDTH)-1, 0) = extractDst(&sub);
      }

      int left = IMMCNT+i*(SUBINSTRLENGTH);
      int right = (i+1)*(SUBINSTRLENGTH) + IMMCNT - 1;
      instruction.range(right, left) = subinstr[i];
    }
  }
  instruction.range(IMMCNT-1, 0) = immediates;
  return instruction;
};

sc_bv<ADDRESSWIDTH> Compiler::extractDst(string* subInstr){
  map<string, sc_bv<ADDRESSWIDTH> >::iterator cur = addresses.find(*subInstr);
  if (cur != addresses.end())  {
    //correct DST address
    return addresses[*subInstr];
  }
```

```
  else {
    cout << "\n ERROR: Invalid DST address " << *subInstr << "at line " << lineCounter
     << endl;
    exit(1);
  }
};

sc_bv<ADDRESSWIDTH> Compiler::extractSrc(string* subInstr, int i, sc_bv<IMMCNT>*
    immbits){
  string src = "";
  unsigned int loc = subInstr->find( REGDELIM, 0 );
  if( loc != string::npos ) {
    src = subInstr->substr(0,loc);  //take src out of string
    subInstr->erase(0,loc+1);
    // check for immediate
    if ( src.substr(0,1) == IMMIDENTIFIER){
      src.erase(0,1);
      (*immbits)[i] = true;
      int imm = 0;
      istringstream ss( src );
      ss >> imm;
      sc_uint<ADDRESSWIDTH> uimm = imm;
      sc_bv<ADDRESSWIDTH> bimm = uimm;
      return bimm;
    }
    else{
      map<string, sc_bv<ADDRESSWIDTH> >::iterator cur = addresses.find(src);
      if (cur != addresses.end()) {
        //correct SRC address
        return addresses[src];
      }
      else {
        cout << "\n ERROR: Invalid SRC address " << src << "at line " << lineCounter <
    < endl;
        exit(1);
      }
    }
  }
  return 0;
};

sc_bv<GUARDCNT> Compiler::extractGuards(string* subInstr){
  unsigned int loc = subInstr->find( GUARDDELIM, 0 );
  string g = "";

  if( loc != string::npos ){
    // found guard bits
    g = subInstr->substr(0,loc);  //take grds out of string
    subInstr->erase(0,loc+1);
    map<string, sc_bv<GUARDCNT> >::iterator cur  = guards.find(g);
    if (cur != guards.end()){  //correct grd string
      return guards[g];
    }
    else {
      cout << "\n ERROR: in guard expression " << g << "at line " << lineCounter <<
    endl;
      return GTRUE;
    }
  }
  else {
    // did not find guard bits
    return GTRUE;
  }
};

Compiler::Compiler(string codeFilename, string addrFilename, string guardFilename):
  GUARDDELIM(":"), INSTRDELIM(";"), REGDELIM(">"), SPACECHAR(" "),
  IMMIDENTIFIER("+"), COMMENTIDENTIFIER("#"){

  outfile = new ofstream("code.bin");
  codefile = new ifstream(codeFilename.c_str());

  initGuards(guardFilename);
  initAddr(addrFilename);

  lineCounter = 0;
```

```

  for(int i = 0; i < GUARDCNT; i++){
    GTRUE[i] = true;
  }

};

Compiler::~Compiler(){
  if(outfile != 0){
    outfile->close();
    delete outfile;
  }
  if(codefile != 0){
    codefile->close();
    delete codefile;
  }
};
```

```
#ifndef DMMU_H
#define DMMU_H

#define PDULENGTH 375
#include "globaldefs.h"
#include "fu.h"
#include "mmu.h"
#include "systemc.h"

/**
 * dMMU is a protocol data memory management unit for accessing and storing
 * protocol data.
 */

class dMMU: public MMU{

  private:

  /**
   * Array for storing information which "slots" are in use. Memory is divided to slots with
   * a size of PDULENGTH words.
   */
  bool* datagramSlotInUse;

  /**
   * Number of PDU slots in the memory. Initialized in constructor as amount/PDULENGTH.
   */
  const int SLOTCNT;

  /**
   * Is there a DMA read operation in progress.
   */
  bool isReading;

  /**
   * Base address for DMA read.
   */
  int baseReadAddress;

  /**
   * Counter for DMA read.
   */
  int readCounter;

  /**
   * Is there a DMA write operation in progress.
   */
  bool isWriting;

  /**
   * Base address for DMA write.
   */
  int baseWriteAddress;

  /**
   * Counter for DMA read.
   */
  int writeCounter;

  public:

  /**
   * Guard bit to the network controller singnalling if the MMU is in the middle of read operation.
   */
  sc_out<bool> guardBitRead;

  /**
   * Guard bit to the network controller singnalling if the MMU is in the middle of write operation.
   */
  sc_out<bool> guardBitWrite;

  ////////////////////////////////////////////////////

  // Connections for InputFU
  ////////////////////////////////////////////////////
  /**
   * Input data from InputFU.
   */
  sc_in<sc_uint<BUSWIDTH> > inData;

  /**
   * Trigger from InputFU.
   */
  sc_in<bool> inTrigger;

  /**
   * New PDU indication from InputFU.
   */
  sc_in<bool> DStart;

  /**
   * Starting address of the PDU for InputFU.
   */
  sc_out<sc_uint<BUSWIDTH> > inAddress;

  ////////////////////////////////////////////////////
  // Connections for OutputFU
  ////////////////////////////////////////////////////
  /**
   * Starting address of a PDU to be sent.
   */
  sc_in<sc_uint<BUSWIDTH> > outAddress;

  /**
   * Trigger for OutputFU.
   */
  sc_in<bool> outTrigger;

  /**
   * Acknowledge to OutputFU for starting a DMA transfer.
   */
  sc_out<bool> outAck;

  /**
   * Output data to OutputFU.
   */
  sc_out<sc_uint<BUSWIDTH> > outData;

  ////////////////////////////////////////////////////
  // Signals
  ////////////////////////////////////////////////////

  sc_signal<sc_uint<BUSWIDTH> > sigInData;
  sc_signal<sc_uint<BUSWIDTH> > sigInAddress;
  sc_signal<sc_uint<BUSWIDTH> > sigOutData;
  sc_signal<sc_uint<BUSWIDTH> > sigOutAddress;
  sc_signal<bool> sigInTrigger;
  sc_signal<bool> sigDStart;
  sc_signal<bool> sigOutTrigger;
  sc_signal<bool> sigOutAck;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Read from input FU, or configure a new DMA transfer.
   */
  void inputData();

  /**
   * Write to output FU, or configure a new DMA transfer.
   */
  void outputData();

  /**
   * Perform memory transfer defined by opcode.
   *
   * Opcode Operation
   * 0      if(!isReading) read from memory address OP+TR (base+offset)
   * 1      if(!isWriting) write OD to memory address OP+TR
   */
  void triggerOperation();

  SC_HAS_PROCESS(dMMU);
  /**
   * Constructor.
   *
   * \param amount Amount of memory (in words).
   */
  dMMU(sc_module_name name,
                sc_clock &c,
                int amount,
                sc_uint<ADDRESSWIDTH>* opId = zero,
                sc_uint<ADDRESSWIDTH>* resId = zero,
                sc_uint<ADDRESSWIDTH>* trigIds = zero
       ): MMU(name, c, amount, opId, resId, trigIds), SLOTCNT(amount/(PDULENGTH)){

    isReading = isWriting = false;

    datagramSlotInUse = new bool[SLOTCNT];

    for(int i = 0; i < SLOTCNT; i++){
      datagramSlotInUse[i] = false;
    }

    // connect ports to corresponding signals
    inData(sigInData);
    inAddress(sigInAddress);
    inTrigger(sigInTrigger);
    DStart(sigDStart);
    outData(sigOutData);
    outAddress(sigOutAddress);
    outTrigger(sigOutTrigger);
    outAck(sigOutAck);

    SC_CTHREAD(inputData, clk.pos());

    SC_CTHREAD(outputData, clk.pos());

  }

  ~dMMU(){
     delete[] datagramSlotInUse;
   }

};
#endif // DMMU_H
```

```
#include "dmmu.h"

void dMMU::inputData(){

  int sleeptime = 1;
  while(true){
    if(!isReading){
      // there are no unfinished DMA reads
      if(inTrigger.read()){
        // start new DMA transfer
        int i = 0;
        while((i < SLOTCNT) | (!isReading)){
          if(datagramSlotInUse[i] == false){
            datagramSlotInUse[i] = true;
            isReading = true;
            baseReadAddress = i*(PDULENGTH);
            cout << name() << " New DMA transfer to slot " << i
                 << " with base address " << baseReadAddress << endl;
            inAddress.write(baseReadAddress);
            readCounter = 0;
            guardBitRead.write(true);
            sleeptime = 2; // must sleep two cycles before transfer can start
            break;
          }
          ++i;
        }
        if(!isReading){
          cout << "Warning: " << name() << " could not start DMA read: memory full" <<
     endl;
        }
      }
    }
    else{
      // there is an unfinished DMA read
      if(DStart.read()){
        // new PDU starts, find free slot
        readCounter = 0;
        for(int i = 0; i < SLOTCNT; i++){
          if(datagramSlotInUse[i] == false){
            datagramSlotInUse[i] = true;
            baseReadAddress = i*(PDULENGTH);
            inAddress.write(baseReadAddress);
            sleeptime = 2; // must sleep two cycles before transfer can start
          }
        }
        if(!isReading){
          cout << "Warning: " << name() << " could not start DMA read: memory full" <<
     endl;
          guardBitRead.write(false);
          isReading = false;
        }
      }
      else{
        // continuing old read operation
        write(baseReadAddress, readCounter, inData.read());
        ++readCounter;
        if((readCounter == 375) | (inTrigger.read() == false )){
          // PDU slot is read full, ending read operation
          readCounter = 0;
          isReading = false;
          guardBitRead.write(false);
        }
      }
    }
    wait(sleeptime);
    sleeptime = 1;
  }
}

void dMMU::outputData(){
  while(true){
    if(!isWriting){
      // there are no unfinished writes
      if(outTrigger.read()){
        // start a new DMA write
        baseWriteAddress = outAddress.read();
        isWriting = true;
        outAck.write(true);
        guardBitWrite.write(true);
        writeCounter = 0;
      }
    }
    else{
      // there is a DMA write

