Geänderte

INAUGURAL–DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich–Mathematischen Gesamtfakultät
der
Ruprecht–Karls–Universität
Heidelberg

vorgelegt von

M. E. Wenxue Gao
aus Liaoning, China

Tag der mündlichen Prüfung: 22 März 2012

Thema

Active Buffer Development in CBM Experiment

Gutachter: Prof. Dr. Reinhard Männer

Prof. Dr. Ulrich Brüning

Abstract

The DAQ (data acquisition) system of the CBM experiment in GSI is featured by the large data rate of about 1 TB/s and the high event rate of about 100 kHz. Such challenge is being a trend in high-energy physics experiments. The Active Buffer concept is applied to establish the proprietary buffering system for the CBM DAQ, which also provides active support in the event building. The project requires a modular framework and the work in this dissertation includes the development, testing and verification of FPGA modules for efficient data transfer, buffering and reconfiguration, as well as software for automatic transformation of HDL codes.

The central control device of the Active Buffer is a high-end FPGA. The storage kernel is a DDR2 SDRAM module of 512 MB. With a FIFO wrapper implemented with a small amount of FPGA resource, the local buffer on the Active Buffer board has both large-size memory space and standard FIFO access ports. To perform reliable and efficient data transfer from the read-out system to the computing nodes, a double-channel scatter-gather DMA engine over PCI Express is developed, which has 543 MB/s DMA read performance and 790 MB/s DMA write performance. Based on the same DMA functionalities, epoch marker indexing is tested in the Active Buffer system.

In terms of the large-scale FPGA design for the Active Buffer, verification plays an important role. For a secure function set on the transaction layer of PCI Express, HDL designs for the Active Buffer are verified by a rich combination of simulation environments. Upon this verification, the logic can be improved quickly and reliably. This is a sound foundation for the future Active Buffer development.

Against the link training issue in the PCI Express reboot and the license limitation of the Virtex4 PCI Express core, as well as for a modularized firmware architecture to the upper-level software applications, DPR (dynamic partial reconfiguration) technology is implemented on Virtex4, Virtex5 and Virtex6 FPGAs. Concurrent PlanAhead software with the partial reconfiguration license proves a competitive framework to do DPR. Attractive features are introduced by DPR experiments into the Active Buffer system. The fast reconfiguration is successfully done via ICAP over PCI Express operations, which leads to a general computing acceleration plug-in card scenario.

During the DPR experiments in high-frequency clock domains, the boundary logic between the static and the dynamic modules needs to be rewritten, which took more time and effort than expected. Therefore the HDL code reuse is studied to save the error-prone HDL rewrite work and to try out a new HDL development pattern. The HDL code reuse tool, *Logro*, transforms original HDL design into target HDL design with specified pipeline reorganization so that the DPR boundary logic is automatically and correctly decoupled. Encouraging results of *Logro* V1.0 are presented.

Keywords: CBM, DAQ, Active Buffer, FPGA, DDR SDRAM, PCI Express, DMA, verification, DPR, PlanAhead, ICAP, HDL reuse

Zusammenfassung

Die Anforderungen an das Datenerfassungssystem (DAQ) des CBM Experiments an der GSI sind mit einer Datenrate von 1TB/s und einer Ereignisrate von 100 kHz sehr hoch und stellen auch im Vergleich zu anderen Experimenten in der Hochenergiephysik eine Herausforderung dar. Bei der Datennahme wird daher ein aktiver Zwischenspeicher („active buffer“) eingesetzt, der durch eine Vorsortierung der Datenfragmente und eine intelligente Übertragung in den Hostrechner den Aufbau der Datenstrukturen zur Ereignisverarbeitung unterstützt. Das Projekt erfordert ein modulares Framework und die Arbeit umfasst die Entwicklung, Verifikation und Test von FPGA Modulen zum effizienten Datentransfer, zur Zwischenspeicherung und zur Rekonfiguration, sowie von Software zur automatischen Transformation von HDL Beschreibungen.

Die zentralen Bauteile dieses Zwischenspeichers sind ein leistungsfähiges FPGA zur Datenflusssteuerung und ein DDR2 SDRAM Modul mit einer Kapazität von 512MB. Durch eine spezielle Ansteuerungsmethode kann das Speichermodul zusammen mit den FPGA-internen Speicherelementen als leistungsfähiges, großes FIFO betrieben werden. Den Datantransfer vom Zwischenspeicher zum PC übernimmt eine spezielle DMA Einheit, die an den PCIe-Kern im FPGA angeschlossen ist. Die zwei DMA Kanäle arbeiten mit Scatter-Gather Unterstützung und erreichen beim Transfer zum PC 543 MB/s und in der Gegenrichtung 790MB/s. Die für die Vorsortierung wichtige Übertragung der Zeitstempel („epoch marker“) erfolgt ebenfalls mit einem DMA Kanal.

Die Verifikation ist eine wichtige Stufe bei der Entwicklung einer umfangreichen FPGA Anwendungen wie des aktiven Zwischenspeichers. Daher wurden die HDL Module der Funktionen für das PCI Express „transaction layer“ mit einer Reihe unterschiedlicher Simulationsumgebungen verifiziert. Auf dieser Grundlage können Verbesserungen an der Funktionalität schnell und zuverlässig umgesetzt werden, womit eine konsistente Weiterentwicklung gewährleistet ist.

Aufgrund der typischen PC-Architektur muss die PCIe-Einheit im FPGA bereits während des Startvorgangs funktionsfähig sein, wohingegen die eigentliche aktive Zwischenspeicherfunktion erst zusammen mit der entsprechenden Anwendungssoftware verfügbar sein muss. Strikte Modularisierung zusammen mit dynamischer, partieller Rekonfigurierung („DPR“) ermöglichen Veränderungen in der Zwischenspeicherfunktion zur Laufzeit. Ein weiter Grund für die Nutzung der DPR sind die Lizenzbedingungen der PCIe-Core-Implementierung mit Virtex4-FPGAs. DPR kann bei den FPGA Familien Virtex-4, -5 und -6 im Rahmen der „PlanAhead“ Software von Xilinx benutzt werden. DPR wird im Projekt im Sinne eines allgemeinen Coprozessors eingesetzt, indem die FPGA Konfiguration über die PCIe und die interne Konfigurationsschnittstelle („ICAP“) im FPGA nachgeladen wird.

Um DPR bei hohen Taktgeschwindigkeiten einsetzen zu können, muss die Verbindungslogik zwischen den statischen und dynamischen Modulen speziellen Anforderungen genügen. Da die manuelle Anpassung existierenden Module an diese Anforderungen aufwändig und fehleranfällig ist, wurde das Programm „*Logro*“ entwickelt, das HDL Beschreibungen mittels einer speziellen Pipeline-Neustrukturierung automatisch so transformiert, dass die DPR Anforderungen erfüllt werden. Mit *Logro* V1.0 wurden dabei gute Ergebnisse erzielt, die hier vorgestellt werden.

Keywords: CBM, DAQ, Active Buffer, FPGA, DDR SDRAM, PCI Express, DMA, Verifikation, DPR, PlanAhead, ICAP, HDL Reuse

Acknowledgement

During the last six years of stay in Mannheim for the CBM experiment, I have received enormous help from all people around me. I owe great thanks to them.

Prof. Dr. Reinhard Männer and Dr. Andreas Kugel have given and are giving me precious supervision and support into the project, building up the sound fundamental to research. Prof. Dr. Reinhard Männer and Prof. Dr. Ulrich Brüning contributed excellent advices and suggestions in improving the dissertation structure. PD. Dr. Joachim Gläß led me into the project with great patience. Andreas Wurz has thousands of wonderful ideas, contributes robust designs and invokes my mind open to solutions. When I have questions, they are always there. Christian Steinle, Markus Gipp, Dr. Guillermo Marcus, Jakob Haufe, Yuning Yang, Thomas Gerlach, Nicolai Schroer, Raul Fajardo, Michael Stapelberg and Philipp Schäfer helped me a lot in discussion and technical debugging as well as test programs. Indispensable PCI driver support is from Dr. Guillermo Marcus. People in the department of Prof. Dr. Ulrich Brüning (CAG Group) are active in cooperation and have provided efficient modules for the Active Buffer FPGA. Thanks go to Frank Lemke, Sven Schenk and Prof. Dr. Ulrich Brüning.

All colleagues in Lehrstuhl Informatik V are so kind and in such a warm atmosphere. Andrea Seeger, Dina Geppert and Christiane Glasbrenner are always ready to help. Towards the literal completion of the dissertation, Lei Zheng made proof-reading for most chapters with expert opinions.

Thanks go to people in GSI, Darmstadt and to people in KIP, Heidelberg University. Without them, almost half of the dissertation would be nowhere. They are Dr. Sergey Linev, Dr. Jörn Adamczewski-Musch, Dr. Hans-Georg Essel and Dr. Walter F.J. Müller in GSI and Dr. Norbert Abel, Sebastian Manz and Prof. Dr. Udo Kebschull in KIP.

My family supports me with heart and love. There stand my mother Yumei Xiao, my father Jimei Gao, my sister Wenying, my wife Bin Xu and my sweet daughter Yina.

I wish my best to all people happy and healthy, mentioned and unmentioned. Thank you all the way.

Acronyms

ABB	Active buffer board
ASIC	Application specific integrated circuit
CBM	Compressed baryonic matter
CPLD	Complex programmable logic device
CRC	Cyclic redundant code
DABC	Data acquisition backbone core
DCB	Data combiner board
DMA	Direct memory access
DPR	Dynamic partial reconfiguration
DW	Double-word (32-bit data)
ECRC	End-to-end CRC
FAIR	Facility for Anti-proton and Ion Research
FEE	Front-end electronics
FPGA	Field programmable gate array
GT	Giga-transfer
GTP	Gigabit transceiver pair
ICAP	Internal configuration access port
LCRC	Link-by-link CRC
MGT	Multiple-giga transceiver
MPRACE	Multi-purpose reconfigurable accelerator/computing engine
ROC	Read-out controller
SFP	Small-factor plug

Contents

Appendices

Chapter 1 Introduction

1.1 High-energy physics (HEP) background

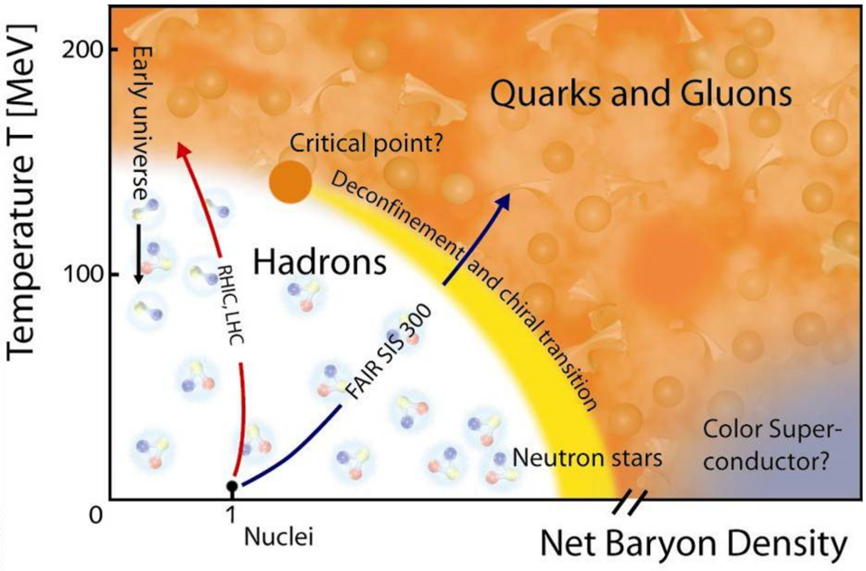
Physicists have been trying to widen their vision for thousands of years, spatially in two extreme directions, the microscopic and the macroscopic. Concerning the extreme microscopic physics research today, the deeper humans want to see into the matter, the larger effort is made to reach that scope. Observations on the particle level are different than those in the human world. In the sub-atomic scale, this is generally accepted and evidenced by higher and higher reaction energy found in today's accelerators, for example, LHC (**L**arge **H**adron **C**ollider), SLC (**S**tanford **L**inear **C**ollider), FAIR (**F**acility for **A**nti-proton and **I**on **R**esearch), and so on. Frontier scientists get the knowledge they want out of these giant machines, dozens of kilometres in size and several TeV in the reaction energy level. [1]

Concerning the extremely macroscopic physics, there are huge telescopes situated on the earth surface and floating in the space, for example, the Hubble Space Telescope and the being-built Giant Magellan Telescope. [2] [3] They observe the huge-size images of the universe and record the evolution of galaxies. Such macroscopic observations differ from the experiments in the HEP accelerators in that they do not affect the observed objects. However, the accelerator experiments are trying to reveal the secret of the initial universe, or the internal state of the extremely dense neutron stars. The microscopic and macroscopic physics are asymmetrical and different in approaches, although they have influence upon each other.

Besides the energy level an accelerator can reach, the target region in the phase diagram is also an critical factor during experiments are constructed. Different equipments serve different scientific purposes. LHC is mainly designed for researching the Higgs particles. [4] [5] FAIR in Darmstadt, Germany targets on anti-proton and ion research. [6] Due to the cost consideration, an accelerator normally feeds multiple experiments, categorized by regions in the phase diagram that an experiment focuses, as in figure 1-1. For instance, the CBM (**C**ompressed **B**aryonic **M**atter) experiment in FAIR project is focused on high temperature and high baryon density; the $\overline{\text{P}}$ANDA (anti-**P**roton **AN**nihilation at **DA**rmstadt) experiment is intended to study weak and strong forces, exotic states of matter and the structure of hadrons. Of course there is much space untouched in the phase diagram, which provides us with attractive possibilities. [7] [8] [9]

Not only the spatial size of particles under research goes extremely tiny, but also become their lives tremendously shorter. For example, lifetime of Sigma particles is from 1.48 x 10^{-10} s ($\mathbf{\Sigma}^-$) to 1.8 x 10^{-23}

s (Σ^*). [10] Such ephemerality has strong influence on the detector design as well as on the hit data processing.



Source: http://www.gsi.de/forschung/fair_experiments/CBM/1intro.html

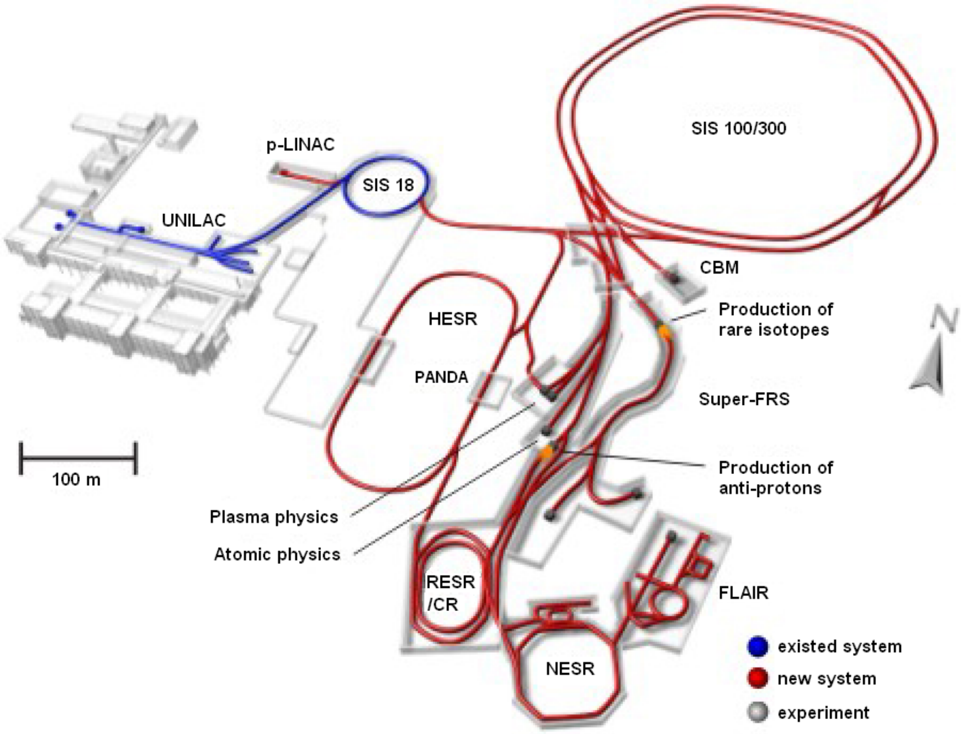
Figure 1-1 QCD Phase diagram for FAIR

In the particle physics experiments today, matter reactions are not really seen with human eyes, as the physics around one hundred years ago, but are *seen* through the sophisticated instruments. And thus, the observer of the under-size world has to compensate the effect from himself, according to the law of uncertainty. Thanks to the material, electronics and information technologies, people are able to build sharper ***eye***s to detect the hyper microscopic events. And at the back-end, DAQ (**D**ata **A**c**Q**uisition) system serves as the storage, archive, and event selection part.

CBM experiment is a part of the FAIR system, located in GSI, Darmstadt, Germany. It is proposed to explore the QCD (**Q**uantum **C**hromo **D**ynamics) phase of high temperature and high baryon density. Researches such as simulating the initial stage of the big-bang and studying the centre status of neutron stars may benefit from its results. CBM experiment is featured by huge amount of data flow density of about 1 TB/s into the DAQ system. This characteristic leads to a self-triggered architecture for the DAQ, which will deliver possibly more events and higher data flow to the back-end processing modules. [11]

The accelerator of FAIR in development is SIS100/300, as figure 1-2 shows. It is expanded from the existing equipment, SIS 18, and is planned to feed several different experiments, such as CBM, $\overline{\text{P}}$ANDA, etc. The double-ring facility of about 1100 meters in diameter is supposed to generate intense high-energy ion beams, out of which secondary beams can be produced. CBM and $\overline{\text{P}}$ANDA both use self-triggered approach and the trigger selection is formally done in the compute nodes. [12] [13] Comparatively, most experiments at LHC in CERN are implemented in level-trigger fashion, for example, the ALICE (**A** **L**arge **I**on **C**ollider **E**xperiment) and the ATLAS (**A** **T**oroidal **L**HC

Apparatu**S**) experiment at CERN. [14] [15]



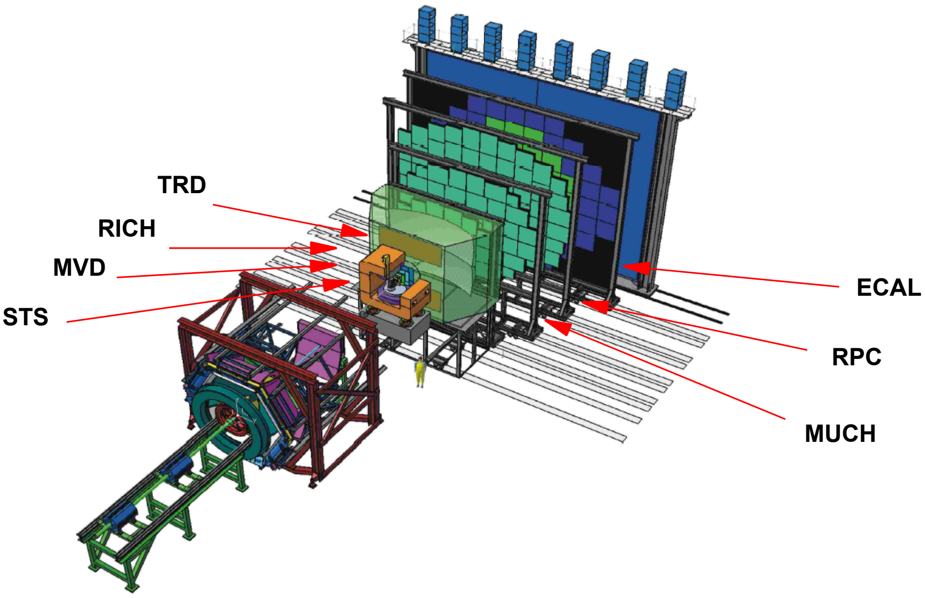
Source: http://www.gsi.de/portrait/fair/beschleuniger.html

Figure 1-2 CBM in FAIR

As shown in figure 1-3, CBM experiment is supposed to consist of the following major detectors, [11]

- — Silicon tracking system (STS) and micro-vertex detector (MVD)
- — Ring imaging Cherenkov detector (RICH)
- — Transition radiation detectors (TRD)
- — Muon detection system (MUCH)
- — Resistive plate chambers (RPC)
- — Electromagnetic calorimeter (ECAL)

Dependent on the sizes, distances to the beam source and physics purposes, the data rates and distributions vary a lot from detector to detector. For example, the STS has much higher data rate than the RPC. Within a single detector, the hit densities also vary according to radius and angle. This is simulated and experimented and has impact upon the DAQ strategy.



Source: CBM Experiment Technical Status Report, January 2006

Figure 1-3 CBM detectors

Data coming from the detectors are processed by machines in order to reproduce the physic events. Event building, either with software or with hardware, is the central task for the DAQ. Machines do most of the work, of course final decision is in man's hand. Information technologies support physicists with faster and more accurate recognition and identification of new physics. The DAQ portion is being naturally enlarged in the HEP field.

For example, the track finding research for CBM STS takes advantage of 3D Hough transform algorithm and tries to fit the algorithms into FPGA devices. [16]

Before the final CBM set-up, many small beam tests are carried out for different modules of the DAQ system. For example in December 2010, a 2.1 GeV beam test was done in COSY, FZ Jülich to test the updated development progress and to find out hidden bugs as early as possible. Similar beam tests have also been made in CERN.

1.2 CBM DAQ characteristics and the Active Buffer

The nature of CBM experiment is the huge data rate of about 1 TB/s due to the high reaction rate of about 10 MHz. If the average bandwidth of each cable, either copper or fibre, is 10 Gbps, the number of links should be about 1000. For the technology standard in 2011, Xilinx Virtex5 and Virtex6 LXT FPGA series can drive 5 Gbps link with its GTP. [17] [18] Therefore the 1000-link estimation is modest.

To process such a data density in the DAQ system, the technologies today and even in the next few years have to compromise between trigger speed and the event granularity. CBM experiment wants a fine observation of the very rare physics events and it does not want to sacrifice the precision.

Therefore CBM is trying a weak-triggered fashion in DAQ, in which the level triggers are not explicit. This brings challenge for DAQ. On the other hand, as a benefit of the triggering pattern, the DAQ latency is not critical to the CBM experiment.

Simulations of the experiment reactions at detectors are deliberated. They help to evaluate the detector geometry and the material utilization as well as the data flow distribution. The detector design and the DAQ data flow distribution are much dependent upon the results of these simulations. [19]

Most of our design is based upon the simulations for STS events. Such simulations are helpful because they take advantages of the newest physics progress and are proven in previous experiments. [20] According to the STS simulation, we set the throughput for every buffer sub-system as 5 Gbps, which can be fit into 2 fibres.

However, simulations cannot tell us everything about the physics output, otherwise we would not have to build huge accelerators and detectors. As we are to find new physics, we should preserve enough redundant space in our design, to cover as many exciting corners as we can. For example in the DAQ system, the event rate should not be under estimated.

Figure 1-4 shows the DAQ chain for CBM. The DAQ system receives detector hit data from the FEE (**F**ront-**E**nd **E**lectronics) and the data travel through the ROC (**R**ead-**O**ut **C**ontroller), DCB (**D**ata **C**ombiner **B**oard), ABB (**A**ctive **B**uffer **B**oard) and reach the software framework DABC (**D**ata **A**cquisition **B**ackbone **C**ore). ROC reads out the hit data from FEE or configures FEE. DCB organizes the data from ROC and manages the synchronization and possesses radiation-hard capability, which the ABB does not have.

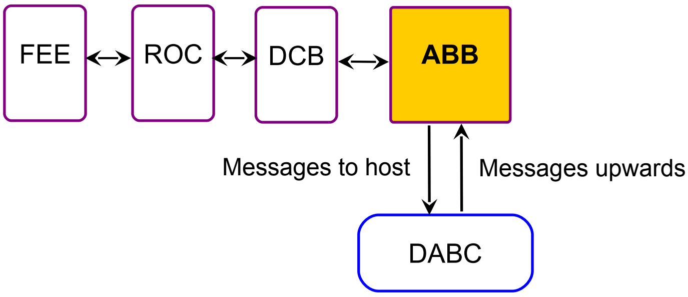


Figure 1-4 CBM DAQ chain

The Active Buffer is the buffer element in CBM experiment for DAQ. Namely, it is not only for passive buffering of incoming hit packets, but also manages to carry out part of the first-level event selection (FLES) for the experiment. It is the last stage in the hardware portion.

Its first major function is to buffer the incoming hit packets over 2.5Gbps or 5Gbps optical links. Logically, the Active Buffer stands between the DCB and the back-end host computer. It carries out also event building, as the second major function. Taken as a black-box, it manages miscellaneous classes of outgoing and incoming packets, whether data packets or control messages. Control messages can be CTL (control) messages or DLM (**D**eterministic **L**atency **M**essage). CTL messages are for the DAQ control purposes to the previous stages and DLM are for measuring and gathering necessary latency information for the DAQ system. The Active Buffer behaves similar to a dual-port memory, so we are trying to make a dual-port memory emulation logic on our Active Buffer board upon the commercial SDRAM memory modules.

From the FEE to the storage, there are many options to build the data path. Storage is mostly attached to a computer. High-speed data paths must be built between these hosts and the FEE. Considering the radiation-hardness and the distance between the front-end and the back-end, the data are firstly read out by the ROC. On account of fibre cost, data are to be aggregated and relayed in the middle by the DCB. And then the aggregated data stream goes into ABB. After ABB, software suite will do online-analysis, monitoring and supervising. Afterwards, the minimum amount of data are stored into the archive net. The ABB is the juncture point between hardware and software, and we develop a high-performance DMA (**D**irect **M**emory **A**ccess) engine over PCI Express Gen1.

In the future FAIR geography, the IT centre building locates about 350m away from the CBM building. [19] The cable cost between these two spots is a factor in data path development. Compared with copper wires, fibre transmission is better in spatial budget, noise tolerance, power consumption, etc. Therefore, optical fibre is the top choice in building the distant data route. Optical fibres and connectors match the requirement of high-speed data transport. Multiple-Gbps fibres with reliable transmission distance of multiple kilometres are purchasable and mature products.

To meet the requirement of the high rate of hit data, the Active Buffer system of CBM experiment takes the challenge of large, fast and intelligent memory design. A directly handy solution is to use the popular and favourable SDRAM modules as the memory kernel, which go easily to Gigabytes in size with benign price, meanwhile running at 2.5~5 GB/s bandwidth. However, due to the single address bus structure, such memory module cannot be written and read concurrently. The concurrent write and read is a fundamental requirement of the Active Buffer design. In this sense, a simplified vision of CBM DAQ buffer system can be modelled as a FIFO. Of course, the ultimate model is much more complex and intelligent.

For the first version, we work out a FIFO-style memory controller (wrapper) around the DDR-2 SDRAM module, which enables the concurrent FIFO standard accessing, without sacrificing the port bandwidth. In this way, 512 MB FIFO with concurrent speed of over 2 GB/s for both write and read is possible. This FIFO wrapper can be scaled to larger sizes, e.g. 1GB, 2GB and so forth. Such development is going on, targeting a really active and intelligent buffer system for the CBM DAQ. Compared with the IC market, FIFO device size is always lower than people expect. For example, the largest IDT FIFO is only 18 Mb for the year 2011. [21]

Dual-channel memory technology in current PC platform is a similar attempt to improve the throughput of the memory modules, where a pair of identical (DDR mode) SDRAM modules is used to double the memory throughput. Dual-channel memory gets support from the chip-sets and relieves the system bottleneck. [22]

1.3 Fast data path

The Active Buffer is implemented to transfer the high rate of data flow to the host memory space. At the most beginning of the prototype beam tests, Gigabit Ethernet and InfiniBand were used to deliver the data to the back-end due to their robustness and availability of software frameworks. As the development advances, DCB and ABB join in. A prominent advantage of optical communication in CBM experiment is the quietness and noise tolerance. [23] [24] Ethernet is a general network, but we do not need many features of it, such as retransmission, WAN (**W**ide **A**rea **N**etwork) address resolution, etc.

Interconnect technologies such as PCI Express, HyperTransport, InfiniBand are proving technical perspectives in DAQ system. Almost every corner in the information technology is well driven by the Moore's Law. These data-path-dominated technologies involve also a lot in the integrity of the data content to provide a reliable link.

Among the high-speed interconnect technologies, PCI Express is a good choice for the ABB. PCI Express, usually abbreviated as PCIe or PCI-E, is a computer expansion card standard that is intended to replace legacy PCI, PCI-X and AGP (**A**ccelerated **G**raphics **P**ort). Its key difference apart from those older standards is the point-to-point serial link on the PHY layer. This makes it very easy to expand the bandwidth by multiplying link number. In this thesis, we use 4 PCI Express Gen1 lanes, which delivers 2.5 GT/s per lane. Another advantage of PCI Express is that it is software-compatible with older PCI drivers. This feature accelerates its occupation in the market. [25]

PCI Express devices build the bridge from the hit data links to the host nodes. They transfer data between the two high-speed buses, the optical and the PCI Express. So the protocol translation and flow control is implemented on such devices.

Another competitive candidate can be HyperTransport, which has also scalable performance and has no demand on license fee. [26] [27]

If every data over these data paths are moved by the host CPU (**C**entral **P**rocessing **U**nit), the host resource will be exhausted in PIO (**P**rogrammable **I**nput-**O**utput) operations. Although PIO mode requires least firmware support, its performance is quite low. Typical PIO write performance in our Linux system is measured about 30 MB/s and PIO read performance only about 3 MB/s. For sake of higher performance over fast data paths and lower host CPU load, DMA mode is generally adopted in the data path card. For example in our tests, the DMA engine over 4-lane PCI Express Gen1 delivers over 700 MB/s for DMA write and over 500 MB/s for DMA read. A difference between these two modes of data moving is the size-dependency. Generally larger DMA size has higher performance but the PIO performance almost does not change according the transfer size.

In terms of the integrated circuit (IC), customized computing favours programmable devices. Data-path-dominated research and development often benefit from FPGA (**F**ield **P**rogrammable **G**ate **A**rray) technologies. High-end programmable logic devices are FPGAs, which provide fast development for complex logic projects. Internal structure of the FPGA is being improved for better timing performance and lower power consumption. High-speed differential serial transceivers are paid great effort in development. 10 Gbps transceiver modules are announced by Xilinx and by Altera. [28] [29] The Active Buffer is implemented in Xilinx FPGA, initially in Virtex4 FX40 for the first version, later on Virtex5 LX110T for the second version.

As a natural extension of the embedded system research, integrating processors into programmable devices has accumulated valuable experience. Xilinx has previously PowerPC into VirtexII Pro series and Virtex 4 FX series. Later on till recently, MicroBlaze takes over the fashion for Xilinx FPGA. [30] [31] Altera has NiOS II processors in their high-end FPGAs. [32] The corresponding software development suites are provided for these embedded processors. For example on the ROC board, the embedded processor (PowerPC) of the FPGA (Virtex4 FX20) is used to manage the read-out behaviour and Ethernet access port.

As debugging approach, Xilinx has ChipScope and Altera has SignalTap II. Such structures give the developer a very good approach to check the internal timing. [33] [34] In the Active Buffer development, the ChipScope facility shows many cases that were not covered by the simulation, and therefore, helps to improve the verification.

1.4 Dynamic partial reconfiguration

The DMA development was done in the Heidelberg University and the soft PCI Express core for Virtex4 FPGA was licensed to research in universities. Therefore GSI cannot use the design, although the logic implementation in the FPGA is transparent to GSI. A solution for such situation came from the dynamic partial reconfiguration (DPR), which encapsulates the protected module (PCI Express core) and provides the final user with dynamic non-licensed modules. With the progress in Xilinx FPGA, PCI Express core has been upgraded to hard-macro in Virtex5 and Virtex6, so that the license problem is no longer there for GSI. However, the DPR technology is still benefiting the high-end FPGA projects.

Given a type of FPGA, the resource is a limited parameter, which has influence not only on price, but also on the board size and power consumption. In chapter 6 we will see that our project is confronted with this universal problem.

DPR is an attempt to mimic the hardware system to a software system. Due to the nature of hardware (gate-based model, wire latency, power-dependence, etc.), the hardware system cannot be so flexible as a software one, in which individual module is renewed just by recompile and reload. Software is supported by the operating system. Hardware cannot have such supporting system to play flexibility on it. However, with recent progress in Xilinx DPR software suite such as PlanAhead, the implementation of DPR becomes much easier.

ICAP (**I**nternal **C**onfiguration **A**ccess **P**ort) is available in Xilinx FPGA families, which makes it possible to load a partial bit file to a reconfigurable module on-line via high-speed general paths, such as HyperTransport, PCI Express and so on. [35]

DPR application has to cope with the boundary clearance between modules. Former DPR projects with Xilinx FPGA used bus-macros to isolate the boundaries and nowadays the bus-macros are no longer needed, while the boundary decoupling is strongly recommended. [36]

In DPR we adapted logic design for bus-macros or for boundary decoupling, both expecting HDL reuse. Without boundary processing, the timing convergence goes mostly to fail across partitions.

If an FPGA application contains time-mutexed modules, DPR can generally save the resource and further make place for more logic functions in one device.

In most recent years, Altera also starts to support dynamic reconfiguration and partial reconfiguration in their FPGAs. This is a good sign for the DPR field, since two FPGA giants are sharing a vivid world in information technology. [37]

1.5 Logic design reuse

In the Active Buffer logic development, both VHDL and Verilog HDL are used. And in the DPR experiments, the VHDL codes had to be rewritten for the boundary processing. Although the rewrite was so straightforward and seemed to be simple enough, it took much time to get the rewritten design working as the original one. To improve the efficiency of similar rewrite practice, HDL-level code reused is studied in this thesis.

Code reuse in HDL design is wanted especially when pipelines are to be inserted into original design to compensate the application of synchronous bus-macros/boundary units between the modules in DPR experiments. Also, such reuse is evidenced by the fact that most of HDL design are somehow overlapped in functions, but hard to be reused by each other.

The two conventional mainstream HDL are VHDL and Verilog HDL. VHDL was advocated by the U.S. Department of Defence for digital circuit documentation in 1980s. It is a strongly typed HDL. It helps the synthesizer to check the syntax with ease. The user is well regulated to prevent typing errors, although it is not case-sensitive. [38] Accellera is responsible for making the VHDL standards and the most recent VHDL revision (IEEE Standard 1076-2008) was released in 2009. [39]

Verilog HDL was originally invented as a simulation language in around 1983/1984 by Phil Moorby in the company Gateway Design Automation (which was in 1990 purchased by Cadence Inc.). Cadence has now the right of Verilog HDL as well as Verilog-XL. Verilog HDL reference more features from C and it gives the user more flexibility. The up-to-date Verilog HDL standard is Verilog 2005 (IEEE Standard 1364-2005). [40]

Although these two major HDL's came from different area, their purposes are quite similar. However, VHDL covers broader range of digital design and Verilog HDL works better in simulation.