      // clear ack bit during the first write
      if(writeCounter == 0) outAck.write(false);
      outData.write(read(baseWriteAddress, writeCounter));
      ++writeCounter;
      if((writeCounter == 375) | (outTrigger.read() == false)){
        // PDU slot is fully written, ending write operation
        writeCounter = 0;
        isWriting = false;
        guardBitWrite.write(false);
        int index = baseWriteAddress/(PDULENGTH);
        datagramSlotInUse[index] = false;
        wait(1);
    outData.write(0);
      }
    }
  wait(1);
  }
}


void dMMU::triggerOperation(){
  int opC = 0;
  sc_uint<32> tr = 0, op = 0, od = 0, r = 0;

  tr = triggerReg.read();
  op = operandReg[0].read();

  switch(opC){
    case 0:
      // read from memory
      //if(!isWriting){
        r = read(op, tr);
        resultReg[0].write(r);
      //}
      /*
      else{
        cout << name() << " WARNING: memory cannot be read when there is DMA transfer
    between dMMU and OutputFU" << endl;
        resultReg[0].write(UINT_MAX);
      }*/
    break;

    case 1:
      // write to memory
      //if(!isReading){
        od = operandReg[1].read();
        write(op, tr, od);
      //}
      /*
      else{
        cout << name() << " WARNING: memory cannot be written when there is DMA
    transfer between dMMU and InputFU" << endl;
        resultReg[0].write(UINT_MAX);
      }*/
    break;
  }
}
```

```
/**
 * Basic FIFO with limitted parametrizable capacity.
 */

#ifndef FIFO_H
#define FIFO_H

template < class T >
class FIFO {

  private:

  /**
   * Pointer to array where data is stored.
   */
  T* dataStorage;

  /**
   * Size of the FIFO, i.e. maximum amount of stored items.
   */
  const int SIZE;

  /**
   * Current amount of items.
   */
  int items;

  /**
   * Head of the FIFO, i.e. index of the oldest item.
   */
  int head;

  /**
   * Tail of the FIFO, i.e. index of next free slot.
   */
  int tail;

  public:

  /**
   * Read item from the FIFO. Returns the oldest item and removes it. Do not read from empty FIFO!
   *
   * \pre !isEmpty()
   * \post Oldest item is returned and removed.
   * \return FIFO[head]
   */
  T read(){
    if(!isEmpty()){
      int ret = head;
      if(head == (SIZE-1)){
        head = 0;
      }
      else{
        ++head;
      }
      --items;
      return dataStorage[ret];
    }
    else{
      return dataStorage[head];
    }
  }

  /**
   * Write item to the FIFO. If FIFO is full, nothing is written and value 1 is returned.
   *
   * \post if(!isFull()) FIFO[tail] = data
   * \return (isFull() ? 1 : 0)
   * \param data Item to be added to the FIFO
   */
  int write(T data){
    if(!isFull()){
      dataStorage[tail] = data;
      if(tail == (SIZE-1)){
        tail = 0;
      }
      else{
        ++tail;
      }
      ++items;
      return 0;
    }
    else{
      return 1;
    }
  }

  /**
   * Is FIFO full.
   */
  bool isFull(){
    return (items == SIZE);
  }

  /**
   * Is FIFO empty.
   */
  bool isEmpty(){
    return (items == 0);
  }

  /**
   * Size of the FIFO.
   */
  int size(){
    return SIZE;
  }

  /**
   * Constructor.
   *
   * \param capacity Size of the FIFO.
   */
  FIFO(const int capacity): SIZE(capacity){
    dataStorage = new T[SIZE];
    head = tail = items = 0;
  }

  /**
   * Destructor.
   */
  ~FIFO(){
    delete[] dataStorage;
  }

};
#endif // FIFO_H
```

```
/**
 * FunctionalUnit is an abstract class modeling a general functional unit in a
    transport triggered processor.
 */

#ifndef FunctionalUnit_H
#define FunctionalUnit_H

#include "systemc.h"
#include "insocket.h"
#include "outsocket.h"
#include "trigsocket.h"

class FunctionalUnit: public sc_module {

  protected:

  /**
   * Array containing one zero. Can be used for default initialization of constructor
     parameters.
   */
  static sc_uint<ADDRESSWIDTH> zero[];

  public:
  /**
   * Pointer to an array of pointers to input sockets.
   */
  InSocket** operand;

  /**
   * Pointer to an array of pointers to output sockets.
   */
  OutSocket** result;

  /**
   * Pointer to trigger socket.
   */
  TriggerSocket* trigger;


  ////////////////////////////////////////////////////
  // Ports
  ////////////////////////////////////////////////////

  /**
   * Clock input.
   */
  sc_in_clk clk;

  /**
   * Trigger bit.
   */
  sc_in<bool> trigBit;

  /**
   * Array of operand registers.
   */
  sc_in<sc_uint<BUSWIDTH> >* operandReg;

  /**
   * Trigger register.
   */
  sc_in<sc_uint<BUSWIDTH> > triggerReg;

  /**
   * Array of result registers.
   */
  sc_out<sc_uint<BUSWIDTH> >* resultReg;

  /**
   * Operation code.
   */
  sc_in<sc_uint<OPCODEWIDTH> > opCode;

  ////////////////////////////////////////////////////
  // Signals
```

```
  ////////////////////////////////////////////////////

  /**
   * Array of signals connecting operand register and input sockets.
   */
  sc_signal<sc_uint<BUSWIDTH> >* sigFuOperand;

  /**
   * Signal connecting trigger register and trigger socket.
   */
  sc_signal<sc_uint<BUSWIDTH> > sigFuTrigger;

  /**
   * Signal connecting opcode register and trigger socket.
   */
  sc_signal<sc_uint<OPCODEWIDTH> > sigOpCode;

  /**
   * Array of signals connecting result register and output sockets.
   */
  sc_signal<sc_uint<BUSWIDTH> >* sigFuResult;

  /**
   * Trigger signal connecting trigger socket and trigger port
   */
  sc_signal<bool> sigTrigBit;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Executes function triggerOperation if triggerbit is 1. Since checkTrigger is a
   * SC_CTHREAD, wait commands may be added inside function triggerOperation for
   * additional exection delays.
   */
  void checkTrigger();

  /**
   * Operation performed when FU is triggered.
   */
  virtual void triggerOperation()=0;

  SC_HAS_PROCESS(FunctionalUnit);
  /**
   * Constructs the base of functional unit. Given amount of sockets are created
   * and bound to buses.
   *
   * Parameters opId, resId and trigId are used for giving sockets predefined
   * addresses. Then lengths must match the amount sockets respectively. Alternatively
     ,
   * if automatic address generation is not used, an array with value 0 at the first
   * index may be passed.
   *
   * \param name Name of this module.
   * \param c Clock input.
   * \param inputCnt Amount of input sockets.
   * \param outputCnt Amount of output sockets.
   * \param trigIdCnt Amount of trigger IDs.
   * \param opId Pointer to addresses of input sockets.
   * \param resId Pointer to addresses of output sockets.
   * \param trigIds Pointer to addresses of the trigger sockets.
   * \param trigIdNames Pointer to string array containing names for trigger
     operations.
   * \pre   (opId.length == inputCnt || opId[0] == 0)
   *      & (resId.length == outputCnt || resId[0] == 0)
   *      & (trigIds.length == MAXTRIGGERIDS || trigIds[0] == 0)
   */
  FunctionalUnit(const sc_module_name& name,
                 sc_clock &c,
         int inputCnt,
         int outputCnt,
         int trigIdCnt,
                 sc_uint<ADDRESSWIDTH>* opId = zero,
                 sc_uint<ADDRESSWIDTH>* resId = zero,
                 sc_uint<ADDRESSWIDTH>* trigIds = zero,
```

```
           char** trigIdNames = 0) : sc_module(name){

    clk(c); // bind clock to c

    char operandnames[inputCnt][MAXSOCKNLENGTH];
    char resultnames[outputCnt][MAXSOCKNLENGTH];

    if(inputCnt > 0){
      operand = new InSocket*[inputCnt];
      operandReg = new sc_in<sc_uint<BUSWIDTH> >[inputCnt];
      sigFuOperand = new sc_signal<sc_uint<BUSWIDTH> >[inputCnt];
    }
    else{
      operand = 0;
      operandReg = 0;
      sigFuOperand = 0;
    }

    if(outputCnt > 0){
      result = new OutSocket*[outputCnt];
      resultReg = new sc_out<sc_uint<BUSWIDTH> >[outputCnt];
      sigFuResult = new sc_signal<sc_uint<BUSWIDTH> >[outputCnt];
    }
    else{
      result = 0;
      resultReg = 0;
      sigFuResult = 0;
    }