There is also HDL brothers such as ActiveHDL, JHDL. [41] [42] SystemC, SystemVerilog, etc. came into being with the anticipation of enhanced aspects, e.g. ease of verification, good usability for software developers. [43] [44]

Design tools convince the scientists with reliability and fast development cycle in the EDA (**E**lectronics **D**esign **A**utomation) field. Design suites of Cadence, Synopsis, Xilinx, Altera, Mentor Graphics, etc. update themselves more than once a year. Design, verification as well as project management are better handled. [45] [46] [47] [48] [28] [29] [49]

All the effort is to make the digital hardware design easier to average engineers. Out of commercial reasons as well as the development tradition of a group, some designs are in VHDL, some in Verilog, and some in SystemC or SystemVerilog for verification. Actually an engineer which uses one design language does not have to endeavour much effort to exploit the advantage of another one. The EDA world does not need so many languages because they just establish unnecessary barriers among engineers or groups and bring confusion to beginners. However, no one single language is so perfect that it can eliminate all others, just similar to the software world.

As the mainstream HDLs today, neither VHDL nor Verilog HDL concerns the "dynamic" reuse of the source code, as we experienced in the DPR projects. HDL codes are created, debugged, verified, but reuse of the previous codes sometimes even takes the same long time as to redesign it all over again from scratch.

The HDL code reuse study in this thesis is to prove the feasibility of code reuse and to suggest the DPR tools to integrate part of such functions into their frameworks. A DPR user should focus on the partitioning of the function modules and their cooperation, instead of the pipeline shifting or elaborating.

Reusability, reliability and readability are top favourable properties of a design code in electronics. Code reuse, or HDL code reuse, has been considered and studied for a long time in EDA world. Plenty of IP (**I**ntellectual **P**roperty) cores are available from electronic companies and from the open-source community. These cores, or macros, are made with great verification effort and minimized bugs before being released. Not all IP cores save financial cost, but they do save time for development. Suppose to redesign an Ethernet MAC controller, even a skilled engineer cannot complete it within one week, but the existing IP core may work in just a work day. However, most of the progress is limited in the "static" reuse scope, where the original design is literally kept in the new design as a block. The reusability is achieved by the pre-paid development time to make the IP cores reusable. [50] The reuse technology is actually reuse requirements rather than reuse tools. Reusing an arbitrary (syntax-correct) HDL design has not been explored. Such study is not directly related to the Active Buffer development, but it has general meaning for the logic designs in the future.

1.6 Organization of the dissertation

This dissertation is to design a buffer system for the CBM DAQ, which not only stores the events, but also makes the necessary acceleration for event building. The questions expected to be answered per chapter include,

Chapter 2, how to build the buffer with enough large size and sufficient bandwidth?

Chapter 3, what kind of data path should be applied to the interface between the network and the host and how to build it?

Chapter 4, how about the performance of the designed buffer system?

The focus is on the Active Buffer design and some related issues aroused from its development. Thereby another two questions might appeal to the reader,

Chapter 5, how can a firmware module have the dynamic loading flexibility as a software module?

Chapter 6, how to accelerate the logic development in terms of HDL reuse?

Chapter 2 The Active Buffer system

2.1 Introduction

As mentioned in chapter 1, the DAQ system of CBM experiment is characterized with the weak-triggered pattern due to its huge amount of data flow, especially for data-intense detectors such as TRD, STS. The trigger decision is distributed to a wide range of units along the DAQ chain, in order to lose as few events as possible.

Concerning the large data rate into CBM DAQ system, we need a DAQ chain to get the data off the FEE and move them to applications, the DABC. This chain is divided into three parts, ROC, DCB and ABB, which are all equipped with FPGAs because integrating them all into one device is not practical, as shown in figure 2-1.

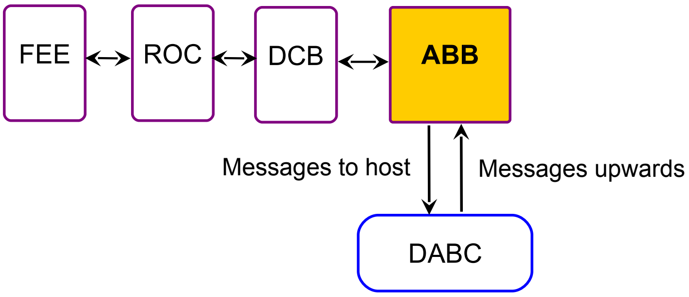


Figure 2-1 DAQ chain diagram (duplication of figure 1-4)

Figure 2-2 is a photo with the major DAQ chain elements, taken from CBM COSY beam test in December 2010 in FZ Jülich. In the set-up, we use 2 DCBs connected to 2 ABBs (plugged in the PCs), and 8 ROCs all connected to DCBs.

DAQ system and FLES (**F**irst-**L**evel **E**vent **S**election) system together get the data off the detectors, process the events, and store them in the back-end storage media, in format which is eligible to the physicists. FLES is logically placed after the DAQ section. It makes initial event building and selecting the epoch boundary out of the DAQ data stream.

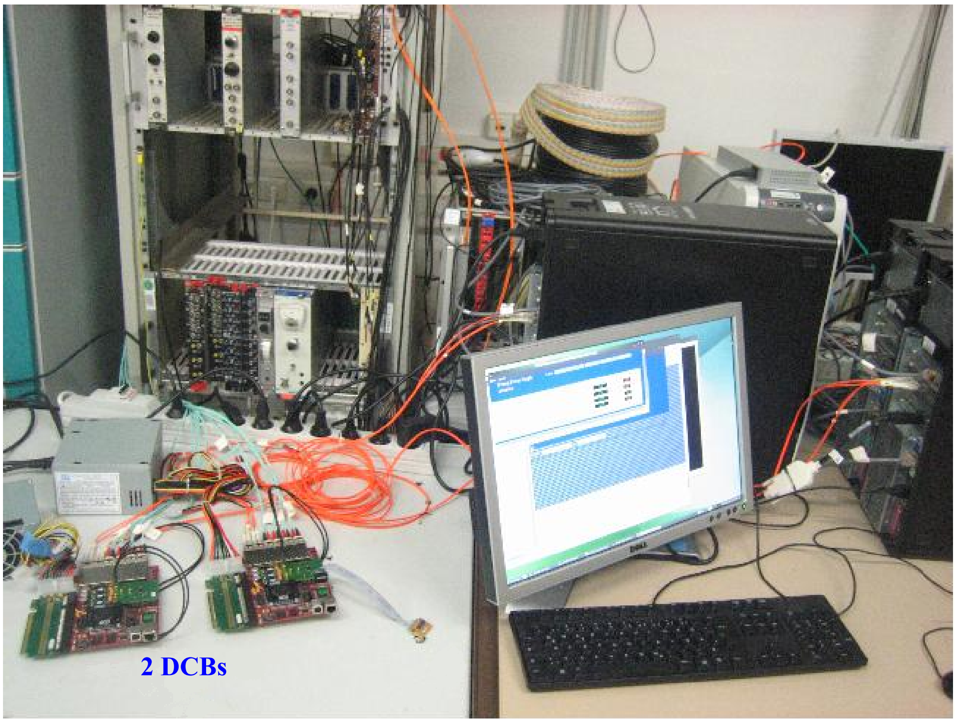


Figure 2-2 DAQ chain set-up in COSY beam test December 2010, FZ Jülich.
ROCs in the cave (not in this photo), and ABBs in the computer's PCI Express slots are all connected via the orange fibres to the 2 DCBs on the desk.

Between the DAQ and FLES is the interface component, the Active Buffer board (ABB), as shown in figure 2-4. Before ABB, the DCB manages the data combining, link balancing, as well as the network protocols. The data stream arriving at ABB is much smoother and more balanced, so that the links can be well utilized. In this sense, ABB is not only a buffering system, but also an event-building assistance system. The content of the data stream is moderately analysed for the sake of the FLES/DABC convenience. FLES is made up of computer farms and the work in it is organized in the software packages. [51] [52] [53] [54]

Large buffer with event building assistance is a scenario in the ABB development. The buffer pattern can be RAM or FIFO. The event building is ideally epoch rearrangement in the DAQ system, which is namely "*active*". However, in the starting phase of the project, the event building function is very rudimentary, and most event building is done by the software and hardware provides the indices to help the software sorting the data packets.

The size estimation for the Active Buffer can be only roughly done with the epoch duration and event building requirement. The epoch length is about 10 μs and the input data rate via the two SPFs is 500 MB/s (5 GT/s). If an event building takes about 1000 epochs, the data amount should be larger than

$$500 \times 10 \times 1000 = 5 \text{ MB}.$$

We should reserve certain leeway for the software application as well as for the future event building development, so a buffer structure of over 16 MB should be the minimum requirement on the Active Buffer size.

In the initial phase, we used to implement a 128KB built-in FIFO in Virtex5 FPGA as the Active Buffer prototype. In the software test, this size turned out to be too small. Concerning the limit of FPGA BRAM resource, built-in FIFO should also not be the final solution.

2.1.1 Active Buffer

The Active-Buffer kernel is illustrated in figure 2-3, which uses DMA engine over PCI Express as the data path.

As shown in figure 2-3, the epoch marker indexing module is designed as the event building assistance module. Some basic settings for the event building are considered to be stored into the Flash on the board, but it is not a must (as denoted with the dashed line).

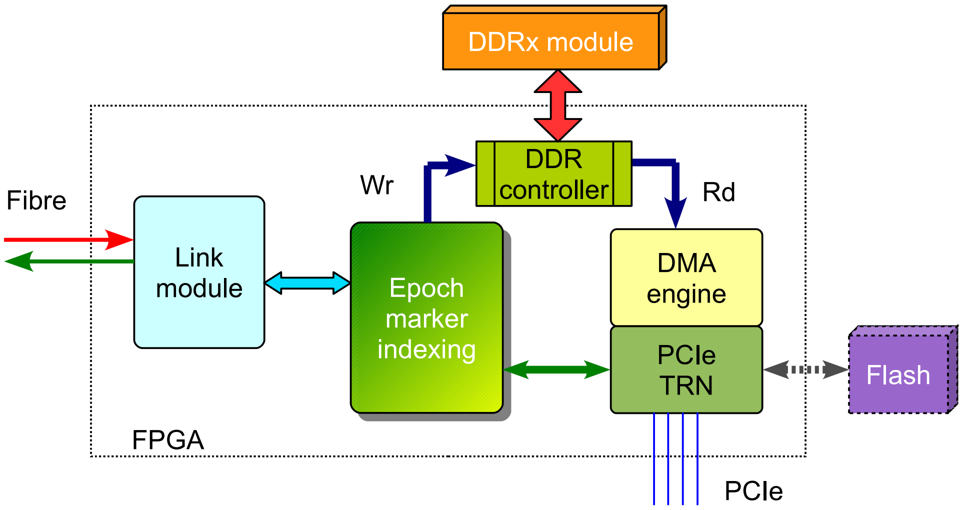


Figure 2-3 FPGA modularization for event building

One option to implement the ABB function in figure 2-1 is with software nodes. Another option can be a system without PC. The data stream is buffered, filtered, and transferred to the storage. Such architecture (without PC) is available for a more succinct system and takes smaller space. However, the processing nodes will nevertheless be installed. And that is why we take the advantage of the standard data path specification, PCI Express. Other standard candidates can be HyperTransport, InfiniBand, or Gigabit Ethernet.

Located in such a position, the ABB function must include high-speed links from the DAQ and high-performance paths to the FLES. A reasonable choice for this combination consists of fibres and PCI

Express lanes. And to be suitable for this function, the central device on ABB should be high-end FPGA, which can be configured to transfer high-speed customized data packets.

As figure 2-4 shows, the ABB sits between DAQ and FLES and its functional partitioning in terms of hardware and software is blurry.

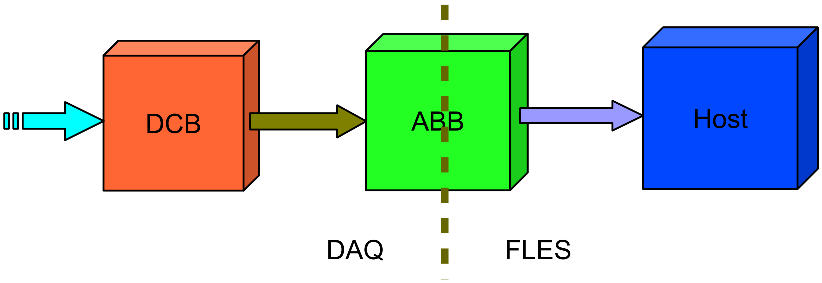


Figure 2-4 ABB between DAQ and FLES

Event building is the major purpose for the Active Buffer. Hit data coming from multiple ROCs are merged (combined) somehow by the DCB, before they land on the ABB. If hit data are reordered or filtered in the host memory, the time cost of the host CPU will be higher than the system, in which the event building is done in hardware. On the other side, the ABB board has enough resource to carry such tasks out. What the host needs to do is to set some rules during the initialization phase and then the FPGA on ABB can accordingly execute the reordering and filtering before transferring the events to the host memory (via DMA). These rules are, from the present vision, some parameter registers that contain control information about the event building policy. These registers of rule primitives should be set in a way of simplification and uniqueness (no ambiguity). They can be static, being defined at the most beginning of an operation. Or they can be dynamic, and the FPGA gathers the event building strategy during beam.

The hit packets from different sources (chips, channels, etc.) are combined through the DCB links and arrive at ABB. The Active Buffer (event buffer) can be 9× bits (e.g. 72-bit) wide with additional framing information, but the software accesses it only in 8× mode (e.g. 64-bit). The framing information is missing to software. If these packets are written into the host memory without pre-processing by the ABB card, special effort is expected for the software to distinguish events/epochs out of its DMA buffer. So it is better for ABB to dispatch the hit packets before making upstream DMA.

The event building involves analysis over the incoming hit packets. Information in the packets such as time stamp, channel number and chip number are the candidate tags to be analyzed. The hit packets will be first dispatched to the peripheral memory and afterward, transferred to the host memory. The dispatching is what the event building module will do. And the original DMA engine is categorized as transport module (data moving).

For current configuration, the fibre link runs at 2.5 GT/s, and the PCI Express Gen1 4-lane provides 8 Gbps (10 GT/s) peak bandwidth per direction. We use Virtex5 LX110T as the FPGA device, which has 16 tiles of GTPs, for SFP interfaces and for PCI Express serial interfaces. Each ABB has two SFPs. So the Buffer should transfer at least 4 Gbps per direction.

Another reason to choose PCI Express is the availability of the stable open-source PCI driver under Linux in our research group. This driver and the companioned library have been integrated into the DABC suite and works reliably in the real DAQ experiments.

The data density varies much from detector to detector. Our DAQ system (including FLES) is mainly targeted on the STS, which has the largest data flow rate among the CBM detectors.

2.1.2 Traffic classes

Hit, control and synchronization data packets travel through the DAQ system. The data packet format is shown in figure 2-5. [55]

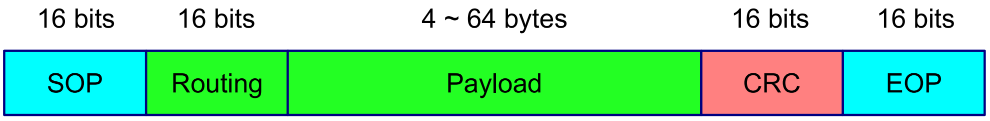


Figure 2-5 Packet format in CBM DAQ system

Further, they are categorized into three traffic classes. [56]

2.1.2.1 CTL protocol

The CTL traffic class is responsible for the coordination between the front-end and the back-end. It does not involves high data rate, but helps to provide the reliable transportation of control and monitor messages. This class is specially implemented to avoid communication crash due to message loss. CTL is in low frequency and needs no buffer for it. Decoding and executing is enough.

One of the important CTL packets downstream is the epoch marker, which defines the epoch boundary.

CTL message tests have been well proceeded because they are the management for almost all DAQ activities. Application-level tests for CTL message latencies result in 0.138 ms for write-read and 0.053 ms for read.

2.1.2.2 Sync mechanism (DLM)

DLM (**D**eterministic **L**atency **M**essage) is a lower-level message class. It is sent periodically to reach the parts that need synchronization. DLM messages have been very fundamentally tested and they are supposed to be further accomplished in terms of the protocol. DLM occupies only a very small portion of the overall bandwidth.

2.1.2.3 DAQ class

DAQ data is the major part of detector data flow and takes over 90% of the total bandwidth. DAQ class needs a large buffer system. When 2 ROCs are connected, DMA performance in transferring DAQ packets is about 300 MB/s on ABB2 in beam test measurements.

A hit packet consists of a precise time-stamp and the hit coordinate information.

2.1.3 Hardware-aided indexing

Other than a pure buffering function, the ABB carries out also the FLES assistance tasks in event building. For current version, epoch markers are indexed and stored in a ring-like buffer in the host memory space, which helps the applications by the event reorganization.

The epoch marker (EM) in CBM DAQ system plays an important role for event building and hit reorganization. With current configuration of ABB, the upper-level software has to search these epoch markers through the whole data in the memory space, because the ABB only moves the incoming hit data packets into the host memory space according to the DMA descriptors and does not have the knowledge inside the data stream.

The need for a new epoch marker handling mechanism arises from the software applications. Epoch marker indexing or selection in the firmware will simplify the software procedures and therefore improve the system capability by relieving the host CPU load.

For future ABB development, such specific message handling will take place for similar boundary identifying of different hit packets. So the implementation of EM handling is a good practice for the future DAQ operation.

As figure 2-6 shows, a ring-buffer is used for collecting the EM indices and it is a part of the host memory space. The Active Buffer logic distils the EMs out of the main data stream and sends them to the ring-buffer in the host in proper timing.

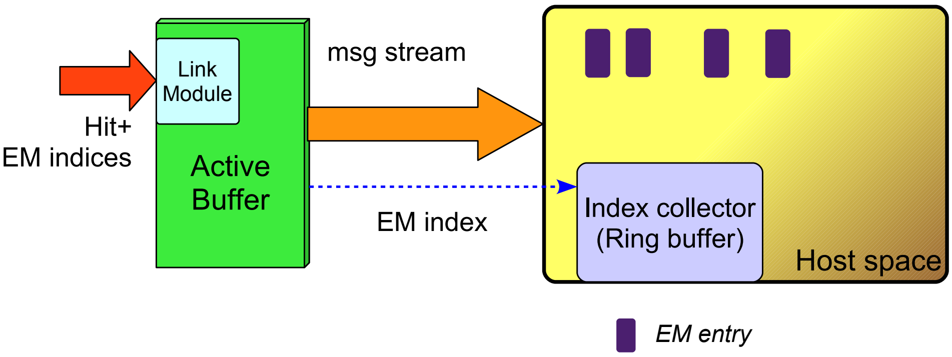


Figure 2-6 Epoch marker handling concept

EM handling is discussed in *Appendix **A.6***.

Similar requirement comes out of the SYNC messages, which helps to define the time slices. The SYNC indexing will benefit from the EM indexing design. In the next version of the Active Buffer design, multiple marker indexing shall be supported.

2.2 Buffer kernel

FIFO (**F**irst-**I**n-**F**irst-**O**ut) is generally used for data buffering. If the data production (writing) and data processing (reading) are not of the same rate, the buffer should be built towards a better overall performance. If the two clocks, write and read, are not of the same rate, the buffer is usually used to separate the clock domains.

Ideally, the buffer works the best if the FIFO has infinite depth. However, on practical circuits, the resource is never unlimited. Therefore, we can only try to make a buffer as deep as we can to approximate that ideal condition.

For a simplified FIFO control, the write rate and read rate should be approximately equal, or the read speed is a little bit faster. In this way, the FIFO is not expected to have a deliberated flow control. Effective buffer utilization reserves not too much space vacant.

Larger FIFO has advantage over smaller one in a buffer system without flow control. The larger (deeper) the FIFO is, the lower overflow risk we have in application. However, FIFO product size is limited by the IC technology, which stomps below hundred million bits by 2011. And such large-size FIFO usually costs several dozens of dollars. Compared with SDRAM, pure FIFO chip density has less competence. In programmable devices, such as FPGA, the possible FIFO is yet smaller because of the resource limitation and expensiveness.

Most FIFO applications in DAQ research field are combined with programmable devices. So we have the solution like

DDR(II/III) SDRAMs + FPGA → Large-size FIFO

Such a solution increases the FIFO depth easily over hundred million entries.

Of course the FPGA is not only used for building FIFO. Most resource is available to other functions.

Due to the irregularity of the arrival and departure, the buffer occupancy is fluctuating. This fluctuation causes the buffer full for some specific time. And when the buffer is full, there are usually two ways to process,

(1) set flow control upstream; or

(2) abandon the overwriting data.

The first solution demands extra logic to control the data flow speed so as to make the system structure complex and the behaviour difficult to predict; and the second solution leads to data loss. Different applications use different buffer management and there is no universal solution.

Flow control can not be essentially eliminated with our solution, but it can be immensely reduced. For a buffering system with quasi regular flow rate, the flow-control logic is able to be spared.

For a buffer system with flow control, the buffer depth has influence upon how frequently the flow control actions. Certainly fewer flow control actions offer higher performance in terms of throughput. A larger FIFO can lower down the flow control probability than a smaller one.

With variable aspect-ratio FIFOs, half-bus access is supported. However, in our project, only full-bus access is supported, because otherwise the memory kernel (SDRAM) controller would be much more complicated.

Four points to note again in building such a FIFO wrapper are,

A) data into and out of the SDRAM are aligned to row boundary, to have a better and simplified management over the SDRAM module. Only when the data amount in the small input FIFO block reaches a row size of the DDR SDRAM, the data transfer into the DDR-Buffer is carried out. And on the opposite side, only after the output FIFO is able to accommodate a row-size data, the data are directed from the SDRAM to the output FIFO again. The benefit lies in minimized row-open and row-close operation for the SDRAM. Hence, the hazard of logic error is diminished.

B) internal data run in double-width data bus, i.e. 64-bit bus, which suffices the possibly full data rate, even if the SDRAM overhead such as auto refresh and row recharge has to be taken into account. This helps to keep the external full-empty state the same as a conventional FIFO, which means, full state is only expressed when all the memory units (small FIFOs and SDRAM module) are occupied; empty state is only expressed when all the memory units are vacant. This is also valid for almost-full and almost-empty flags.

C) PAD delays are variable and calibratable, so that the design has general meaning to the variety of boards. On the Xilinx FPGA board, IDELAY is used with variable delay value. Auto-calibration logic acts in the initialization and finds the optimized delay parameters for the IO pins.

D) SDRAM module clock frequency is some percent higher than the external FIFO clock frequency, in order to compensate the SDRAM operation overhead such as refresh. Taking burst length = 4, page size = 128 as example,

- Page activate needs 2 cycles of DDR clock
- Page precharge needs 6 cycles (without auto-precharge)
- 14 cycles of auto-refresh are inserted every 1024 cycles

Hence, the minimum clock upgrade ratio is approximately

$$1 + (2+6)/128 + 14/1024 = 1.08$$

In our test, we use 150 MHz clock for the SDRAM module compared to 125 MHz of the external clock, where the ratio is

$$150/125 = 1.20 > 1.08$$

Such increase upon clock rate is done by the frequency synthesis module in FPGA.

2.2.1 Requirement

The FIFO specification should be well complied, which means, on the write and read ports the timing diagram is the same as the standard FIFO,

– Write enable comes together with the input data.

– The output data goes out at exactly the next cycle of read enable (provided the FIFO not empty).

– The `Full` flag prevents further writes into the FIFO and the `Almost-Full` flag tells a possible overwrite hazard.

– The `Empty` flag is asserted when there is no more data in the entire buffer system and the `Almost-Empty` flag tells a possible under-read hazard.

– The FIFO should support asynchronous write and read, which means, the write clock and the read clock rates can be different.

However, the DDR (II) SDRAM module has certain overhead during operation, e.g. page-open and -close, refresh, which makes the real-time receiving and sending of data impossible. A sub-buffer layer is needed between the user logic and the SDRAM modules. Therefore, setting smaller standard (asynchronous) FIFO modules at the input and output ports is a reasonable solution, so that all the requirements above are naturally fulfilled. The smaller FIFO here is called proxy FIFO.

Random data flow arbitration between the smaller port FIFOs and the SDRAM module is complex and prone to faults because of the page initialization actions entangled with write or read control of small FIFOs. Therefore, we have the solution of page-aligned internal data transfer, in which every transfer involves one and only one page initialization of the SDRAM module. Such regular operation helps to simplify the arbitration logic and to reduce the debug effort. And as a result, those smaller standard FIFOs at the interface ports should be deeper than one page size of the SDRAM module. In our system, 512-word FIFOs are chosen on the safe side, corresponding to a 128-word page size of the SDRAM module.

2.2.2 Logic building

Following discussion is with a data bus width of 32 bits, double clock edges. And in practice the data bus can of course be wider.

The internal asynchronous FIFO blocks in FPGA are together used for this design. Of course the internal FIFOs are quite small in size and we use altogether 4 of them to make the conversion, 2 for input port and 2 for output port. They are used in pairs because of the double clock edge transfer.

The diagram is depicted in figure 2-7. One pair of these 4 small FIFO blocks is connected to the input port, and the other pair to the output port. Between them is a DDR buffer control module, which takes over the processing upon the external DDR SDRAM module. This DDR buffer receives the input written data and decides whether to transfer them into the SDRAM or to the small output FIFO. If the saved data in SDRAM are to be transferred to the output port of the large FIFO, they are arbitrated by a MUX before the small FIFO block of the output port.

To simplify the refresh operation of the DDR SDRAM, the internal transfer takes 512 entries as the block size unit. In this way, every read or write access to the DDR SDRAM is well aligned to the page boundary, to achieve simplified and robust control logic.

As can be seen in figure 2-7, the internal data bus is 64 bits wide. This helps reducing the possibility of flow congestion and to achieve sufficient time for processing the input data, which can be full rate. The data flows alternate among the channels in the data path of the large FIFO. The DDR buffer module consists of two major parts, a FIFO-to-DDR interface and a DDR-to-FIFO interface. With these two interfaces, the DDR SDRAM is "transformed" into FIFO.

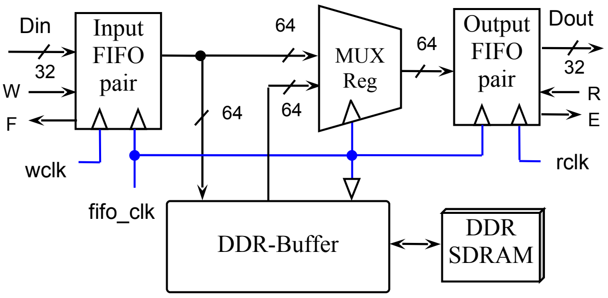


Figure 2-7 DDR-FIFO block diagram

Figure 2-8 shows the input FIFO details. It is built upon a pair of primitive FIFOs from the FPGA vendor.

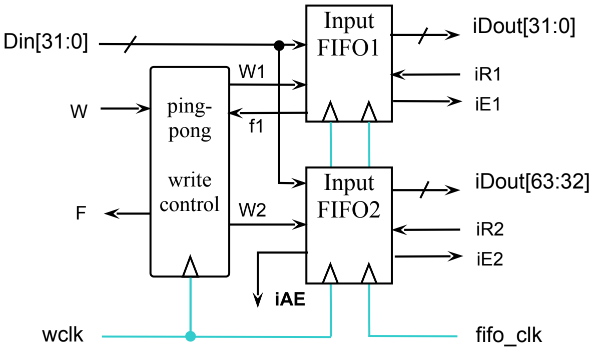


Figure 2-8 Input FIFO

The "ping-pong" write-control in figure 2-8 illustrates the alternate writing into the 2 input-FIFOs, i.e. the rising-edge 32-bit data is put into input-FIFO2, and the falling-edge data into input-FIFO1. A waterline, iAE, is set to control the transfer from the input FIFO pair into the DDR-Buffer, instead of directly into the output FIFO pair. iAE flag is asserted when $N \leq 128$, where N denotes the entry number in the FIFO.

Similarly, output FIFOs are built at the output port of the hybrid FIFO, as shown in figure 2-9. This pair of small FIFO blocks is multiplexed for the final FIFO output. Another waterline signal, oAF, is used for flow arbitration. When this signal is asserted, N ≥ 384, the successive data will be put into DDR-Buffer instead of into the small output FIFO pair.

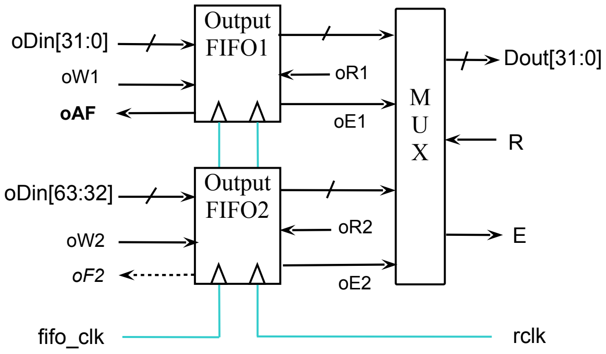


Figure 2-9 Output FIFO

In figure 2-9, the output MUX is not synchronized because of the empty flag timing requirement. These two waterline signals, iAE and oAF, help to keep the buffer performance while the SDRAM is well page-aligned.

Next, we discuss the data flow between these two channels. Figures 2-10 through 2-13 are the different situations for this large FIFO and explain how the data flow, where the bold lines denote the current data flow paths for each situation. In the brackets attached to each figure, the full-empty states of the input, output, as well as of the total FIFOs are listed.

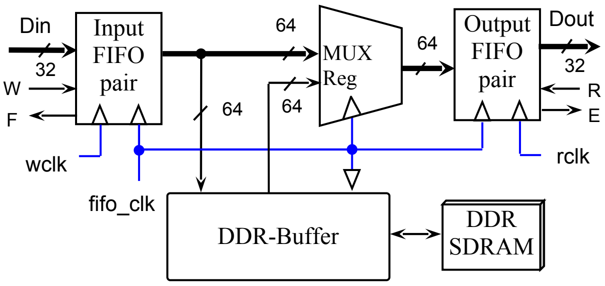


Figure 2-10 Data flow – loose

(DDR-Buffer: empty= **1**; Input-FIFO: empty= **0**; Output-FIFO: oAF= **0**)

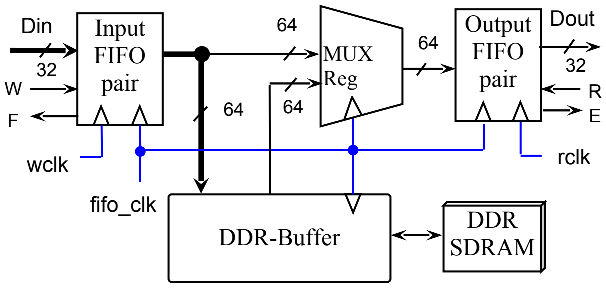


Figure 2-11 Data flow – congested output

(DDR-Buffer: empty= **0**/**1**; Input-FIFO: empty= **0**; Output-FIFO: oAF= **1**)

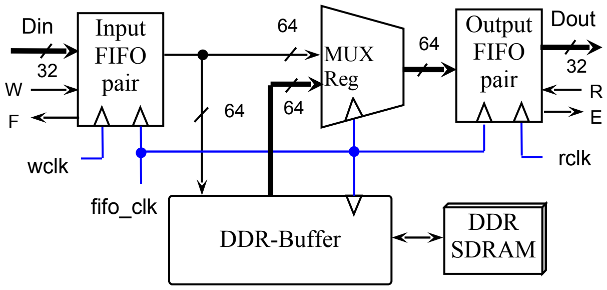


Figure 2-12 Data flow – weak input

(DDR-Buffer: empty= **0**; Input-FIFO: iAE= **1**/**0**; Output-FIFO: oAF= **0**)

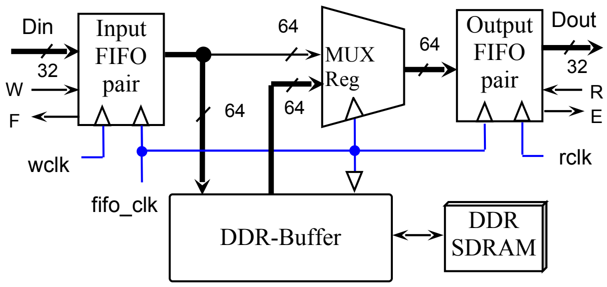


Figure 2-13 Data flow – dense

(DDR-Buffer: empty= **0**; Input-FIFO: iAE= **0**; Output-FIFO: oAF= **0**)

The flags coherence is especially handled because there are more than one data buffer stages and the mutual effect is made complex by the SDRAM overhead operations. All buffering stages must be well monitored to ensure the entire FIFO reports correct flags. The `Empty` state should be asserted only when there is no more data in the whole data module. And the same must be guaranteed for `Almost-Empty` as well as for `Full` and `Almost-Full` flags.

Data count logic is additionally designed, which provides information about the data amount to the upper-level application. For such a large-size FIFO, the data count has to be deliberated separately. As the standard FIFOs, the data count can be synchronized to the write clock or to the read clock. We have only implemented the data count that is synchronized to the read clock. The write-clock synchronized data count is not implemented, because the write port does not have to know the information of how many data inside. At the write port, the almost-full signal is sufficient to assert the flow control. The size of the FIFO we use is a little bit larger than 512MB, so the width of the data count with respect to the 64-bit bus should be greater than 26 bits. Therefore we use 27 bits for the FIFO data count.

2.2.3 Test and verification

2.2.3.1 Simulation

All behaviour of this transformed FIFO follows the approved FIFO specification, because the proxy (input and output) modules are real FIFOs. FWFT (**F**irst-**W**ord-**F**all-**T**hrough) mode is not supported in this version because of the cascaded FIFOs along the data path and the block-buffering property of the input FIFO. The EMPTY-falling logic has a bigger latency than the standard FIFO because the first data into the FIFO has to be transferred from one FIFO (input-FIFO) to another (output-FIFO). `EMPTY` is deasserted when data is available in the output-FIFO.

To make a strict verification over the wrapper logic, we build the simulation environment including the wrapper logic and the DDR SDRAM simulation module. Micron Co. provides the simulation module for their DDR chip in Verilog HDL. And our simulation is behavioural, to test the functionality of the logic.

(1) Independent write and read clock rates variable within a defined range. The frequencies of the write clock and the read clock can change randomly. This is the requirement for an asynchronous FIFO and also helps to create the different ratios of read and write.

(2) Random write and read rates variable. The write and read densities are varied over time, to get the Full and Empty function tested. Both `Full` and `Empty` signals are expected toggling. For the write and read rates, we use double-scale randomization, which means the involved signal varies both in small time-scale and in large time-scale. If the rate is randomized only with one time-scale, the toggling of empty or almost-full will be hard to observe, because they are averaged in a larger scale. Multiple-time-scale randomization prevents data flow smoothing in large time scale.

(3) Practically, the `Empty` state is easy to get because it happens at the most beginning initial state and sometimes during lower data rate. However, the `Full` signal goes up after a quite long simulation time, if we use the whole size of the SDRAM. The full-size simulation also takes longer time because

of the multiplied memory consumption. So we use the partial size of the SDRAM, by setting it to 1/8 of the whole size and the Full signal toggles much more frequently. This is equivalent to shrink the SDRAM address size, e.g. from 15 bits to 12 bits. The smaller size should be limited over a column size (8KB) and the bank boundary should also be straddled, so that the points where errors are likely to take place are covered. In this way, we can make sure the logic works for the true-size memory module and the prototype simulation can prove the real functionality. Such address down-scaling makes the simulation faster and the verification process is accelerated.