    // generate socket names
    // might not be neccessary, much easier to do with string class if
    // synthetization allows it to be used in constructors
   for(int i = 0; i < inputCnt; i++){
      operandnames[i][0] = 'O';
      operandnames[i][1] = ( i == 1 ? 'D' : 'P');
      int j = 2;
      while( j < (MAXSOCKNLENGTH - 3) ){
        operandnames[i][j] = name[j-2];
        j++;
        if(name[j-2] == '\0') break;
      }
      // socket names will be OPXXX, ODXXX and OPXXX3 -> OPXXX9 and OPXXXA->
      // ASCII codes for 1 and A are 48 and 55+10=65
      if(i < 3){
        operandnames[i][j] = '\0';
      }
      else{
        operandnames[i][j] = i + (i < 10 ? 50 : 55);
        operandnames[i][j+1] = '\0';
      }
    }

    for(int i = 0; i < outputCnt; i++){
      resultnames[i][0] = 'R';
      int j = 1;
      while( j < (MAXSOCKNLENGTH - 2) ){
        resultnames[i][j] = name[j-1];
        j++;
    if(name[j-1] == '\0') break;
      }
      // socket names will be XXX1 -> XXX9 and XXXA->
      // ASCII codes for 1 and A are 48 and 55+10=65
      if((i == 0) && (outputCnt == 1)) resultnames[i][j] = '\0';
      else{
        resultnames[i][j] = i + (i < 10 ? 48 : 55);
        resultnames[i][j+1] = '\0';
      }
    }

    char triggername[MAXSOCKNLENGTH];
    triggername[0] = 'T';
    int k = 1;
    while( k < (MAXSOCKNLENGTH - 2) ){
      triggername[k] = name[k-1];
      k++;
      if(name[k-1] == '\0') break;
```

```
    }
    triggername[k] = '\0';

    // create and bind input sockets to operand reg
    for(int k = 0; k < inputCnt; k++){
      if((opId[0] == (sc_uint<ADDRESSWIDTH>)0) || (opId[k] == (sc_uint<ADDRESSWIDTH>)
      0)){
        operand[k] = new InSocket(operandnames[k]);
      }
      else{
        operand[k] = new InSocket(operandnames[k], opId[k]);
      }
      operand[k]->clock(c);
      operand[k]->fuData(sigFuOperand[k]);
      operandReg[k].bind(sigFuOperand[k]);
    }

    // create and bind output sockets to result reg
    for(int k = 0; k < outputCnt; k++){
      if((resId[0] == (sc_uint<ADDRESSWIDTH>)0) || (resId[k] == (sc_uint
      <ADDRESSWIDTH>)0)){
        result[k] = new OutSocket(resultnames[k]);
      }
      else{
        result[k] = new OutSocket(resultnames[k], resId[k]);
      }
      result[k]->clock(c);
      result[k]->fuData(sigFuResult[k]);
      resultReg[k].bind(sigFuResult[k]);
    }

    // create and bind trigger socket
    trigger = new TriggerSocket(triggername, trigIdCnt, trigIds, trigIdNames);
    trigger->clock(c);
    trigger->fuData(sigFuTrigger);
    triggerReg(sigFuTrigger);
    trigger->opCode(sigOpCode);
    opCode(sigOpCode);
    trigger->trigBit(sigTrigBit);
    trigBit(sigTrigBit);

    SC_CTHREAD(checkTrigger, clk.pos());

    cout << "FU " << name << " constructed" << endl;
  }

  ~FunctionalUnit(){
    cout << "Destructing " << name() << " ... ";
    if(operand != 0) delete[] operand;
    if(trigger != 0) delete trigger;
    if(result != 0) delete[] result;
    if(resultReg != 0) delete[] resultReg;
    if(operandReg != 0) delete[] operandReg;
    if(sigFuOperand != 0) delete[] sigFuOperand;
    if(sigFuResult != 0) delete[] sigFuResult;
    cout << "done" << endl;
  }
};
#endif // FunctionalUnit_H
```

```
1 #include "fu.h"
2
3 sc_uint<ADDRESSWIDTH> FunctionalUnit::zero[1] = {0};
4
5 void FunctionalUnit::checkTrigger(){
6   while(true){
7     if(trigBit.read()) triggerOperation();
8     wait(1);
9   }
10 }
11
```

```
#ifndef GLOBALDEFS
#define GLOBALDEFS

// width of data bus
#define BUSWIDTH 32
// width of address bus
#define ADDRESSWIDTH 8
// maximum value for address bus
#define MAXADDRESS 255
// number of buses
#define BUSES 2
// opcode width
#define OPCODEWIDTH 3
// number of guard bits in the system
#define GUARDCNT 9
// number of buses able for immediate dispatching
#define IMMCNT 4
// instruction word length
#define INSTRLENGTH ((GUARDCNT + (2*ADDRESSWIDTH))*BUSES)+IMMCNT
// amount of program memory in instruction words
#define PROGRAMMEM 256
// minimum length for program counter
// must be large enough to address whole program memory
#define PCWIDTH 9
// maximum length for a socket name
// needed in automatic name generation for sockets
#define MAXSOCKNLENGTH 20
// maximum amount of ids for a trigger socket
#define MAXTRIGGERIDS 8

#endif

```

```
1
2  #ifndef INPUTFU H
3  #define INPUTFU_H
4
5  #define IFIFOLENGTH 20
6  #include "globaldefs.h"
7  #include "fu.h"
8  #include "fifo.h"
9  #include "systemc.h"
10
11 /**
12  * InputFU is the input interface of the processor.
13  */
14
15 class InputFU: public FunctionalUnit{
16
17   private:
18
19   /**
20    * FIFO for addresses.
21    */
22   FIFO<sc_uint<BUSWIDTH> >* addressFIFO;
23
24   /**
25    * FIFO for interface IDs.
26    */
27   FIFO<sc_uint<BUSWIDTH> >* interfaceFIFO;
28
29   /**
30    * FIFO for PDU lengths.
31    */
32   FIFO<sc_uint<BUSWIDTH> >* lengthFIFO;
33
34   /**
35    * Mutual exclusion for FIFO access;
36    */
37   bool fifolock;
38
39   public:
40   /**
41    * Guard bit to the network controller signalling if the FIFOs are empty.
42    */
43   sc_out<bool> guardBitEmpty;
44
45   /**
46    * Guard bit to the network controller singalling if the FIFOs are full.
47    */
48   sc_out<bool> guardBitFull;
49
50   ////////////////////////////////////////////////////
51   // Connections to dMMU
52   ////////////////////////////////////////////////////
53
54   /**
55    * Data to dMMU
56    */
57   sc_out<sc_uint<BUSWIDTH> > iData;
58
59   /**
60    * Trigger to dMMU.
61    */
62   sc_out<bool> iTrigger;
63
64   /**
65    * New PDU indication to dMMU.
66    */
67   sc_out<bool> DStart;
68
69   /**
70    * Starting address of the PDU from dMMU.
71    */
72   sc_in<sc_uint<BUSWIDTH> > iAddress;
73
74   ////////////////////////////////////////////////////
75   // Connections to network interface
76   ////////////////////////////////////////////////////
77
78   /**
79    * Databus from network interface.
80    */
81   sc_in<sc_uint<BUSWIDTH> > networkData;
82
83   /**
84    * Databus from network interface.
85    */
86   sc_in<sc_uint<BUSWIDTH> > networkLength;
87
88   /**
89    * Trigger from network interface.
90    */
91   sc_in<bool> networkTrigger;
92
93   /**
94    * Acknowledge signal to network interface.
95    */
96   sc_out<bool> networkAck;
97
98   ////////////////////////////////////////////////////
99   // Functions
100  ////////////////////////////////////////////////////
101
102  /**
103   * Set memory address, interface ID and length of the oldest PDU to result registers
104   */
105  void triggerOperation();
106
107  /**
108   * Poll input interface for new data.
109   */
110  void pollInterface();
111
112  /**
113   * Update guard signals.
114   * Note: guard signals cannot be set directly from triggerOperation and
       pollInterface
115   * since signal can be driven by only one process.
116   */
117  void updateGuards();
118
119  SC_HAS_PROCESS(InputFU);
120  InputFU(sc_module_name name, sc_clock &c,
121          sc_uint<ADDRESSWIDTH>* opId = zero,
122          sc_uint<ADDRESSWIDTH>* resId = zero,
123          sc_uint<ADDRESSWIDTH>* trigIds = zero
124        ) : FunctionalUnit(name, c, 0, 3, 1, opId, resId, trigIds){
125
126    addressFIFO = new FIFO<sc_uint<BUSWIDTH> >(IFIFOLENGTH);
127    interfaceFIFO = new FIFO<sc_uint<BUSWIDTH> >(IFIFOLENGTH);
128    lengthFIFO = new FIFO<sc_uint<BUSWIDTH> >(IFIFOLENGTH);
129
130    SC_CTHREAD(pollInterface, clk.pos());
131
132    SC_METHOD(updateGuards);
133    sensitive << clk.pos();
134
135  }
136
137  ~InputFU(){
138    delete addressFIFO;
139    delete interfaceFIFO;
140    delete lengthFIFO;
141   }
142
143 };
144 #endif // InputFU_H
145
```

```
#include "inputfu.h"

void InputFU::triggerOperation(){
  if(!addressFIFO->isEmpty()){

    sc_uint<BUSWIDTH> temp1, temp2, temp3;
    temp1 = addressFIFO->read();
    temp2 = interfaceFIFO->read();
    temp3 = lengthFIFO->read();
    resultReg[0].write(temp1);
    resultReg[1].write(temp2);
    resultReg[2].write(temp3);
    cout << name() << " executing. Addr: " << temp1 << ", Id: " << temp2 << ", length: " << temp3 << endl;
/*
    resultReg[0].write(addressFIFO->read());
    resultReg[1].write(interfaceFIFO->read());
    resultReg[2].write(lengthFIFO->read());
*/
  }
  else {
    cout << name() << " executing, but there is no data." << endl;
    resultReg[0].write(0);
    resultReg[1].write(0);
    resultReg[2].write(0);
  }
}

void InputFU::updateGuards(){
  if(addressFIFO->isEmpty()){
    // FIFO is empty
    guardBitEmpty.write(true);
  }
  else{
    if(addressFIFO->isFull()){
      // FIFO is full
      guardBitFull.write(true);
    }
    else{
      // FIFO is neither full nor empty
      guardBitFull.write(false);
      guardBitEmpty.write(false);
    }
  }
}

void InputFU::pollInterface(){

  sc_uint<BUSWIDTH> data, length, address, interface;