2.2.3.2 Self-test

Figure 2-14 shows a self-test scheme for the design. A pseudo-random (**L**inear **F**eed-back **S**hift **R**egister, LFSR) generator feeds data into the FIFO logic with pseudo random write enable, and the output data are checked by the checker. The read enable is also calculated from a pseudo-random generator. The generator and checker use the same sequence of the data stream. When the checking fails, an error signal is asserted. The error signal is verified in an error-injection test that generates deliberated failure for all sorts of errors and it is asserted as expected.

In bench test, two buttons, AW (**A**ccelerate **W**rite) and AR (**A**ccelerate **R**ead), are used to accelerate the write or read. Write enable and read enable are both combinations of the pseudo-random sequence and the manual button state. Without pressing these buttons, the read and write both run at a lower speed, e.g. 1/16 speed. And the acceleration buttons can drive write or read to a configured speed, in order to force the buffer full or empty or critical. The configured speed is set by some switches (sw for write, sr for read) on the test board. For instance, discrete speed grades, 1/8, 1/4, 1/2, full speed and combination speeds such as 3/8, 5/8, 3/4 or 7/8, are available. These buttons make the test closer to a random environment. Of course, the generator logic takes care of that the FIFO is not overwritten. When it is full, the write must stop and the random registers stops counting up.

Without these buttons, the data flow inside the FPGA is more regular and some subtle bugs might be hidden even if a (pseudo-)random sequence of write and read is used, because the logic can never be able to generate true random events.

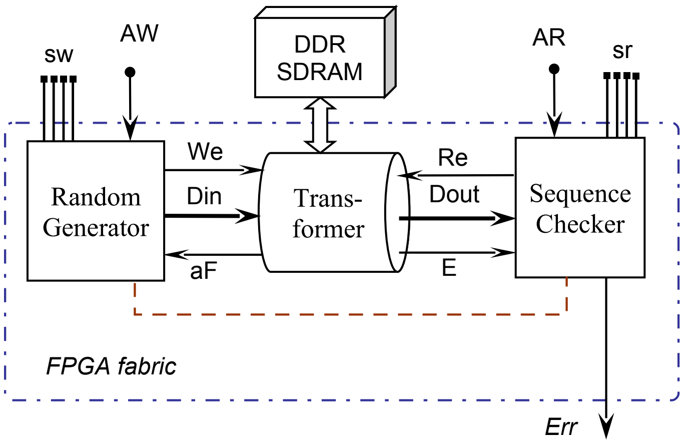


Figure 2-14 DDR FIFO Self-test diagram

After a thorough test over 4 hours, with automatic and manual modes mixed, the error signal is never asserted. That means the large FIFO has succeeded in emulating all functional behaviours of a standard FIFO. The resource consumption of this self-test can be found in *Appendix **A.3***.

Timing diagrams of DDR FIFO from the ChipScope viewer prove that this FIFO behaves correctly as the standard FIFO, both in small occupancy and in large occupancy. These diagrams are shown with EMPTY, FULL, ALMOST_FULL, as well as the generated data count signals. The experiments are made with the help of AW and AR buttons. The write and read control signals are generated with pseudo-random logic. The data integrity is proven by the deasserted chk_err signal. From the reset timing diagram in figure 2-15 we can see this SDRAM FIFO has longer latency from write to empty low. The proxy BRAM FIFO is 1024 words altogether and hence, the SDRAM kernel is used if the data count is greater than 0x100, as in figure 2-16, figure 2-17 and figure 2-18.

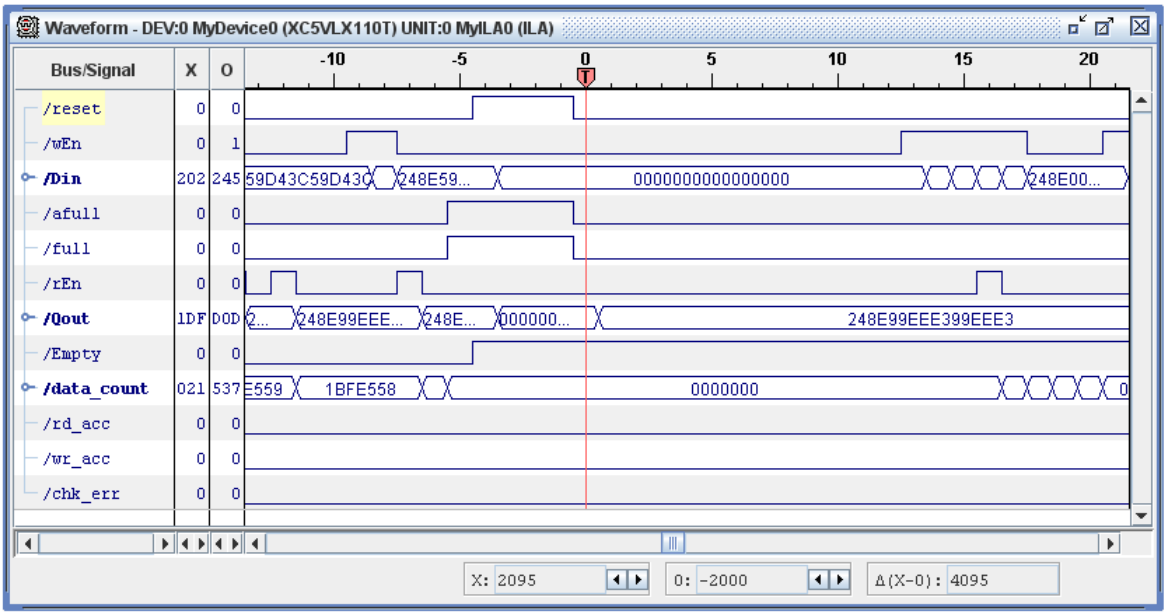


Figure 2-15 Reset

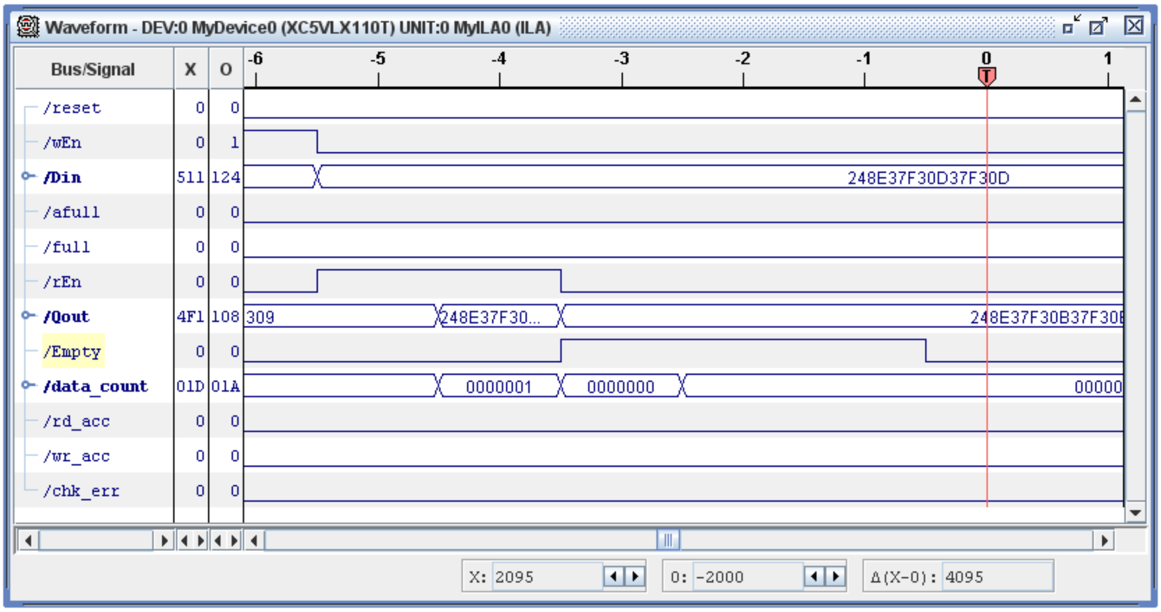


Figure 2-16 Empty logic

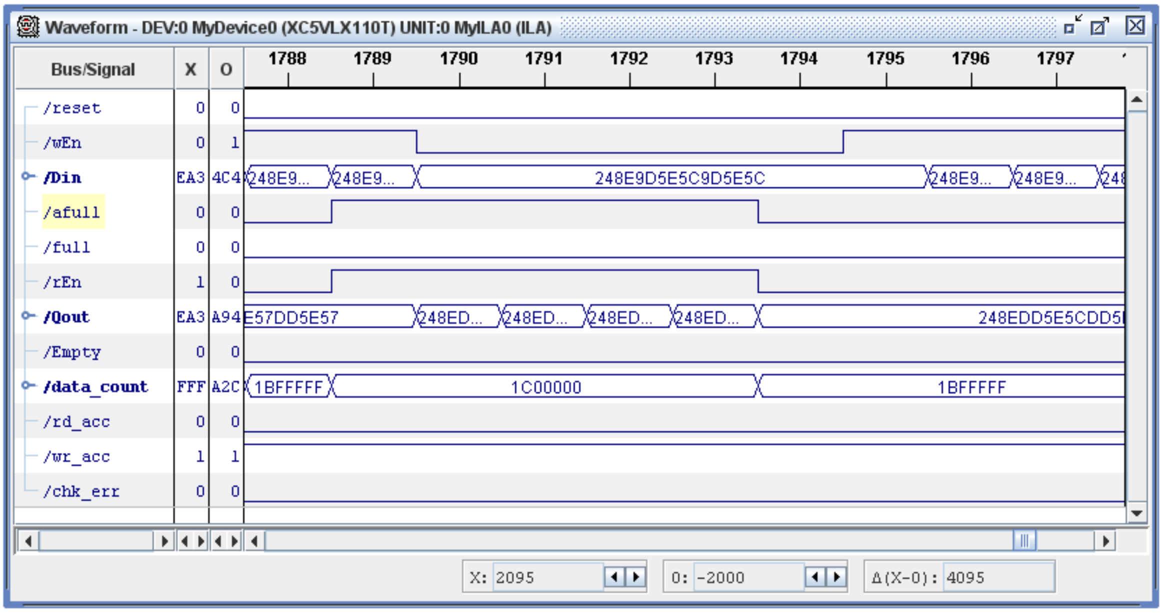
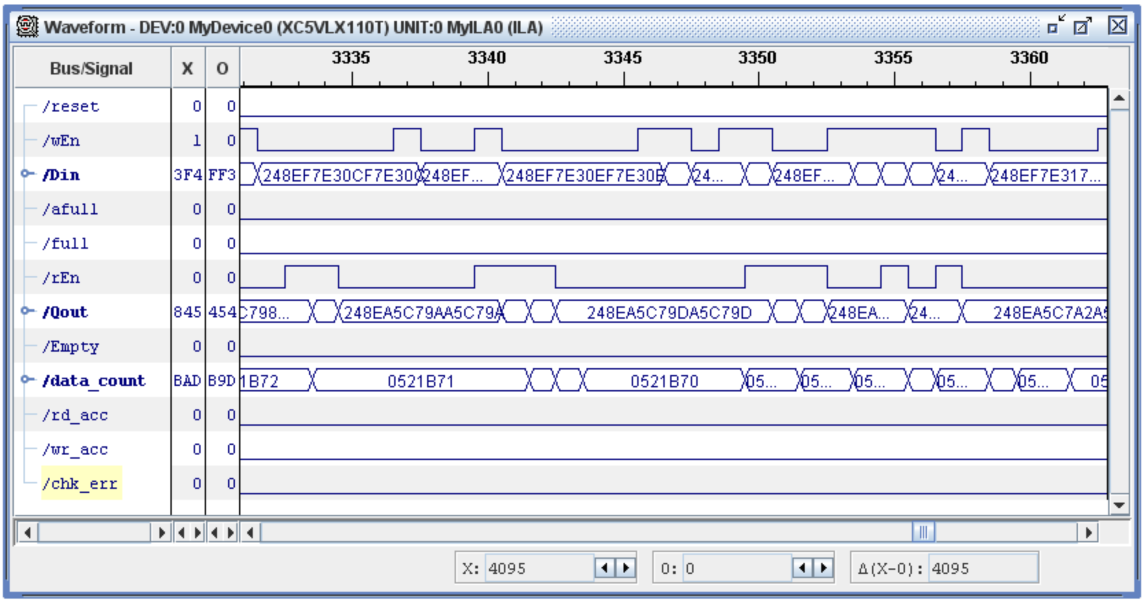


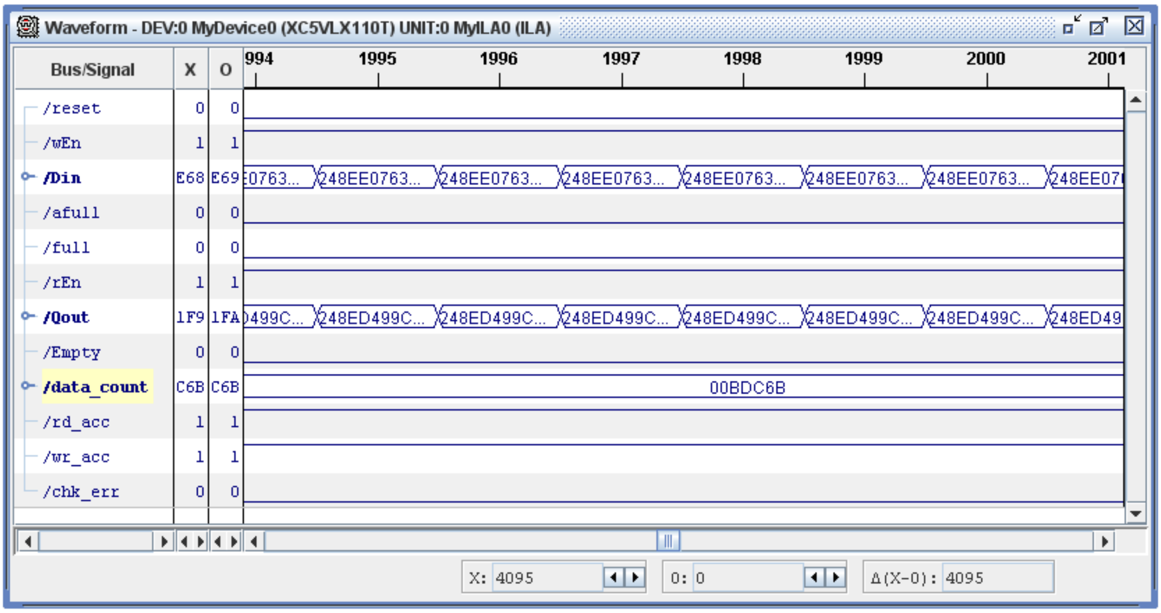
Figure 2-17 Almost-full logic



Bus/Signal	X	O
/reset	0	0
/wEn	1	0
/Din	3F4	FF3
/afull	0	0
/full	0	0
/rEn	0	0
/Qout	845	454
/Empty	0	0
/data_count	BAD	B9D
/rd_acc	0	0
/wr_acc	0	0
/chk_err	0	0

X: 4095   O: 0   Δ(X-O): 4095

Figure 2-18 Low rate flow



Bus/Signal	X	O
/reset	0	0
/wEn	1	1
/Din	E68	E69
/afull	0	0
/full	0	0
/rEn	1	1
/Qout	1F9	1FA
/Empty	0	0
/data_count	C6B	C6B
/rd_acc	1	1
/wr_acc	1	1
/chk_err	0	0

X: 4095   O: 0   Δ(X-O): 4095

Figure 2-19 High rate flow

2.2.4 Implementation in FPGA

Timing constraint can be met to over 175 MHz for Virtex4 FPGA with -10 speed grade for this design. The numbers of flip-flops and (4-input) look-up-tables are both under 1000 and the number of 18Kb RAM blocks is 4. For the adjusting of data bus of the SDRAM module, 20 IDELAYCTRLs are used.

We use two parallel 32MB, 16-bit DDR SDRAMs with CAS latency of 3 as the kernel memory. It is driven by a DDR clock of 175MHz. Our global clock for the FIFO wrapper runs at 150 MHz.

With a long and sustained test, mixed with manual accelerations, the FIFO behaves well in terms of performance and in stability. So, the peak bandwidth goes up to

$$150\text{ MHz} \times 2 \times 32\text{ bit} = 9.6\text{ Gbps}$$

in the test system. However, in the DMA logic, the transaction layer clock rate is 125 MHz, where the performance is

$$125\text{ MHz} \times 2 \times 32\text{ bit} = 8.0\text{ Gbps}$$

This performance estimation suffices the 4 Gbps requirement.

2.2.5 Stand-alone IC scenario

The design in FPGA can be transported to build a stand-alone ASIC chip that transforms DDR SDRAM into a large-size FIFO, as figure 2-20 shows. The transformer logic consists of the (small-size) input and output FIFOs, MUX and the DDR buffer. The clock signals are omitted in figure 2-20.

Here we have

SDRAM modules + Wrapper ASIC => large-size FIFO.

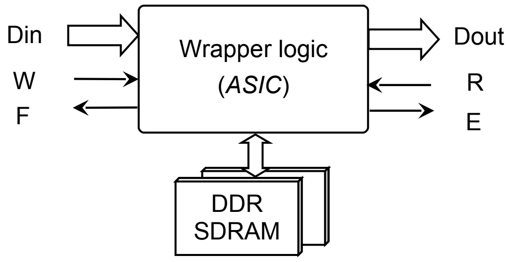


Figure 2-20 Stand-alone IC

And the data buses can be varied in width, dependent upon the applied number of the DDR SDRAM chips. Of course the operation frequency is mainly limited by the timing of the DDR SDRAM chips. As we have seen, the resource is not a big problem, but the pin out.

The extra built-in FIFOs cost is proportional to the buffer bus width, if their depths are fixed to e.g. 512 words. So this part is the major resource consumer, but it is not proportional to the buffer depth.

The data counts can be implemented as in our project and they are proportional to the address width of the SDRAM modules, about the level of *O*(log*N*), where *N* is the volume of the entire buffer.