  // only one interface for now
  interface = 0;

  while(true){
    if(networkTrigger.read() && !(addressFIFO->isFull())){
      cout << name() << ": reading PDU." << endl;
      //new data
      length = networkLength.read();
      iTrigger.write(true);
      networkAck.write(true);
      wait(2);

      address = iAddress.read();
      addressFIFO->write(address);
      interfaceFIFO->write(interface);
      lengthFIFO->write(length);

      for(int i = 0; i < (int)length; i++){
        data = networkData.read();
        cout << name() << ": writing PDU data " << data << endl;
        iData.write(data);
        if(i != ((int)length-1)) wait(1);
      }
      networkAck.write(false);
      iTrigger.write(false);
      cout << name() << ": PDU read successfully." << endl;
      wait(1);
      iData.write(0);
    }
    else{
      wait(1);
    }
  }

}
```

```
/**
 * InSocket is a class modeling input socket connecting buses and a FU in a transport triggered processor.
 */

#ifndef InSocket_H
#define InSocket_H

#include "globaldefs.h"
#include "socket.h"
#include "systemc.h"
class InSocket: public Socket {

  private:
  /**
   * ID of this socket.
   */
  sc_uint<ADDRESSWIDTH> socketId;

  /**
   * Read data from the bus on next cycle.
   */
  bool readData;

  /**
   * Index of the bus to be read.
   */
  int busNumber;

  public:

  ////////////////////////////////////////////////////
  // Ports
  ////////////////////////////////////////////////////
  /**
   * Array of input ports for buses.
   */
  sc_in_rv<BUSWIDTH> inDataPorts[BUSES];

  /**
   * Output port to the FU.
   */
  sc_out<sc_uint<BUSWIDTH> > fuData;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Decode addresses on address buses and perform actions if neccessary.
   */
  void decodeId();

  SC_HAS_PROCESS(InSocket);
  /**
   * Constructor.
   *
   * \param name_ Name of this module.
   * \param id Id of this socket.
   */
  InSocket(sc_module_name name_, sc_uint<ADDRESSWIDTH> id = 0): Socket(name_){
    // <NOT SYNTHESIZABLE>
    // bind input ports to data buses and id ports to Dst buses
    Bus* bptr;
    for(int i = 0; i < BUSES; i++){
      bptr = Buscontroller::getBus(i);
      inDataPorts[i].bind(bptr->sigData);
      inIdPorts[i].bind(bptr->sigDst);
    }
    // </NOT SYNTHESIZABLE>
    if(id == (sc_uint<ADDRESSWIDTH>)0){
      // <NOT SYNTHESIZABLE>
      socketId = Buscontroller::getSocketId(name_);
      // </NOT SYNTHESIZABLE>
    }
    else{
      socketId = id;
    }

    readData = false;
    busNumber = 99;
  }

};
#endif // InSocket_H

```

```
#include "insocket.h"

void InSocket::decodeId(){
    // Data read was scheduled during last cycle
    if(readData){
      cout << name() << " reading..." << endl;
      fuData.write(inDataPorts[busNumber].read());
      readData = false;
    }

    // Decode
    for(int i = 0; i < BUSES; i++) {
      if (socketId == inIdPorts[i].read()) {
        cout << name() << " triggered..." << endl;
    readData = true;
        busNumber = i;
      }
    }
}

```

```
#include "systemc.h"

#include "busctrl.h"
#include "netctrl.h"

// Functional Units:
#include "fu.h"
#include "shifter.h"
#include "comparator.h"
#include "masker.h"
#include "counter.h"
#include "matcher.h"
#include "chksum.h"
#include "rli.h"
#include "mmu.h"
#include "dmmu.h"
#include "inputfu.h"
#include "outputfu.h"
#include "icmp.h"
#include "register.h"

#include "testbench.h"
#include <ctime>

int sc_main(int argc, char* argv[]) {

  time_t startTime;
  time_t endTime;

  sc_time simPeriod(20, SC_US);
  sc_time simStartTime(5, SC_US);
  sc_clock clk("clock", simPeriod, 0.5, simStartTime, true);

  Buscontroller bctrl(clk);

  NetworkController ntc("Netc", clk);

  Shifter sh1("SH1", clk);
  sh1.guardBit(bctrl.getGuard(3));

  Comparator cm1("CM1", clk);
  cm1.guardBit(bctrl.getGuard(1));

  Counter c1("C1", clk);
  c1.guardBit(bctrl.getGuard(2));

  Matcher m1("M1", clk);
  m1.guardBit(bctrl.getGuard(0));

  Masker ms1("MS1", clk);

  Checksum ch1("CH1", clk);

  RouterLocalInfo rli("RLI", clk);

  ICMP ic("IC", clk);

  Register r1("R1", clk);

  Register r2("R2", clk);

  Register r3("R3", clk);

  Register r4("R4", clk);

  MMU ummu1("UMMU1", clk, 100);

  ////////////////////////////////////////////
  // I/O connections
  ////////////////////////////////////////////

  dMMU dmmu1("DMMU1", clk, 16000);
  dmmu1.guardBitRead.bind(bctrl.getGuard(7));
  dmmu1.guardBitWrite.bind(bctrl.getGuard(8));

  InputFU in1("IN1", clk);
  in1.guardBitEmpty.bind(bctrl.getGuard(4));
  in1.guardBitFull.bind(bctrl.getGuard(5));

  in1.iTrigger.bind(dmmu1.sigInTrigger);
  in1.iData.bind(dmmu1.sigInData);
  in1.DStart.bind(dmmu1.sigDStart);
  in1.iAddress.bind(dmmu1.sigInAddress);

  OutputFU out1("OUT1", clk);
  out1.guardBit.bind(bctrl.getGuard(6));

  out1.oTrigger.bind(dmmu1.sigOutTrigger);
  out1.oData.bind(dmmu1.sigOutData);
  out1.oAck.bind(dmmu1.sigOutAck);
  out1.oAddress.bind(dmmu1.sigOutAddress);

  Testbench tb("Testbench", clk);
  in1.networkData(tb.sigIfuData);
  in1.networkLength(tb.sigIfuDataLength);
  in1.networkTrigger(tb.sigIfuTrigger);
  in1.networkAck(tb.sigIfuAck);

  out1.networkData(tb.sigOfuData);
  out1.networkLength(tb.sigOfuDataLength);
  out1.networkTrigger(tb.sigOfuTrigger);
  out1.networkAck(tb.sigOfuAck);

  sc_time runtime(60000, SC_US);

  time(&startTime);

  sc_start(runtime);

  time(&endTime);

  double simulationTime = difftime(endTime, startTime);
  cout << "************************************************************" << endl;
  cout << " Simulation complete. Elapsed time was " << simulationTime << " seconds." <<
    < endl;
  cout << "************************************************************" << endl;

  return 0;
};
```

```
/**
 * MMU is a basic memory management unit for accessing memory.
 */

#ifndef MMU_H
#define MMU_H

#include "globaldefs.h"
#include "fu.h"
#include "systemc.h"

class MMU: public FunctionalUnit{

  protected:

  /**
   * Pointer to an array representing memory.
   */
  sc_uint<BUSWIDTH>* dataMemory;

  /**
   * Amount of addressable memory in words.
   */
  const int DATAMEMORY;

  /**
   * Array of strings that represent the name of operations.
   */
  static char* opNames[];

  /**
   * Read from memory address base+offset
   *
   * \pre base+offset < DATAMEMORY
   */
  sc_uint<BUSWIDTH> read(sc_uint<BUSWIDTH> base, sc_uint<BUSWIDTH> offset);

  /**
   * Write data to memory address base+offset
   *
   * \pre (base+offset < DATAMEMORY)
   * \post (dataMemory[base+offset] == data)
   */
  void write(sc_uint<BUSWIDTH> base, sc_uint<BUSWIDTH> offset, sc_uint<BUSWIDTH> data)
    ;

  public:

  /**
   * Perform memory transfer defined by opcode.
   *
   * Opcode Operation
   * 0      read from memory address OP+TR (base+offset)
   * 1      write OD to memory address OP+TR
   */
  void triggerOperation();

  SC_HAS_PROCESS(MMU);
  MMU(sc_module_name name,
                sc_clock &c,
                int amount,
                sc_uint<ADDRESSWIDTH>* opId = zero,
                sc_uint<ADDRESSWIDTH>* resId = zero,
                sc_uint<ADDRESSWIDTH>* trigIds = zero) : FunctionalUnit(name, c, 2, 1,
     2, opId, resId, trigIds, opNames), DATAMEMORY(amount){

    dataMemory = new sc_uint<BUSWIDTH>[amount];

    // initialize some values
    dataMemory[0]=0x00000000;
    dataMemory[1]=0xffffffff;
    dataMemory[2]=0x00000002;
    dataMemory[3]=(sc_uint<BUSWIDTH>)375;

    dataMemory[4]=(sc uint<BUSWIDTH>)1460;
    dataMemory[5]=(sc_uint<BUSWIDTH>)375;

    //b 11111111 11111111 11111111 11110000
    // mask for getting first 4 bits
    dataMemory[6]=(sc_uint<BUSWIDTH>)0xfffffff0; //4294967280;

    //b 00000000 11111111 11111111 11111111
    // mask for getting last 8 bits
    dataMemory[7]=(sc_uint<BUSWIDTH>)0xffffff;

    //b 11111111 11111111 00000000 00000000
    // mask for getting first 16 bits
    dataMemory[8]=(sc_uint<BUSWIDTH>)0xffff0000;

    //b 11111111 00000000 11111111 11111111
    // mask for getting next header field
    dataMemory[9]=(sc_uint<BUSWIDTH>)0xff00ffff;

    //b 00000000 00000000 11111111 11111111
    // mask for getting last 16 bits
    dataMemory[10]=(sc_uint<BUSWIDTH>)0x0000ffff;

  }

  ~MMU(){
    delete[] dataMemory;
  }

};
#endif // MMU_H
```

```
 1 #include "mmu.h"
 2
 3 char* MMU::opNames[] = {"TRMM", "TWMM"};
 4
 5 void MMU::triggerOperation(){
 6   int opC = 0;
 7   sc_uint<32> tr = 0, op = 0, od = 0, r = 0;
 8
 9   tr = triggerReg.read();
10   op = operandReg[0].read();
11
12   switch(opC){
13     case 0:
14       // read from memory
15       r = read(op, tr);
16       wait(1);
17       resultReg[0].write(r);
18     break;
19
20     case 1:
21       // write to memory
22       od = operandReg[1].read();
23       wait(1);
24       write(op, tr, od);
25     break;
26   }
27 }
28
29 sc_uint<BUSWIDTH> MMU::read(sc_uint<BUSWIDTH> base, sc_uint<BUSWIDTH> offset){
30   long addr = base + offset;
31   if((addr >= DATAMEMORY) | (addr < 0) ){
32     cout << name() << " ERROR: memory address " << addr << " out of bounds" << endl;
33     exit(1);
34   }
35   else{
36     cout << name() <<  " reading " << dataMemory[addr] << " from address " << addr <<
       endl;
37     return dataMemory[addr];
38   }
39 }
40
41 void MMU::write(sc_uint<BUSWIDTH> base, sc_uint<BUSWIDTH> offset, sc_uint<BUSWIDTH>
       data){
42   long addr = base + offset;
43   if((addr >= DATAMEMORY) | (addr < 0) ){
44     cout << name() << " ERROR: memory address " << addr << " out of bounds" << endl;
45     exit(1);
46   }
47   else{
48     cout << name() <<  " writing " << data << " to address " << addr << endl;
49     dataMemory[addr] = data;
50   }
51 }
52
```

```
1
2
3  #ifndef NetworkController_H
4  #define NetworkController_H
5
6  #include "globaldefs.h"
7  #include "busctrl.h"
8  #include "systemc.h"
9  #include "trigsocket.h"
10
11 #include <fstream>
12 #include <string>
13 using namespace std;
14
15 #define SUBINSTRLENGTH GUARDCNT + (2*ADDRESSWIDTH)
16
17 /**
18  * Network controller of a transport triggered processor.
19  */
20
21 class NetworkController : public sc_module {
22
23   private:
24
25   /**
26    * Output filestream for storing bit patterns on buses.
27    */
28   ofstream outfile;
29
30   /**
31    * Program counter.
32    */
33   sc_uint<PCWIDTH> pc;
34
35   /**
36    * Programmed jump is detected, no new fetches for three cycles.
37    */
38   bool jumpDetected;
39
40   /**
41    * Counter for counting cycles when jump is detected.
42    */
43   sc_uint<2> cycleCnt;
44
45   /**
46    * Specifies the buses where immediate values were written.
47    * Values need to be cleared after two cycles.
48    */
49   sc_bv<BUSES> dirtyBuses;
50
51   /**
52    * Array of 1 bit values for counting one cycle.
53    */
54   sc_uint<1> dirtyCnt[BUSES];
55
56   /**
57    * Temporary storage for immediate values.
58    */
59   sc_uint<ADDRESSWIDTH> immbuffer[BUSES];
60
61   /**
62    * All Z. Needed for clearing data bus after writing immediates.
63    */
64   sc_lv<BUSWIDTH> HIGHIMPEDANCE;
65
66   /**
67    * Bit vector with all ones. Used to evaluate guards.
68    */
69   sc_bv<GUARDCNT> GONES;
70
71   /**
72    * Program memory.
73    */
74   sc_bv<INSTRLENGTH> programMemory[PROGRAMMEM];
75
76   /**
77    * Temporary variable holding loaded instruction.
78    */
79   sc_bv<INSTRLENGTH> instruction;
80
81   /**
82    * Immediate bits of the current instruction.
83    */
84   sc_bv<IMMCNT> immediateBits;
85
86   /**
87    * Temporary storage for current subinstruction.
88    */
89   sc_bv<SUBINSTRLENGTH> subInstruction;
90
91   /**
92    * Guard bits if current subinstruction.
93    */
94   sc_bv<GUARDCNT> guardBits;
95
96   /**
97    * SRC field of current subinstruction.
98    */
99   sc_uint<ADDRESSWIDTH> srcValue;
100
101  /**
102   * DST field of current subinstruction.
103   */
104  sc_uint<ADDRESSWIDTH> dstValue;
105
106  /**
107   * Total number of completed cycles.
108   */
109  int totalCycleCnt;
110
111  /**
112   * IDs of the program counter.
113   */
114  sc_uint<ADDRESSWIDTH> PCTrigIds[4];
115
116  public:
117
118  /**
119   * Trigger socket for programmed jumps.
120   */
121  TriggerSocket* trigger;
122
123  ////////////////////////////////////////////////////
124  // Signals
125  ////////////////////////////////////////////////////
126
127  /**
128   * Trigger signal connecting trigger socket and trigger port
129   */
130  sc_signal<bool> sigTrigBit;
131
132  /**
133   * Signal connecting opcode register and trigger socket.
134   */
135  sc_signal<sc_uint<OPCODEWIDTH> > sigOpCode;
136
137  /**
138   * Signal connecting trigger register and trigger socket.
139   */
140  sc_signal<sc_uint<BUSWIDTH> > sigTrigger;
141
142  ////////////////////////////////////////////////////
143  // Ports
144  ////////////////////////////////////////////////////
145
146  /**
147   * Clock input.
148   */
149  sc_in_clk clock;
150
151  /**
152   * Trigger bit.
153   */
154  sc_in<bool> trigBit;
155
156  /**
157   * Operation code.
158   */
159  sc_in<sc_uint<OPCODEWIDTH> > opCode;
160
161  /**
162   * Trigger register.
163   */
164  sc_in<sc_uint<BUSWIDTH> > triggerReg;
165
166  /**
167   * Array of ports connected to data buses.
168   */
169  sc_inout_rv<BUSWIDTH> data[BUSES];
170
171  /**
172   * Array of ports connected to destination address buses.
173   */
174  sc_out<sc_uint<ADDRESSWIDTH> >dst[BUSES];
175
176  /**
177   * Array of ports connected to source address buses.
178   */
179  sc_out<sc_uint<ADDRESSWIDTH> > src[BUSES];
180
181  /**
182   * Array of guard input ports.
183   */
184  sc_in<bool> guards[GUARDCNT];
185
186  ////////////////////////////////////////////////////
187  // Functions
188  ////////////////////////////////////////////////////
189
190  /**
191   * Operation for updating program counter status.
192   *
193   * \post if (opcode == 0) pc = triggerReg
194   *       else if(opcode = 1) pc += triggerReg
195   *       else if(opcode = 2) pc -= triggerReg
196   */
197  void updatePc();
198
199  /**
200   * Fetch instruction pointed by pc, evaluate guards, split instruction words,
201   * dispatch addresses and dispatch immediates.
202   */
203  void fetch();
204
205  /**
206   * Evaluate given guards.
207   */
208  bool evaluateGuards(sc_uint<GUARDCNT> guards);
209
210  /**
211   * Constructor.
212   *
213   * \param name  Name of this module.
214   * \param c Clock input.
215   */
216  SC_HAS_PROCESS(NetworkController);
217  NetworkController(const sc_module_name name_, sc_clock &c) : sc_module(name_){
218
219    // <NOT SYNTHESIZABLE>
220    // bind data and address ports
221    Bus* bptr;
222    for(int i = 0; i < BUSES; i++){
223      bptr = Buscontroller::getBus(i);
224      data[i].bind(bptr->sigData);
225      dst[i].bind(bptr->sigDst);
226      src[i].bind(bptr->sigSrc);
227    }
228    // </NOT SYNTHESIZABLE>
229
230    for(int i = 0; i < GUARDCNT; i++){
231      guards[i].bind(Buscontroller::getGuard(i));
232    }
233
234    clock(c);
235    pc = 0;
236    jumpDetected = false;
237    cycleCnt = 0;
238    totalCycleCnt = 0;
239
240    // generate PC trigger addresses
241    PCTrigIds[0] = (sc_uint<ADDRESSWIDTH>)(MAXADDRESS-2);
242    PCTrigIds[1] = (sc_uint<ADDRESSWIDTH>)(MAXADDRESS-1);
243    PCTrigIds[2] = (sc_uint<ADDRESSWIDTH>)MAXADDRESS;
244    //PCTrigIds[3] = (sc_uint<ADDRESSWIDTH>)0;
245    trigger = new TriggerSocket("PC", 3, PCTrigIds);
246
247    // connect trigger
248    trigger->clock(c);
249    trigger->fuData(sigTrigger);
250    triggerReg(sigTrigger);
251    trigger->opCode(sigOpCode);
252    opCode(sigOpCode);
253    trigger->trigBit(sigTrigBit);
254    trigBit(sigTrigBit);
255
256    for(int i = 0; i < GUARDCNT; i++){
257      GONES[i] = true;
258    }
259
260    for(int i = 0; i < BUSWIDTH; i++){
261      HIGHIMPEDANCE[i] = SC_LOGIC_Z;
262    }
263
264    for(int i = 0; i < BUSES; i++){
265      dirtyBuses[i] = false;
266      dirtyCnt[i] = 0;
267    }
268
269    SC_METHOD(updatePc);
270    sensitive << clock.pos();
271
272    SC_METHOD(fetch);
273    sensitive << clock.pos();
274
275    cout << "Network controller constructed" << endl
276         << "Instruction word length is " << INSTRLENGTH << endl
277         << "Subinstruction word length is " << SUBINSTRLENGTH << endl;
278
279    outfile.open("bit_patterns.txt");
280
281    string b = "Bus";
282    for(int i = 0; i < BUSES; i++){
283      char idx = i+48;
284      string b2 = b + idx;
285      outfile << b2;
286      int b2size = b2.size();
287      for(int j = 0; j < BUSWIDTH - b2size + 1; j++){
288        outfile << " ";
289      }
290    }
291    outfile << endl;
292
293    ifstream infile( "code.bin" );
294    string word;
295    int i = 0;
296    if (!infile) cerr << "ERROR: unable to open input file code.bin!" << endl;
297    while (infile >> word) {
298      cout << "NetControl: read word " << i << " : " << word << endl;
299      //sc bv<INSTRLENGTH> instr = axtoi(word.c_str());
300      sc_bv<INSTRLENGTH> instr = word.c_str();
301      programMemory[i] = instr;
302      i++;
303      if(i == PROGRAMMEM){
304        cout << "ERROR: Out of program memory, program code is too large." << endl;
305      }
306    }
307
308    int codeLength = i;
309    cout << "NetControl: Read " << codeLength << " instructions to memory." << endl;
310    cout << "NetControl: program memory contents:" << endl;
311    for (i=0; i < codeLength; i++) {
312      cout << "NetControl: " << programMemory[i] << endl;
313    }
314
315  }
316
317  ~NetworkController(){
318      cout << "Destructing " << name() << " ... ";
319      if(trigger != 0) delete trigger;
320      cout << "done" << endl;
321    }
322 };
323 #endif // NetworkController_H
324
```