By the FPGA resource estimation from the stand-alone test implementation report (*Appendix **A.3***), such an ASIC (64-bit data bus) can be achieved within 10 million gates.

Such structure might be useful to build up more powerful computing sys tems, which are in great need of buffer resource. For instance, the DAQ systems in high-energy physics often have to work a lot for triggering strategies in saving buffer resource, where SDRAM FIFO is supposed to be a nice solution to simplify the design complexity. It can also be combined into chipset (northbridge) in the commercial PC main-board to provide a specific channel of data flow. With more and more network video-audio applications today, such buffer architecture shall provide preferably higher performance and neater structure for network instruments.

2.2.6 Notes to the transformed FIFO

Advantages of such FIFO include good scalability and economic. The cost is acceptable and the clock rate is satisfactory. More importantly, the depth is much larger, usually dozens of times larger, even hundreds of times larger than the marketable devices.

This approach can be easily expanded to DDR2 or DDR3 SDRAM and other sorts of memory chips, such as SRAM, or ZBT, even Flash memories.

On the opposite direction of integrating, we provide large-size FIFO solution in separate fashion. Stand-alone transforming ASIC chip is feasible and the functionality has been verified by FPGA implementation.

For data integrity reason, the FIFO bus width today is popular with multiple of 9, instead of 8. However in our example, constrained by the DDR SDRAM chip we take, the bus can only be 16, 32 or 64, or so on.

Later in this chapter we can see that the event buffer prefers additional bit parallel to the data bus. However, this FIFO is difficult to expand its bus width, because we use the SDRAM module without parity check. If framing or other marking signals are to be accompanied, work-around has to be found.

To have 9-multiplied bus width, 1-bit wide internal RAM module can be used parallel with the main SDRAM module and these 2 modules should have the same depth (address range) so that their address pins can be merged and treated as one address bus. However, this solution usually takes too much resource in the wrapper IC. For example, a 1GB FIFO with bus width 64 bits should be additionally attached with

$$10^{30} / 8 \text{ bits} = 128 \text{ Mbits}$$

memory. It is a big cost.

Therefore, a more plausible solution is to use available 9-folded SDRAM module as the FIFO kernel. COTS can be purchased like 2GB SODIMM MT18HTF25672PKZ-667 of Micron Co, which is x72 module. [57]

In the FIFO tests, we use the separate write and read indices. A counter is set for the data inside the DDR SDRAM module. And those indices are calculated with the used counts of those small input and output port FIFOs. In this case, the common questions of synchronizing the read and write addresses/counts are similar to the normal FIFO designs. [58] [59]

2.3 Future Active Buffer upgrade

To have much more abilities for marking the data packets, additional bits are highly demanded for the next version of Active Buffer design. The disadvantage for token insertion is obvious, because it makes the logic complexer of both insertion and filtering and it introduces extra latency into the data stream. For the moment, our data bus is 64 bits. One extra bit is enough to bring noticeable simplification to the event markers. Extra bits are transparent to the software, so the software does not have to change.

A customizable indexing mechanism is considered, where the hardware (FPGA) prepares the virtual index channels and the software allocates index memory space in the host. In this way, the software decides how to pick up the indices and the hardware accelerates the picking-up. This is a prototype of event building.

2.4 Summary

In this chapter we build up a large-size FIFO-standard buffer with the DDR2 SDRAM module. It exceeds the maximum size of available products in the market meanwhile has the competent performance on the ports. The Active Buffer project uses it as the central storage component. Extra logic resource to build such buffer in FPGA is acceptable. Such design can be expanded to the ASIC implementation in order to have large-size FIFO wrapper chips with modest prices.

Chapter 3 ABB – Interfacing networks to hosts

3.1 Introduction

In the CBM DAQ system, the problems to be solved by the Active Buffer include,

- **receiving**: receive data stream from the network,
- **event building**: process the messages and reformat them, and
- **storing**: store the message data into the host memory space.

As figure 3-1 shows, the FPGA on the Active Buffer board (ABB) has the network interface (NI) contributed by CAG (**C**omputer **A**rchitecture **G**roup) of Heidelberg University and the host interface (HI) IP core provided by Xilinx. The NI is the solution for data receiving. Event building solution is initially attempted with the epoch marking support in *Appendix **A.6***. The throughput around the north bridge and south bridge in the host computer is much higher than that around the Active Buffer. However, the CPU cannot afford such high bandwidth to the Active Buffer all the time and this is why we need a large buffer developed in chapter 2, as well as the memory interface (MI) to the large buffer. In this chapter, we focus on the data storing solution.

NI runs at 5 Gbps and HI runs at 8 Gbps. Due to the incompatibility of these two interfaces, they cannot be directly connected to achieve the steady data transfer rate of 5 Gbps. In figure 3-1, to meet the optical link speed of 5 Gbps, the bottleneck should not be inside the ABB – the PCI Express card. If the PCI Express channel between this card and the chipset runs at low speed, the congestion will sooner or later happen on the ABB, although it has large-size local buffer. From the processor point of view, there are two options to transfer data from the peripheral to the host processor, PIO (**p**rogrammed **I/O**) and DMA (**d**irect **m**emory **a**ccess). The PIO mode is obviously too slow concerning the transfer speed, which provides read performance under 0.05 Gbps. Hence, the reasonable solution is to implement a DMA engine with the resource on the peripheral in order to make the data transfer as fast as possible. The DMA function is not included in the Xilinx PCI Express IP core and it is an important part of the Active Buffer development.

For the user, DMA is a rapid way to transfer large chunk of data between two devices without CPU intervention, most usually one of them being the host memory. Such bypass of the processor needs extensive hardware, namely a DMA controller. The DMA controller receives the user instructions to start the DMA transfer. The DMA instructions include at least the source pointer and the destination pointer. The transfer size is often another essential parameter but in some cases, it does not have to be specified, then it is taken as infinite and the DMA never stops until the user wants to. The buffer management in the host memory space is carried out by the user to guarantee the data source provides correct data or the data destination has enough space to store the coming data.

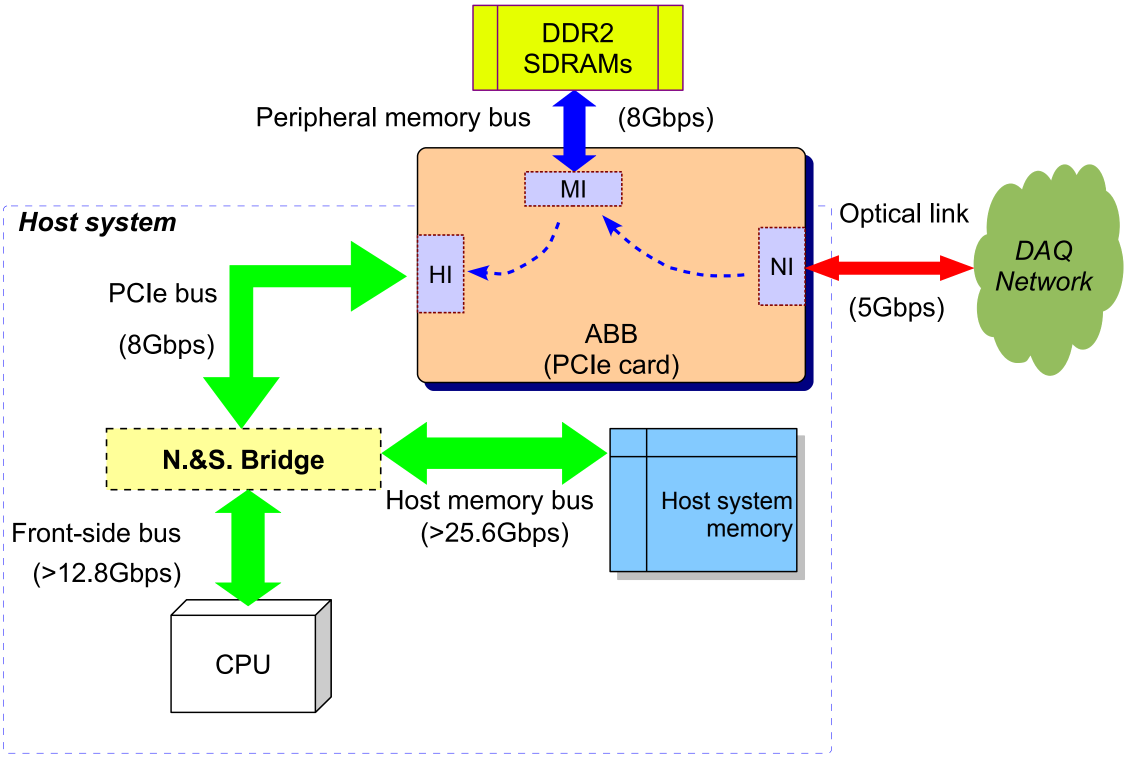


Figure 3-1 The role of the Active Buffer board (ABB)

A DMA controller must provide the possibility to peek the DMA status, either by polling the status register or by receiving interrupts, so that the user knows whether the DMA is going well and whether the DMA is successfully finished. And when the DMA is not working well or gets time-out, some DMA can be stopped or cancelled via the DMA controller.

A DMA is initiated by the user with a set of parameter instructions. Thereafter the DMA controller manages every data moving, incrementing the counter, sometimes even checking the flow control, until the current DMA is finished. Most often, another DMA transaction is to be executed as the subsequent transaction. Here for the subsequent DMA transactions, the DMA controller can choose to which extent it is guided by the processor. In terms of the processor involvement, DMA engines can be divided into 3 major modes, register-controlled (RC), scatter-gather (SG), and master DMA. Of course, the most CPU consuming scheme has the least hardware cost in building the DMA controller.

3.1.1 Register-controlled (RC) DMA

RC DMA consumes the most intervention of the CPU. The first DMA transaction is started by writing an instruction into a DMA control register in the peripheral device. After the first DMA transaction is done, the DMA controller halts with correct status information available to the user. The user application decides whether to carry out the next transaction. If yes, the same control register has to be written in order to start the next DMA. In this way, the user controls every single DMA transaction and the DMA controller is only responsible for correct and timely resolution of the DMA instruction and then transferring the defined amount of data.

The routine of an RC DMA is illustrated in figure 3-2.

(1) the user program sends the DMA start command with the transaction information to the control register (CR) in the DMA controller. Suppose that the necessary buffer allocation has been done in the host memory space.

(2) the DMA engine moves the corresponding block of data between the peripheral and the host memories. After finished, the DMA engine stops for current transaction.

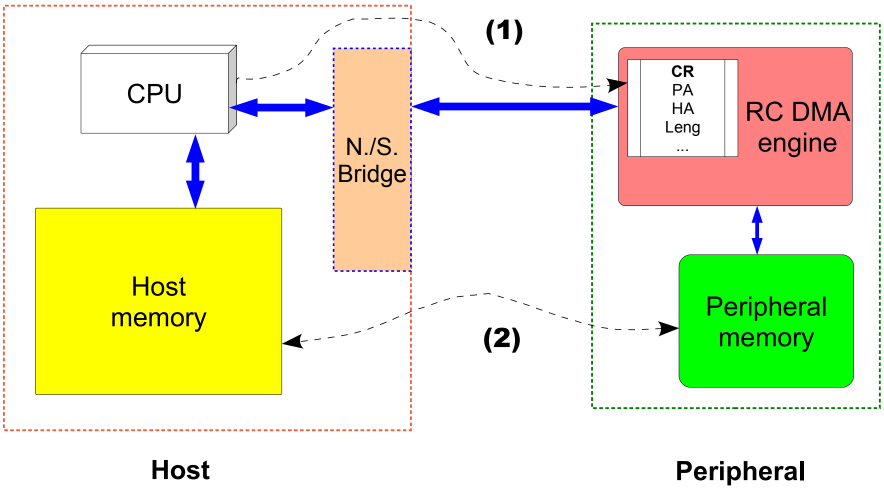


Figure 3-2 Register-controlled DMA procedure

In figure 3-2, the dashed arrows are the DMA transaction steps and the solid arrows are connections among the components. For a given DMA transaction, the bidirectional step (2) arrow has only one direction, where DMA read has only the left arrow and DMA write only the right one.

The RC DMA controller is easy to implement and takes minimal hardware resource, but the CPU has to pay more time to the DMA transaction procedure. In terms of the buffer management in the host memory space, the RC DMA is the simplest one.

3.1.2 Master DMA

Quite apart from the RC DMA, the master DMA controller needs only the initial start command from the user. Then, it manages all DMA transactions afterwards. In the initial command, there should be the pointer informations such as the source address and the destination address. The DMA size parameter is often omitted because the application on the master DMA usually cannot or does not have to define the DMA size in advance. However, the buffer size in the DMA destination memory space should be provided so that the producer knows when to rewind the writing pointer to the start of the buffer beginning. This target buffer is normally implemented as a ring-buffer, with the user application controlling one side and the DMA engine controlling the other side. Each side has its own counter about the data amount it has transferred. Usually the host application also knows the counter in the peripheral so that it is able to slow down the input or output port rate of the ring-buffer before it goes to overflow or underflow. The pointer synchronization problem is similar to that with an asynchronous FIFO. Interrupts are implemented in the master DMA controller when certain amount of data are ready or certain number of special token are collected.

The routine of a master DMA is illustrated in figure 3-3.

(1) the user program prepares the ring-buffer in the system memory space.

(2) the user program sends the DMA start command with the ring-buffer (RB) information to the DMA controller.

(3) the DMA engine moves the corresponding blocks of data between the peripheral and the host memories. Assume that the host always keeps the ring-buffer available for the DMA engine to write or read.

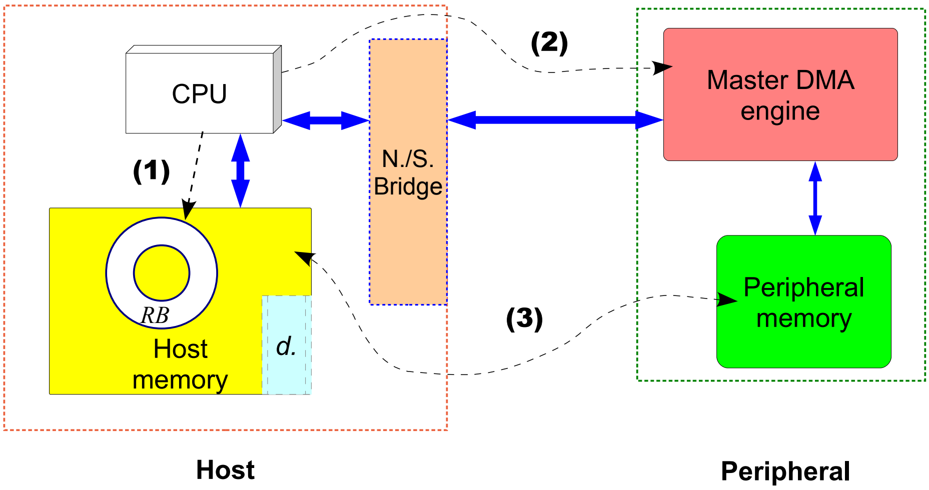


Figure 3-3 Master DMA procedure

For a given DMA transaction, the bidirectional step (3) arrow in figure 3-3 has only one direction. The master DMA mode is suitable for the burst data arrival with greatly fluctuating data rate.

Later we can find out by comparing the master DMA with the SG DMA that the former does not have to prepare the descriptor list but must implement complex buffer management algorithms for the payload data. Some master DMA controllers take over the management of the buffer descriptors in another area of the host memory space so that the host application can have the handle to the specific blocks of data. In the other case of master DMA controllers, the descriptor (*d.*) management can be spared if the data blocks are regular in size.

3.1.3 Scatter-gather (SG) DMA

SG DMA is something like a compromise between RC DMA and master DMA. This DMA mode does require so much involvement as the RC DMA, and is also not so complex in implementation as the master DMA. It is a suitable DMA mechanism for large size DMA in segmented USER memory space. The SG DMA is executed in a chain of descriptors. The DMA transfer is initiated via writing the first descriptor to corresponding DMA channel registers. The DMA engine supports multiple descriptors. The routine of an SG DMA is illustrated in figure 3-4.

(1) the user program prepares the descriptor list in the system memory space.

(2) the user program sends the initial descriptor and DMA start command to the DMA controller.

(3) the DMA engine moves the corresponding blocks of data between the peripheral and the host memories according to the current descriptor. After the current transaction is done, if the current descriptor is not `LAST`, do (4); if the current descriptor is `LAST`, end the current transaction and assert status bits.

(4) DMA controller fetches the next descriptor from the descriptor space in the host memory, and then does (3).

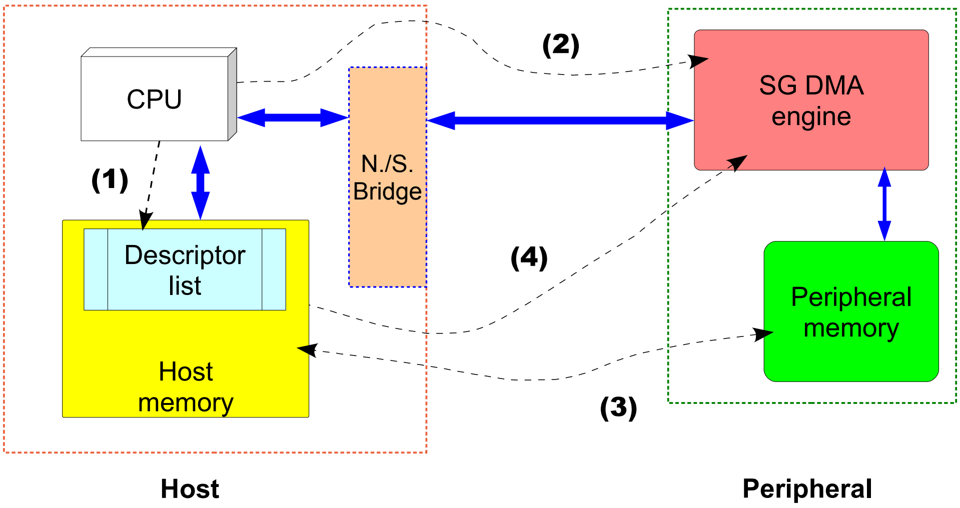


Figure 3-4 Scatter-gather DMA procedure

For a given DMA transaction, the bidirectional step (3) arrow in figure 3-4 has only one direction. In multiple-descriptor DMA, the next descriptor (a group of parameters) is requested by the FPGA according to the next buffer descriptor address in current DMA descriptor, after the current descriptor is executed. Then the DMA engine will execute that new descriptor. This process will continue until a descriptor with an asserted `LAST` bit in the Control parameter is done. If no descriptor with the `LAST` bit comes, DMA does not stop.