```
#include "netctrl.h"

void NetworkController::updatePc(){

  if(trigBit.read()){

    cout << name() << "Updating PC" << endl;

    sc_uint<BUSWIDTH> TR = triggerReg.read();
    sc_uint<OPCODEWIDTH> OP = opCode.read();

    switch((int)OP){
      case(0):
        pc = TR;
        cout << "PC is now " << pc << endl;
      break;

      case(1):
        pc += TR;
        cout << "PC is now " << pc << endl;
      break;

      case(2):
        if((int)pc < (int)TR){
          cout << "ERROR: Trying to subtract " << TR
               << " from " << pc << "! PC cannot be negative." << endl;
          exit(1);
        }
        pc -= TR;
        cout << "PC is now " << pc << endl;
      break;

      default:
        cout << "WARNING: undefined opcode for Network controller" << endl;
      break;
    }
  }
}

void NetworkController::fetch(){
  // clear address buses
  for(int i = 0; i < BUSES; i++){
    dst[i].write(0);
    src[i].write(0);
  }

  cout << "============================================================" << endl
       << "Starting cycle " << totalCycleCnt << endl;
  totalCycleCnt++;

  for(int i = 0; i < BUSES; i++){
    cout << "SRC" << i << ": " << src[i] << endl
         << "DST" << i << ": " << dst[i] << endl
         << "DATA" << i << ": " << data[i] << endl;

    outfile << data[i] << " ";
  }
  outfile << endl;

  // if immediate values were written two cycles ago, clear them out
  for(int i = 0; i < BUSES; i++){
    if(dirtyBuses[i] == true){
      if(dirtyCnt[i] == (sc_uint<1>)1){
        cout << "Clearing data buses" << endl;
        data[i].write(HIGHIMPEDANCE);
        dirtyCnt[i] = 0;
        dirtyBuses[i] = false;
      }
      else{
        dirtyCnt[i]++;
        data[i].write(immbuffer[i]);
      }
    }
  }

  // if programmed jump was detected, clear the pipeline by waiting for three cycles
  if(jumpDetected){
    if(cycleCnt < (sc_uint<2>)2){
      cycleCnt++;
      return;
    }
    else{
      cycleCnt = 0;
      jumpDetected = false;
      return;
    }
  }

  instruction = programMemory[pc];

  immediateBits = instruction.range(IMMCNT-1,0);

  cout << "Instruction " << pc << " : " << instruction << endl;
  cout << "Immediate bits : " << immediateBits << endl;

  // divide & dispatch instructions
  for(int i = 0; i < BUSES; i++){
    subInstruction = instruction.range(((i+1)*(SUBINSTRLENGTH))+IMMCNT-1, (i*(SUBINSTRLENGTH))+IMMCNT);
    guardBits = subInstruction.range(SUBINSTRLENGTH-1, 2*ADDRESSWIDTH);
    dstValue = (sc_bv<ADDRESSWIDTH>)subInstruction.range(ADDRESSWIDTH-1,0);
    srcValue = (sc_bv<ADDRESSWIDTH>)subInstruction.range(2*ADDRESSWIDTH-1, ADDRESSWIDTH);

    cout << "Subinstruction " << i << " is: " << subInstruction << endl;

    bool guardeval = true;
    if(guardBits != GONES){
      guardeval = evaluateGuards(guardBits);
    }

    cout << "Guard evaluation " << (guardeval ? "true" : "false") << endl;

    if(guardeval){

      // check if this is a jump command
      for(int k = 0; k < 3; k++){
        if(dstValue == PCTrigIds[k]){
          jumpDetected = true;
        }
      }

      if((i < IMMCNT) && (immediateBits[i] == true)){
        // write immediate value to buffer and dispatch address
        dst[i].write(dstValue);
        src[i].write(0); // clear old address from src
        immbuffer[i] = srcValue;
        dirtyBuses[i] = true;
        dirtyCnt[i] = 0;
        cout << "Writing immediate " << srcValue << " to " << dstValue << endl;
      }
      else{
        cout << "Moving from " << srcValue << " to " << dstValue << endl;
        dst[i].write(dstValue);
        src[i].write(srcValue);
      }
    }
    else{
      // old addresses need to be cleared
      dst[i].write(0);
      src[i].write(0);
    }
  }

  pc++;
  if((int)pc == (PROGRAMMEM-1)){
  //if((int)pc == 4){
    cout << "PC has reached the end of program memory" << endl;
    exit(0);
  }

}

bool NetworkController::evaluateGuards(sc_uint<GUARDCNT> grd){
  int iguard = grd;
  bool value = false;

  switch (iguard){
    case 0:
      // matcher 1 true, a
      if (guards[0] == true) {
        value = true;
      }
    break;

    case 1:
      // matcher 1 false, !a
      if (guards[0] == false) {
       value = true;
      }
    break;

    case 2:
      // compare 1 true, b
      if (guards[1] == true) {
        value = true;
      }
    break;

    case 3:
      // compare 1 false, !b
      if (guards[1] == false) {
        value = true;
      }
    break;

    case 4:
      // counter 1 zero, c
      if (guards[2] == true) {
        value = true;
      }
    break;

    case 5:
      // counter 1 not  zero, !c
      if (guards[2] == false) {
        value = true;
      }
    break;

    case 6:
      // inputFU empty
      if (guards[4] == true) {
        value = true;
      }
    break;

    case 7:
      // inputFU not empty
      if (guards[4] == false) {
        value = true;
      }
    break;

    case 8:
      // outputFU full
      if (guards[4] == true) {
        value = true;
      }
    break;

    case 9:
      // outputFU not full
      if (guards[4] == false) {
        value = true;
      }
    break;

    case 10:
      // matcher 1 and compare 1 true, a.b
      if ( (guards[0] == true) && (guards[1] == true) ) {
        value = true;
      }
    break;

    case 11:
      // matcher 1 false and compare 1 true, !a.b
      if ( (guards[0] == false) && (guards[1] == true) ) {
        value = true;
      }
    break;

    case 12:
      // matcher 1 true and compare 1 false, a.!b
      if ( (guards[0] == true) && (guards[1] == false) ) {
        value = true;
      }
    break;

    case 13:
      // matcher 1 false and compare 1 false, !a.!b
      if ( (guards[0] == false) && (guards[1] == false) ) {
        value = true;
      }
    break;

    default:
      value = true;
    break;
  }

  return value;
}
```

```
1
2  #ifndef OUTPUTFU H
3  #define OUTPUTFU_H
4
5  #define OFIFOLENGTH 50
6  #include "globaldefs.h"
7  #include "fifo.h"
8  #include "fu.h"
9  #include "systemc.h"
10
11 /**
12  * OutputFU is the output interface of the processor.
13  */
14
15 class OutputFU: public FunctionalUnit{
16
17   private:
18
19   /**
20    * FIFO for addresses.
21    */
22   FIFO<sc_uint<BUSWIDTH> >* addressFIFO;
23
24   /**
25    * FIFO for interface IDs.
26    */
27   FIFO<sc_uint<BUSWIDTH> >* interfaceFIFO;
28
29   /**
30    * FIFO for PDU lengths.
31    */
32   FIFO<sc_uint<BUSWIDTH> >* lengthFIFO;
33
34   public:
35
36   /**
37    * Guard bit to the network controller.
38    */
39   sc_out<bool> guardBit;
40
41   ////////////////////////////////////////////////////
42   // Connections to dMMU
43   ////////////////////////////////////////////////////
44
45   /**
46    * Starting address of a PDU to be sent.
47    */
48   sc_out<sc_uint<BUSWIDTH> > oAddress;
49
50   /**
51    * Trigger to dMMU.
52    */
53   sc_out<bool> oTrigger;
54
55   /**
56    * Acknowledge from dMMU for starting a DMA transfer.
57    */
58   sc_in<bool> oAck;
59
60   /**
61    * Output data from dMMU.
62    */
63   sc_in<sc_uint<BUSWIDTH> > oData;
64
65   ////////////////////////////////////////////////////
66   // Connections to network interface
67   ////////////////////////////////////////////////////
68
69   /**
70    * Databus to network interface.
71    */
72   sc_out<sc_uint<BUSWIDTH> > networkData;
73
74   /**
75    * Databus to network interface.
76    */
77   sc_out<sc_uint<BUSWIDTH> > networkLength;
78
79   /**
80    * Trigger to network interface.
81    */
82   sc_out<bool> networkTrigger;
83
84   /**
85    * Acknowledge signal from network interface.
86    */
87   sc_in<bool> networkAck;
88
89   ////////////////////////////////////////////////////
90   // Functions
91   ////////////////////////////////////////////////////
92
93   /**
94    * Put information about processed PDU to the FIFO.
95    *
96    * OP = starting memory address
97    * OD = PDU length
98    * TR = interface ID
99    */
100  void triggerOperation();
101
102  /**
103   * Send oldest PDU out.
104   */
105  void sendData();
106
107  /**
108   * Update guard signal.
109   * Note: guard signal cannot be set directly from triggerOperation and sendData
110   * since signal can be driven by only one process.
111   */
112  void updateGuards();
113
114  SC_HAS_PROCESS(OutputFU);
115  OutputFU(sc module name name, sc clock &c,
116    sc_uint<ADDRESSWIDTH>* opId = zero,
117    sc_uint<ADDRESSWIDTH>* resId = zero,
118    sc uint<ADDRESSWIDTH>* trigIds = zero)
119    : FunctionalUnit(name, c, 2, 0, 1, opId, resId, trigIds){
120
121    addressFIFO = new FIFO<sc uint<BUSWIDTH> >(OFIFOLENGTH);
122    interfaceFIFO = new FIFO<sc_uint<BUSWIDTH> >(OFIFOLENGTH);
123    lengthFIFO = new FIFO<sc_uint<BUSWIDTH> >(OFIFOLENGTH);
124
125    SC_CTHREAD(sendData, clk.pos());
126
127    SC METHOD(updateGuards);
128    sensitive << clk.pos();
129
130  }
131
132  ~OutputFU(){
133    delete addressFIFO;
134    delete interfaceFIFO;
135    delete lengthFIFO;
136  }
137
138 };
139 #endif // OutputFU_H
140
```

```
#include "outputfu.h"

void OutputFU::triggerOperation(){

  if(!addressFIFO->isFull()){
    sc_uint<BUSWIDTH> op, tr, od;
    op = operandReg[0].read();
    tr = triggerReg.read();
    od = operandReg[1].read();
    cout << name() << " executing with values " << op << " " << od << " " << tr << endl;

    if(od != (sc_uint<BUSWIDTH>)0){
      addressFIFO->write(operandReg[0].read());
      interfaceFIFO->write(triggerReg.read());
      lengthFIFO->write(operandReg[1].read());
    }
  }
}

void OutputFU::updateGuards(){
  if(addressFIFO->isFull()){
    guardBit.write(true);
  }
  else{
    guardBit.write(false);
  }
}

void OutputFU::sendData(){

  sc_uint<BUSWIDTH> address, id, length, data;

  while(true){
    if(addressFIFO->isEmpty()){
      // nothing to do
      cout << name() << ": idle" << endl;
      wait(1);
    }
    else{

      address = addressFIFO->read();
      id = interfaceFIFO->read();
      length = lengthFIFO->read();

      cout << name() << " writing PDU from address " << address << " with length "
           << length << " to interface " << id << endl;



      if(id >= (sc_uint<BUSWIDTH>)4){
        // writing to interface 5 discards PDU
        cout << name() << " discarding PDU" << endl;
        oAddress.write(address);
        oTrigger.write(true);
        wait(1);
        oTrigger.write(false);
        oAddress.write(0);
      }
      else{
        oTrigger.write(true);
        oAddress.write(address);
        networkTrigger.write(true);
        networkLength.write(length);

        // wait until dMMU is ready
        while(!oAck.read() || !networkAck.read()){
          wait(1);
        }
        wait(1);

        for(int i = 0; i < (int)length; i++){
          data = oData.read();
          networkData.write(data);
          cout << name() << " wrote " << data << " from dMMU to network interface." << endl;
          wait(1);
        }
        // transfer complete
        oTrigger.write(false);
        networkTrigger.write(false);
        wait(1);
        networkData.write(0);
        networkLength.write(0);
        oAddress.write(0);
        cout << name() << ": PDU written successfully." << endl;
        wait(1);
      }
    }
  }
}

```

```
/**
 * OutSocket is a class modeling output socket connecting buses and a FU in a transport triggered processor.
 */

#ifndef OutSocket_H
#define OutSocket_H

#include "globaldefs.h"
#include "socket.h"
#include "systemc.h"
class OutSocket: public Socket {

  private:
  /**
   * ID of this socket.
   */
  sc_uint<ADDRESSWIDTH> socketId;

  /**
   * Logic vector containing Z values. Used for disconnecting socket from data buses.
   */
  sc_lv<BUSWIDTH> HIGHIMPEDANCE;

  public:

  ////////////////////////////////////////////////////
  // Ports
  ////////////////////////////////////////////////////
  /**
   * Array of output ports to buses.
   */
  sc_out_rv<BUSWIDTH> outDataPorts[BUSES];

  /**
   * Input port for the FU.
   */
  sc_in<sc_uint<BUSWIDTH> > fuData;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Decode addresses on address buses and perform actions if neccessary.
   */
  void decodeId();

  SC_HAS_PROCESS(OutSocket);
  /**
   * Constructor.
   *
   * \param name_ Name of this module.
   * \param id Id of this socket.
   */
  OutSocket(sc_module_name name_, sc_uint<ADDRESSWIDTH> id = 0): Socket(name_){

    // <NOT SYNTHESIZABLE>
    // bind output ports to data buses and id ports to src buses
    Bus* bptr;
    for(int i = 0; i < BUSES; i++){
      bptr = Buscontroller::getBus(i);
      outDataPorts[i].bind(bptr->sigData);
      inIdPorts[i].bind(bptr->sigSrc);
    }
    // </NOT SYNTHESIZABLE>

    if(id == (sc_uint<ADDRESSWIDTH>)0){
      // <NOT SYNTHESIZABLE>
      socketId = Buscontroller::getSocketId(name_);
      // </NOT SYNTHESIZABLE>
    }
    else{
      socketId = id;
    }

  for(int i = 0; i < BUSWIDTH; i++){
    HIGHIMPEDANCE[i] = SC_LOGIC_Z;
  }

  }

};
#endif // OutSocket_H

```

```
#include "outsocket.h"

void OutSocket::decodeId(){

    for(int i = 0; i < BUSES; i++) {
      if(socketId == inIdPorts[i].read()) {
        cout << name() << " writing..." << endl;
        outDataPorts[i].write(fuData.read());
      }
      else{
        outDataPorts[i].write(HIGHIMPEDANCE);
      }
    }

}

```

```
1 /**
2  * Socket is an abstract class modeling a general socket connecting buses and a FU in a↙
      transport triggered processor.
3  */
4
5 #ifndef Socket_H
6 #define Socket_H
7
8 #include "globaldefs.h"
9 #include "busctrl.h"
10 #include "systemc.h"
11 class Socket: public sc_module {
12
13   public:
14
15   ////////////////////////////////////////////////////
16   // Ports
17   ////////////////////////////////////////////////////
18   /**
19    *  Clock input.
20    */
21   sc_in_clk clock;
22
23   /**
24    * Address buses.
25    */
26   sc_in<sc_uint<ADDRESSWIDTH> > inIdPorts[BUSES];
27
28   ////////////////////////////////////////////////////
29   // Functions
30   ////////////////////////////////////////////////////
31
32   /**
33    * Decode addresses on address buses and perform actions if neccessary.
34    */
35   virtual void decodeId()=0;
36
37   SC_HAS_PROCESS(Socket);
38   /**
39    * Constructor.
40    *
41    * \param name Name of this module.
42    */
43   Socket(const sc_module_name& name): sc_module(name){
44
45     SC_METHOD(decodeId);
46     sensitive << clock.pos();
47
48     cout << "Socket " << name << " constructed" << endl;
49
50   }
51
52 };
53 #endif // Socket_H
54
```

```
/**
 * Testbench class for I/O testing.
 */

#ifndef Testbench_H
#define Testbench_H

#include "systemc.h"
#include "globaldefs.h"

class Testbench : public sc_module {

  private:
  ifstream if0;
  ofstream of0;
  sc_biguint<128> axtoi(const char *hexStg);
  sc_uint<32> outBuffer[376];
  sc_uint<32> inBuffer[376];
  bool sendData;
  bool busy;
  int inLength;
  int outLength;

  public:

  sc_in_clk clk;

  ////////////////////////////////////////////////////
  // Connections for InputFU
  ////////////////////////////////////////////////////
  sc_out<sc_uint<BUSWIDTH> > ifuData;
  sc_out<sc_uint<BUSWIDTH> > ifuDataLength;
  sc_out<bool> ifuTrigger;
  sc_in<bool> ifuAck;

  sc signal<sc uint<BUSWIDTH> > sigIfuData;
  sc_signal<sc_uint<BUSWIDTH> > sigIfuDataLength;
  sc_signal<bool> sigIfuTrigger;
  sc_signal<bool> sigIfuAck;

  ////////////////////////////////////////////////////
  // Connections for OutputFU
  ////////////////////////////////////////////////////
  sc_in<sc_uint<BUSWIDTH> > ofuData;
  sc in<sc uint<BUSWIDTH> > ofuDataLength;
  sc_in<bool> ofuTrigger;
  sc_out<bool> ofuAck;

  sc_signal<sc_uint<BUSWIDTH> > sigOfuData;
  sc_signal<sc_uint<BUSWIDTH> > sigOfuDataLength;
  sc signal<bool> sigOfuTrigger;
  sc_signal<bool> sigOfuAck;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Read "packets" from input file.
   */
  void readInputFile();

  /**
   * "Receive" data from input file and write it to memory through InputFU.
   */
  void getPDU();

  /**
   * "Send" data coming from OutputFU to output file
   */
  void sendPDU();

  SC_HAS_PROCESS(Testbench);
  /**
   * Constructor.
   */
  Testbench(const sc_module_name& name, sc_clock& c) : sc_module(name){
    clk(c);

    ifuData(sigIfuData);
    ifuDataLength(sigIfuDataLength);
    ifuTrigger(sigIfuTrigger);
    ifuAck(sigIfuAck);

    ofuData(sigOfuData);
    ofuDataLength(sigOfuDataLength);
    ofuTrigger(sigOfuTrigger);
    ofuAck(sigOfuAck);

    if0.open("fi0.txt");
    if (!if0) cout << "TestBench: Cannot open file fi0.txt" << endl;
    of0.open("fo0.txt");
    if (!of0) cout << "TestBench: Cannot open file fo0.txt" << endl;