For a DMA chain consisting of multiple descriptors, subsequent descriptors are in the same order and format as the initial one, yet resident in host memory. To do a chained DMA, the upper-level application should get the next descriptors prepared in the host memory before it initiates the first DMA by writing the first descriptor in the peripheral. The DMA engine goes along the chain until the one with `LAST`=1 is executed, after which the DMA finishes.

Figure 3-5 illustrates a non-loop DMA consisting of multiple descriptors and figure 3-6 illustrates a looped DMA consisting of multiple descriptors. In both figures, `PA` is the peripheral address, `HA` is the host address, `BDA` is the next descriptor address (or buffer descriptor address), `Leng` is the current DMA size and `Ctrl` is the Control parameter. If the descriptor #0 in figure 3-5 is labelled with `LAST`=1, it will turn out to be a single-descriptor DMA transaction.

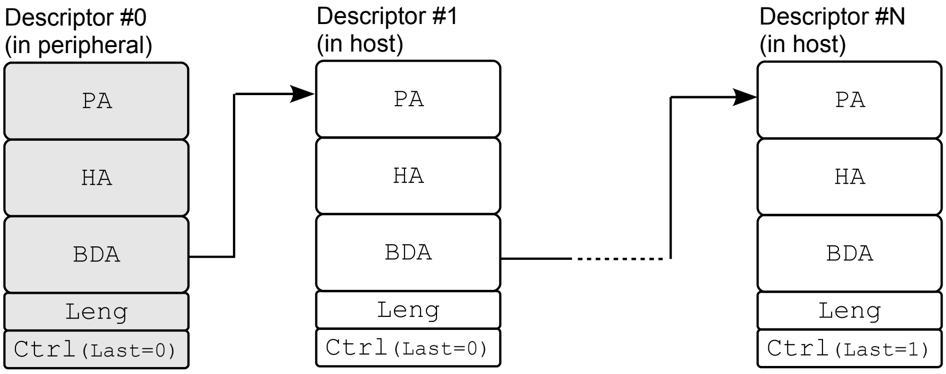


Figure 3-5 DMA Chain illustration – not looped

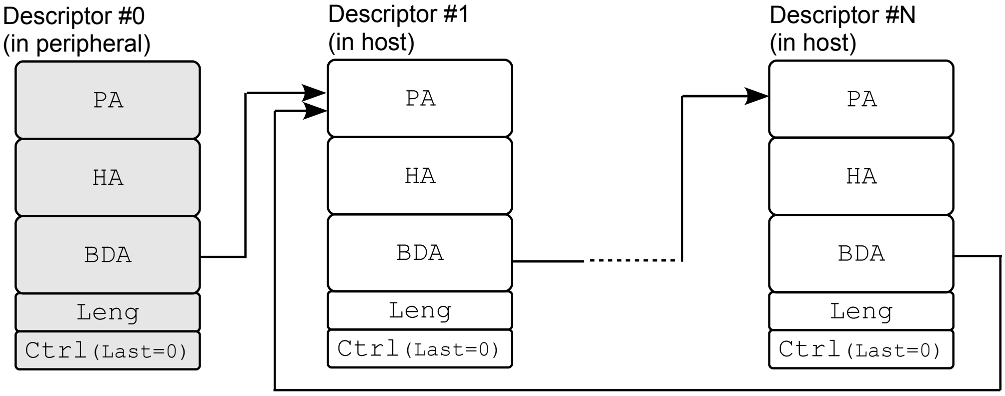


Figure 3-6 DMA Chain illustration – looped

3.1.4 Selected solution

The current version of the ABB DMA engine is of the SG mode, where the host defines the size, address, beginning and ending of the DMA transaction, as well as prepares the descriptor list for the DMA chain. The RC DMA is a simple mode but involves too much CPU cost, unsuitable for our DAQ requirement. The SG DMA reduces the host load between the DMA transactions and this is favourable for a continuous data receiving and transmitting. The SG DMA is sufficient for current DAQ applications and has presented stable performance in several beam tests.

If the DMA size is difficult to define prior to DMA transfer, the SG mode DMA requires still some commitment of the host. This can happen especially in circumstances of fluctuating data stream or random hit arrival. If a DMA is started at a time of lower arrival rate, it is very possible to go time-out. If the time-out is disabled, the host has to poll the DMA status frequently and quenches the time resource from neighbour threads. The host programs will be busy in polling the status of the peripheral storage, calculating or adjusting the next DMA size. Sometimes when the data buffer is in the edge state between empty and not-empty, the next DMA size is difficult to calculate in advance because a normal TLP (32 DW) is combined from several hit packets (about 8 to 16 DW each). Then, the DMA control program has to either break down the DMA to smaller descriptors, or use the time-out mechanism, or just wait for a longer time until the buffer occupancy fulfils a complete DMA size. Any method brings extra overhead for the host. Here, the master mode is preferable, because master DMA does not have to know the defined size in advance from the software side. A prerequisite is, the host still needs to allocate enough memory space for the events.

Still, as we can see in epoch marker handling (*Appendix **A.6***), due to the 64-bit bus interface of the event buffer, extra markers (epoch marker, SYNC marker, etc.) will introduce the token words insertion into the buffer. This solution has a side-effect of fake data count for the DMA decision. For example, if the DMA (read) engine detects that the current data count in the event buffer is 32 DWs (128 bytes) and assumes a DMA request is feasible to issue, but the actual word number is 31 DWs because of one inserted token. Then time-out might happen if no data comes in next span of time to compensate the fake DW(s). Time-out needs reset and data might get lost. For such situation, more flexibility should be assigned to the peripheral to decide what size of packet should be built concerning the current data amount available.

Epoch marker (EM) indexing module can be seen as an experiment, in which the peripheral card makes initiative data transfer. After the initial parameters (range and size) are sent to the peripheral, the host does not need to step in the data transfer control, if the ring-buffer is large enough and the host application processes the EM fetching fast enough. Such a scenario builds up the master DMA miniature. In master DMA, the host allocates a ring-buffer of a plausible size, and gets prepared for the coming interrupts. On interrupts, the host application does necessary processing, either moving the data in the ring-buffer to other storage destinations, or pausing the current peripheral procedure.

We have a working SG DMA engine for the beam tests and have collected a great deal of experience to integrate the DMA engine into the DAQ system. As the future development, master mode DMA is a clear goal.

3.2 PCI Express

To build the DMA channel for data transportation in the DAQ system, there are several options, PCI Express, HyperTransport, Gigabit Ethernet and so on. A good candidate is the PCI Express for its acceptable performance and full-duplex transaction style, which is good for quasi-real-time processing. It has favourable scalability by lane-wise configuration and it is point-to-point communication.

Another reason to use PCI Express is the already released PCI driver under Linux from the software group. In addition, the software for PCI Express is mostly compatible to previous PCI family. Drivers and libraries do not have to be rewritten if they have worked for legacy PCI. This provides the Active Buffer development with a sound foundation and a reduced much debugging time. The cooperation between the FPGA development and the PCI driver development is proved very efficient. PCI Express is more and more utilized today in data path applications.

PCI Express, officially abbreviated as PCIe or PCI-E, came into being with the initiation from IT top companies such as Intel, Dell, IBM and HP in April 2003 with the base specification revision 1.0a. It is a prominent upgrade of legacy PCI family (PCI, PCI-X). Comparing with legacy PCI components, PCI Express uses differential pairs instead parallel buses and brings great simplification upon the PCB routing. A successful change is the replacement of AGP with ×16 PCI Express in nowadays PC systems.

PCI Express provides attractive properties such as scalability, full-duplex and plug-and-play. The physical layer data path is built with differential serial pair and each pair provides 2.5 GT/s data rate for Gen1. It supports lane multiplication, such as ×1, ×4, ×8, ×12, ×16, ×32, etc. It has been widely accepted in the field of industry and research, which benefits from the compatibility to legacy PCI specifications on the driver peer.

PCI Express has layered architecture. Bottom-up are there three layers in the PCI Express protocol stack, as listed in table 3-1.

Table 3-1 Three layers of the PCI Express specification stack

Layer	Abbr.	Description	unit
transaction layer	TRN	Assembles and disassembles upper-level data packet and manages credit-based flow control.	TLP
data link layer	DL	Provides reliable packet exchange for transaction layer, including error identification and fault tolerance.	DLLP
physical layer	PHY	Manages the circuitry signals and low-level states. Divided into logical and electrical sub-blocks.	bit

Above physical layer, data on data link and transaction layers are in packet format, as figure 3-7 shows. TLP means transaction-layer packet and DLLP, data-link-layer packet. Payload and ECRC are TLP type independent.

ECRC is disabled in the application. The data error on the transaction layer is never observed and the overhead can be saved for better performance and simplified data path processing. The data from the network are CRC protected.

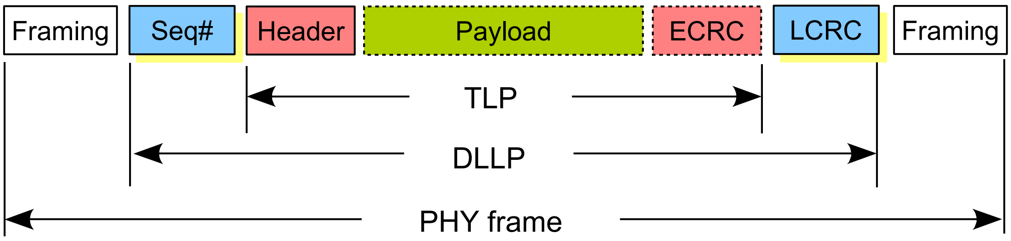


Figure 3-7 PCI Express packet encapsulation

To be legally and logically visible to the root via addressing, the peripheral memory space should be properly mapped. This is achieved by BAR (**B**ase **A**ddress **R**egion) configuration and mapping. In the address space, the peripheral can have maximum 6 normal regions of memory and if necessary, extended BARs can be used. The memory type can be of RAM for on-hole. The maximum size for one region is 4 GB. We have three BAR regions for ABB,

BAR[0] – 64 KB, system registers,

BAR[1] – 1 MB, mapped to a 32 KB Block RAM, and

BAR[2] – 4 KB, FIFO.

Since our project is on the transaction layer, table 3-2 gives the memory request TLP format and table 3-3 is the completion TLP, both in 64-bit interface. Table 3-4 shows the message TLP format, which is used for INT. 'R' denotes reserved and those bits must be zero. In PCI Express specification, all kinds of TLP are defined in details.

Table 3-2 Memory request TLP (MWr and MRd) format for 32-bit addressing in 64-bit bus interface (without digest)

<table>
<tr><td colspan="8">Byte +0</td><td colspan="8">Byte +1</td><td colspan="8">Byte +2</td><td colspan="8">Byte +3</td><td colspan="8">Byte +4</td><td colspan="8">Byte +5</td><td colspan="8">Byte +6</td><td colspan="8">Byte +7</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="2">F
X0</td><td colspan="5">type</td><td>R</td><td colspan="3">TC</td><td colspan="4">R</td><td>T
D</td><td>E
P</td><td colspan="2">Att</td><td colspan="2">R</td><td colspan="10">length</td><td colspan="16">requester ID</td><td colspan="8">tag</td><td colspan="4">last
DW BE</td><td colspan="4">1st DW
BE</td></tr>
<tr><td colspan="30">address [31:2]</td><td colspan="2">R</td><td colspan="32">data #0 or no data</td></tr>
<tr><td colspan="64">(*Further payload data or no more data*)</td></tr>
</table>

F: format

TC: traffic class (*nothing to do with the traffic classes in CBM DAQ*)

TD: TLP with digest

EP: error poisoned

Att: attribute

BE: byte enable

Table 3-3 Completion TLP format in 64-bit bus interface (without digest)

<table>
<tr><td colspan="8">Byte +0</td><td colspan="8">Byte +1</td><td colspan="8">Byte +2</td><td colspan="8">Byte +3</td><td colspan="8">Byte +4</td><td colspan="8">Byte +5</td><td colspan="8">Byte +6</td><td colspan="8">Byte +7</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="2">F
X0</td><td colspan="5">type</td><td>R</td><td colspan="3">TC</td><td colspan="4">R</td><td>T
D</td><td>E
P</td><td colspan="2">Att</td><td colspan="2">R</td><td colspan="10">length</td><td colspan="16">completer ID</td><td colspan="3">CS</td><td>R</td><td colspan="12">byte count</td></tr>
<tr><td colspan="16">requester ID</td><td colspan="8">tag</td><td>R</td><td colspan="7">lower address</td><td colspan="32">data 0</td></tr>
<tr><td colspan="64">(<i>Further payload data or no more data</i>)</td></tr>
</table>

Table 3-4 Message TLP format in 64-bit bus interface

<table>
<tr><td colspan="8">Byte +0</td><td colspan="8">Byte +1</td><td colspan="8">Byte +2</td><td colspan="8">Byte +3</td><td colspan="8">Byte +4</td><td colspan="8">Byte +5</td><td colspan="8">Byte +6</td><td colspan="8">Byte +7</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="2">F
X1</td><td colspan="5">type</td><td>R</td><td colspan="3">TC</td><td colspan="4">R</td><td>T
D</td><td>E
P</td><td colspan="2">Att
00</td><td colspan="2">R</td><td colspan="10">length</td><td colspan="16">requester ID</td><td colspan="8">tag</td><td colspan="8">message code</td></tr>
<tr><td colspan="64">(<i>Fields in bytes 8 through 15 depend on type of message</i>)</td></tr>
</table>

Interrupts are also carried by TLP, namely Msg or MsgD. PCI Express supports legacy INT mode and MSI (**M**essage **S**ignalled **I**nterrupt) mode for issuing interrupts.

Traffic class field (3 bits) in the TLP header provides possibility for virtual channel applications.

3.2.1 Root complex and endpoint

Only the root complex is physically connected to the processor(s). And this entitles the root complex with more responsibility, such as PM messaging, configuration, and so on. Figure 3-8 shows a typical PCI Express system architecture.

The message packets are transported in the PCI Express system. For a posted transaction, which needs acknowledgement or completion, the requester ID is used by the receiver to identify where the packet comes from. For a non-posted transaction, the requester ID is used by the receiver as the information where to send the completion.

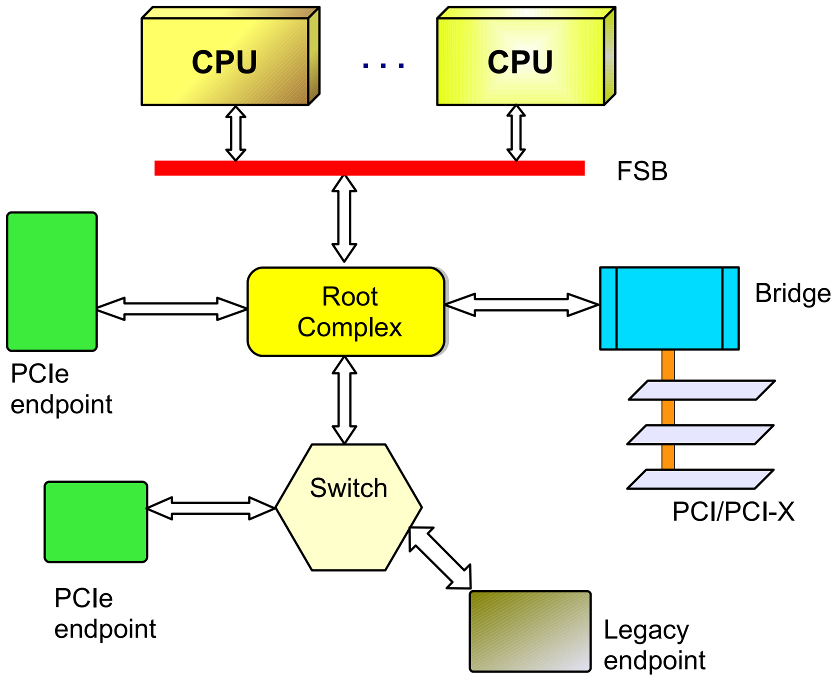


Figure 3-8 Root and peripheral in PCI Express

3.2.2 Transaction layer packet (TLP)

TLP category includes 15 different sorts of packets. They are listed in table 3-5.

In terms of acknowledgement necessity, non-posted TLPs need acknowledgement, such as MRd, IORd; posted TLPs need no acknowledge, such as MWr, IOWr.

Zero-length read is allowed. It is deliberated for flush requests. The processing for zero-length read is not much work for RAM-type memory, but it should be careful for one-hole memory, such as FIFO. If the FIFO is read to fulfil the dummy read request, data loss will take place.

If the proposed data is not returned in an obviously long time, time-out is triggered to protect other healthy data packet traffic on the path. Without time-out protection, the system might crash and a cold reboot is compulsory, when a transaction holds the link for a deadly period. That is not happy for debug work.

Time-out for completion is between 10 ms and 50 ms, according to the PCI Express specification Gen1. We prevent transaction-layer time-out (MWr) in DMA read by checking the available data amount before fetching the buffer.