    SC_CTHREAD(readInputFile,clk.pos());
    SC_CTHREAD(getPDU,clk.pos());
    SC_CTHREAD(sendPDU,clk.pos());

    sendData = busy = false;

    cout << name << " constructed." << endl;
  }

  ~Testbench();

};
#endif // Testbench_H

```

```
#include "testbench.h"

sc_biguint<128> Testbench::axtoi(const char *hexStg){
  int n = 0;         // position in string
  int m = 0;         // position in digit[] to shift
  int count = 0;         // loop index
  //      unsigned long intValue = 0;  // integer value of hex string
  sc_biguint<128> intValue = 0;  // integer value of hex string
  sc_biguint<128> digit[129];      // hold values to convert
  while (n < 128) {
    if (hexStg[n]=='\0')
      break;
    if (hexStg[n] > 0x29 && hexStg[n] < 0x40 ) //if 0 to 9
      digit[n] = hexStg[n] & 0x0f;           //convert to int
    else if (hexStg[n] >='a' && hexStg[n] <= 'f') //if a to f
      digit[n] = (hexStg[n] & 0x0f) + 9;      //convert to int
    else if (hexStg[n] >='A' && hexStg[n] <= 'F') //if A to F
      digit[n] = (hexStg[n] & 0x0f) + 9;      //convert to int
    else break;
    n++;
  }
  count = n;
  m = n - 1;
  n = 0;
  while(n < count) {
    // digit[n] is value of hex digit at position n
    // (m << 2) is the number of positions to shift
    // OR the bits into return value
    intValue = intValue | (digit[n] << (m << 2));
    m--;   // adjust the position to set
    n++;   // next digit to process
  }
  return (intValue);
}

void Testbench::readInputFile(){
// reads the input file for interface0 and stores data in the inputbuffer0.
//when finishing, it enables the sendData signal to write the data inthe memory
  char line[41]=""; //reads one line of the input file
  char shortline[9]; // stores the firs 32 bits of the line
  char s[5]="0000";     // stores

  while(true){
    if (!sendData){
      while (if0.getline(line,40)){
        cout << name() << ": reading file fi0.txt" << endl;
        if (line[0] == '#'){
          //new datagram, always accompanied by the datagram length
          int i = 1;
          while (line[i] != '\0'){
            s[i-1] = line[i];
            i++;
          }
          s[i-1] = '\0';
          inLength = atoi(s);
          cout << name() << ": incoming datagram length on inteface 0, length " << atoi(s) << endl;
          for (int i = 0; i < inLength; i++){
            if0.getline(line, 40);
            strncpy(shortline, line, 8);
            //  cout <<"TestBench: line =" << shortline  << endl;
            inBuffer[i] = axtoi(shortline);
            cout << name() << ": line " << i << ": "  << inBuffer[i]  << endl;
          }//for
          sendData=true; //exits while
          //cout << "Testbench: enabling SendPacket[0]" << endl;
          //wait(1);
          break; //exits while
        }
      }//while
      if (if0.eof()) {
        cout << name() << ": EOF fi0.txt detected! " << endl;
        if0.close();
        cout << name() << ": fi0.txt closed" << endl;
        return;
      }
      //else cout << "TestBench: exiting while" << endl;
    }//if
    else {
      wait(1);
      //cout << "TestBench::ReadInputFile waiting for false" << endl;
    }
  }//while
}

void Testbench::getPDU() {
  //myId is the interface number
  const int myId = 0;

  while(true){
    if ( sendData && (!busy) ) {
      cout << name() << ": Incoming data on interface "<< myId << endl;
      // trigger the PPin for this interface
      // Also, write the length and the first data word
      busy = true;
      ifuTrigger.write(true);

      ifuDataLength.write(inLength);

      // Now wait for the PPin to acknowledge
      while(!(ifuAck.read())) wait(1);

      for (int j = 0; j < inLength; j++){

        ifuData.write(inBuffer[j]);
        cout << name() << ": interface " << myId << " writing through PPIN into memory, length "
             << inLength << " data " << inBuffer[j] << endl;

        wait(1);
      }

      // cout << "TestBench: finished storing in the memory for interface 0" << endl;

      sendData = false;
      ifuTrigger.write(false);
      busy = false;
      wait(2);
    }
    else {
      if (busy) cout << name() << ": PPin Busy! (intf" << myId << ")" << endl;
      else cout << name() << ": intf " << myId << " idle" << endl;
      wait(1);
    }
  } // while
}

void Testbench::sendPDU() {
  sc_uint<32> data;

  bool newData = false;
  while(true){
    cout << name() << ": checking for outgoing on intf 0" << endl;
      newData = ofuTrigger.read();

      if (newData) {
    ofuAck.write(true);
        cout << name() << ": Outgoing data on interface 0" << endl;
        wait(3);
        outLength = ofuDataLength.read();
        cout << name() << ": Length of output data is " << outLength << endl;
    for (int l = 0; l < outLength; l++){
          outBuffer[l] = ofuData.read();
          cout << name() << ": read " << outBuffer[l] << endl;
          wait(1);
        }
    ofuAck.write(false);
    //writing to file
    char str[9];
    sprintf(str,"#%d", outLength);
    of0 <<  str << endl;
    for (int l = 0; l < outLength; l++){
          sprintf(str,"%08X", (int)outBuffer[l]);
          cout << outBuffer[l] << " = " << str << endl;
          of0 <<  str << endl;
    }
    cout << name() << ": wrote interface file 0" << endl;
        wait(1);
     }
     else{
       wait(1);
     }
  } // while
}

Testbench::~Testbench(){
  cout << "Testbench destructed." << endl;
}
```

```
#ifndef TriggerSocket_H
#define TriggerSocket_H

#include "globaldefs.h"
#include "socket.h"
#include "systemc.h"
#include "string"

/**
 * TriggerSocket is a class modeling trigger socket of a FU in a transport triggered processor.
 */

class TriggerSocket: public Socket {

  private:
  /**
   * Array of IDs of this socket.
   */
  sc_uint<ADDRESSWIDTH> socketIds[MAXTRIGGERIDS];

  /**
   * Number of IDs.
   */
  int idCnt;

  /**
   * Read data from the bus on next cycle.
   */
  bool readData;

  /**
   * Index of the bus to be read.
   */
  int busNumber;

  /**
   * Temporary holder for opCode.
   */
  sc_uint<OPCODEWIDTH> tempOpCode;

  public:

  ////////////////////////////////////////////////////
  // Ports
  ////////////////////////////////////////////////////
  /**
   * Array of input ports for buses.
   */
  sc_in_rv<BUSWIDTH> inDataPorts[BUSES];

  /**
   * Output port to the FU.
   */
  sc_out<sc_uint<BUSWIDTH> > fuData;

  /**
   * Opcode output to the FU.
   */
  sc_out<sc_uint<OPCODEWIDTH> > opCode;

  /**
   * Trigger bit to the FU.
   */
  sc_out<bool> trigBit;

  ////////////////////////////////////////////////////
  // Functions
  ////////////////////////////////////////////////////

  /**
   * Decode addresses on address buses and perform actions if neccessary.
   */
  void decodeId();

  SC_HAS_PROCESS(TriggerSocket);

  /**
   * Constructor.
   *
   * \param name_ Name of this module.
   * \param trigIdCnt Amount of IDs.
   * \param ids Array of IDs for this socket.
   * \param opNames Pointer to string array containing operation names.
   */
  TriggerSocket(sc_module_name name_, int trigIdCnt, sc_uint<ADDRESSWIDTH>* ids, char* opNames = 0): Socket(name_){

    // <NOT SYNTHESIZABLE>
    // bind input ports to data buses and id ports to Dst buses
    Bus* bptr;
    for(int i = 0; i < BUSES; i++){
      bptr = Buscontroller::getBus(i);
      inDataPorts[i].bind(bptr->sigData);
      inIdPorts[i].bind(bptr->sigDst);
    }
    // </NOT SYNTHESIZABLE>

    if(ids[0] == (sc_uint<ADDRESSWIDTH>)0){
      idCnt = 0;
      // <NOT SYNTHESIZABLE>
      const char* n = name_;
      std::string socName = n;
      socName.erase(0,1);
      std::string opNamesS[trigIdCnt];
      for(int i = 0; i < trigIdCnt; i++ ){
        if(opNames == 0){
          socketIds[i] = Buscontroller::getSocketId(name_);
        }
        else{
          opNamesS[i] = opNames[i] + socName;
          socketIds[i] = Buscontroller::getSocketId(opNamesS[i].c_str());
        }
    idCnt++;
      }
      // </NOT SYNTHESIZABLE>
    }
    else{
      idCnt = trigIdCnt;
      for(int i = 0; i < trigIdCnt; i++){
        socketIds[i] = ids[i];
      }
    }

    readData = false;
    busNumber = 99;
  }

};
#endif // TriggerSocket_H

```

```
#include "trigsocket.h"

void TriggerSocket::decodeId(){
  if(trigBit.read()) trigBit.write(false);

  // read data if a data read was scheduled during last cycle
  if(readData){
    cout << name() << " reading..." << endl;
    fuData.write(inDataPorts[busNumber].read());
    opCode.write(tempOpCode);
    trigBit.write(true);
    readData = false;
  }

  // decode
  for(int i = 0; i < BUSES; i++) {
    int k = 0;
    while(k < idCnt){
      if (socketIds[k] == inIdPorts[i].read()) {
        cout << name() << " triggered with opcode " << k << endl;
        tempOpCode = socketIds[k] - socketIds[0];
        readData = true;
        busNumber = i;
      }
      k++;
    }
  }
}

```