`MAX_READ_REQUEST_SIZE` and `MAX_PAYLOAD_SIZE` are two important parameters concerning the maximum amount of data for read and for write. They can be one of the following values,

`0x40, 0x80, 0x100, 0x200, 0x400, 0x800, 0x1000.`

Table 3-5 TLP category in PCI Express Gen1

TLP Type	Full name
MRd	Memory read request
MRdLk	Memory read request - locked
MWr	Memory write request
IORd	I/O read request
IOWr	I/O write request
CfgRd0	Configuration read type 0
CfgRd1	Configuration read type 1
CfgWr0	Configuration write type 0
CfgWr1	Configuration write type 1
Msg	Message request (without data payload)
MsgD	Message request with data payload
Cpl	Completion without data
CplD	Completion with data
CplLk	Completion without data for locked memory read
CplDLk	Completion with data for locked memory read

DMA engine takes all of these possible values into account, either in the integrated version or the DPR (**D**ynamic **P**artial **R**econfiguration) version. For completions, there is a parameter to give freedom to some extent, namely RCB (**R**ead **C**ompletion **B**oundary). There are only two possible values for RCB in PCI Express Gen1, 64 bytes and 128 bytes. This parameter states that the completion to one MRd may be cut to smaller RCB-aligned packets, but this is not a must. This property of PCI Express must be treated carefully so that the completions are not missing. For example, in our system RCB = `0x40`. For an MRd with size = `0x60` bytes and the address = `0x12340030`, there are 4 valid completion possibilities,

```
a. 0x60
b. 0x10+0x50
c. 0x50+0x10
d. 0x10+0x40+0x10
```

Although in our tests, only the possibility d is found, the logic should be able to take correct data out all possible combinations of the completions.

Throttling includes source throttling and destination throttling. Source throttling is equivalent to the sending buffer empty status and the destination throttling to the receive buffer full status. Throttling is allowed for the TLP transfer and only when both the transmitter and the receiver are not throttling, the data on the bus is valid.

Flow control on transaction layer is based on credits.

It should also be mentioned during our experience with PCI Express Gen1 on different platforms that PCI Express is suffering larger overhead for upstream data transfer with newer chip-sets. Where the performance penalty comes is for us still an open question.

3.2.3 PCI Express evolution

PCI Express Gen1 uses 8b/10b encoding scheme, which has 20% overhead on physical layer. Therefore the 2.5GT/s bit rate is equivalent to 250MB/s data rate per lane. [60]

PCI Express Gen2 specification is released in January 2007, drives 5GT/s per lane, which is double of the Gen1. With 8b/10b encoding, its actual per-lane data rate is 500MB/s. [61]

PCI Express Gen3 is released in November 2010, drives 8GT/s per lane. Instead of 8b/10b in Gen1 and Gen2, Gen3 uses scrambling polynomial upon the raw data stream which is equivalent to 128b/130b encoding scheme, reducing the overhead to 1.5%. In this way, the 8GT/s lane bit rate corresponds to 985MB/s, delivering double data rate compared to Gen2. [62]

3.3 DMA transaction-layer behaviour

In our design the Xilinx PCI Express IP core for FPGA covers the lower two layers (PHY and DL), so we do DMA design on the transaction (TRN) layer. As in PCI Express specification, altogether 15 types of TLP's are defined. However, as far as our project is concerned, only 5 of them are resolved and used, MRd (read request), MWr (write request with payload), Msg (message as interrupt), CplD (successful completion for read request, with data payload), and Cpl (unsuccessful completion for read request, without data payload).

A most simple data transfer model over PCI Express can be demonstrated with a PIO logic on the transaction layer, which supports basic read and write TLPs as well as completions for read request. This model is illustrated in figure 3-9. The peripheral memory is a generalized BAR space and it can denote multiple regions. For a PIO read request, the peripheral should response with completion (with data or without data). For a PIO write, the payload data is stored into the memory space.

PIO transactions create a direct and simplified access to the peripheral. In terms of performance, PIO is not satisfactory because every TLP (MRd, MWr, or CplD) carries only one double-word and the latency between consecutive TLPs is quite large (dozens of clock cycles). Therefore, for applications concentrated on data transfer, a DMA engine is expected on the transaction layer of PCI Express besides PIO support, and should be available for scatter-gather mode. Scatter-gather DMA is a natural result of segmented USER space memory in the host node.

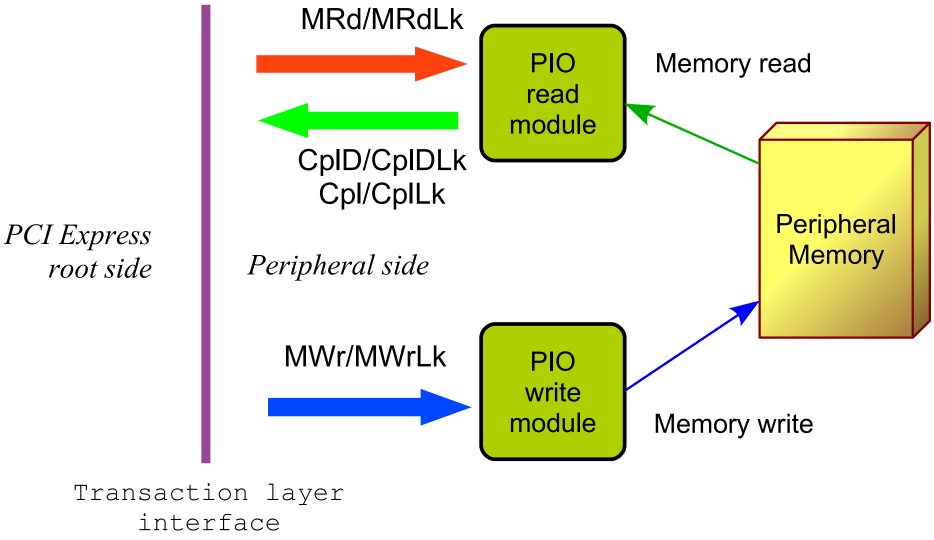


Figure 3-9 Basic PIO module set-up

For DMA function, the basic PIO data processing is formally mirror-duplicated, as figure 3-10 shows. The DMA read will be done in memory write TLPs and need no completion or acknowledge. DMA write is a request-acknowledge process, which is done by issuing read requests and storing the payload data in completion. Because the limitation from `MAX_READ_REQUEST_SIZE`, usually the DMA write is carried out in several MRd TLPs and several CplD TLPs. The completions are distinguished with the TAG field to ensure the correct storage destination.

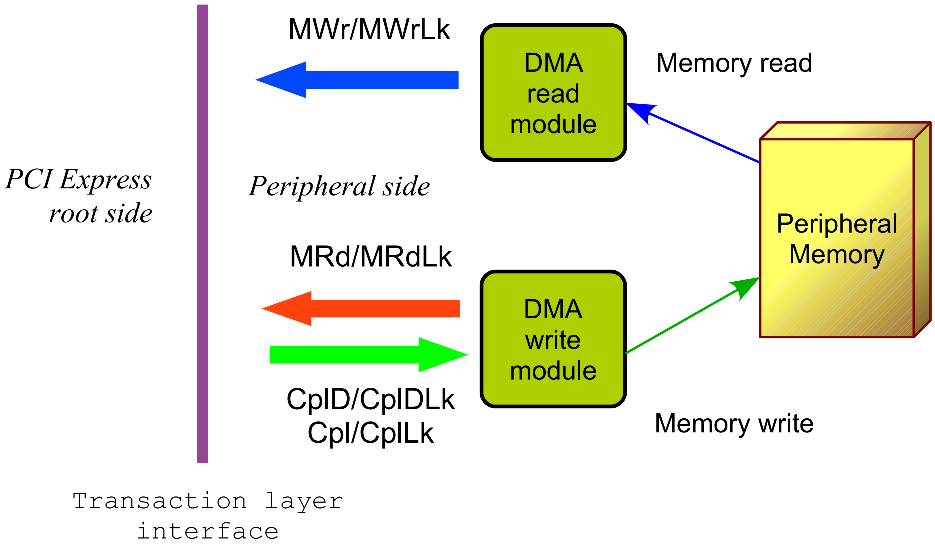


Figure 3-10 Basic DMA module set-up

The PIO module can work alone. But the DMA function needs the PIO module as a prerequisite, because the initial DMA command, in descriptor, must be told to the peripheral in PIO write. And if polling mode is used for DMA DONE status, it will be done in PIO read. This means, we have both modules in our DMA design.

The last two figures are quite simplified module scenarios, emphasizing the interfaces with the transaction layer of the PCI Express. If the simulation knows nothing about how the DMA is built up, the fore-going figures can be the image structure.

3.4 DMA engine development

The internal fabric of the Active Buffer logic is illustrated in figure 3-11. The DMA engine and the PIO module work together in the data path.

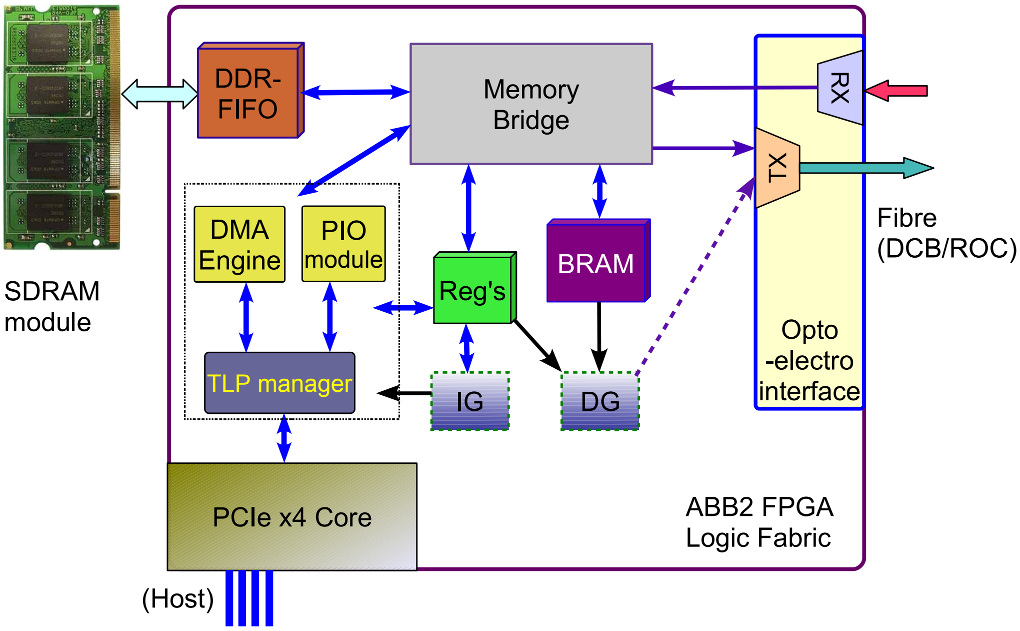


Figure 3-11 ABB logic fabric

As figure 3-11 shows, the hit packets coming through the DCB links are forwarded by the Memory Bridge to the DDR-FIFO module and go into a DDR2 SDRAM module. At the same time the Memory Bridge counts the packet number and makes necessary statistics. The Memory Bridge is a data switching module and composed of minimum logic. It has high-speed channels between the NI and the MI, and between the MI and the HI, but it has none between the NI and the HI, as figure 3- 1 shows. The host monitors the occupancy of the event buffer, or waits for the Memory Bridge to issue an interrupt when there are sufficient data to build the event. Thereafter the host starts a scatter-gather

DMA upstream (from the Event Buffer to the Host, or DMA read), moving the specific event or events to the host memory for preparation of further processing or transferring into the back-end networks.

The data path is built upon the PCI Express core coming along with Xilinx FPGA families. It manages the PHY and DL layers of the PCI Express so that our DMA design is focused on the TRN layer. [63] [64] [65]

This DMA design is considered as a central function of the Active Buffer. It is to make efficient data transfer between the host and the event buffer. DMA function is essential because the host computing resource is mostly spent upon the track finding, data storage and other complex algorithms, not enough for PIO data transfer. Another reason for DMA lies in the speed requirement. The incoming hit data flow sums up to 5 Gbps or even 10 Gbps for an Active Buffer board. PIO cannot afford such high bandwidth of data transportation. The best PIO read performance we made is 2.4 MB/s.

One of the memory target to make DMA test is the internal block RAM module, 32 KB assigned to BAR[1]. The other target is a FIFO structure, internally 32 KB, or externally 512 MB, assigned to BAR[2].

In the DAQ of CBM experiment, the upstream DMA on Active Buffer is emphasized because it is the major channel for hit data from front-end to back-end. The downstream DMA, from host to DCB/ROC, serves as the message channel, which does not demand high bandwidth of data transfer.

In the context of PCI Express and dual-channel DMA engine logic, the concurrent DMA operation is supported. PIO transactions are also supported and must be supported, because a DMA can only be started by a PIO write, aimed at the DMA control registers. The status register reading in the polling mode is correspondingly done via PIO read. For PIO zero-length read transactions, the completion is made after fetching data from BAR[0], address offset `+0x0`. BAR[0] is chosen to fill the zero-length read request because it is the common BAR for all DMA designs and there is no side-effect to read any of the registers in it.

Zero-length DMA is also supported for a completeness of design and avoidance of bad user DMA commands. IG (**I**nterrupt **G**enerator) and DG (**D**ata **G**enerator) are for debug purposes and the details can be found in *Appendix **A.7*** and *Appendix **A.8***.

The DMA engine block diagram is illustrated in figure 3-12.

Inside the DMA engine depicted in figure 3-12, five virtual channel buffers are applied for the Tx arbitration, DMA downstream channel, DMA upstream channel, PIO read channel, INT channel and (epoch) marker index channel. PIO write does not need specific channel buffer because the `MWr` TLP of PIO write transaction is posted transaction packet and the payload inside is directed to the memory space. The benefit from channel buffers lies in the possibly larger utilization of the PCI Express duplex bandwidth.

The DMA engine has two independent channels of opposite directions, one for upstream (DMA read, from the peripheral to the host), and one for downstream (DMA write, from the host to the peripheral). These two channels run simultaneously and hence, the bandwidth resource in both directions is well utilized.

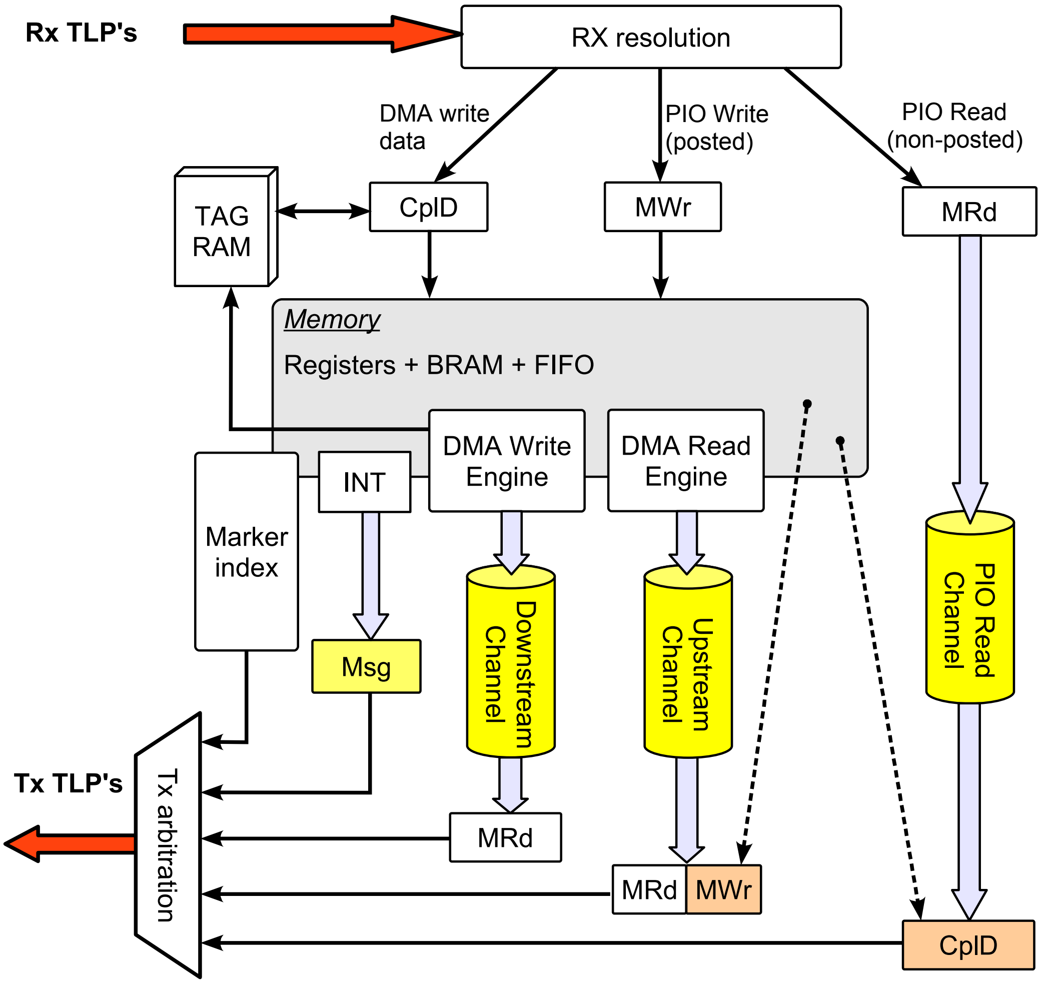


Figure 3-12 DMA engine block diagram

The event buffer may not have sufficient data for a DMA read transaction and this case is handled by time-out mechanism. DMA transaction is also able to be terminated when a time-out happens and is detected. DMA-level time-out does not hurt the system.

3.4.1 DMA procedure

Figure 3-13 illustrates the upstream DMA (DMA read) procedure in FPGA, and figure 3-14, the downstream DMA (DMA write). They are both demonstrated with 2 descriptors. DMA in more than 2 descriptors can be deduced from figure 3-13 or figure 3-14.

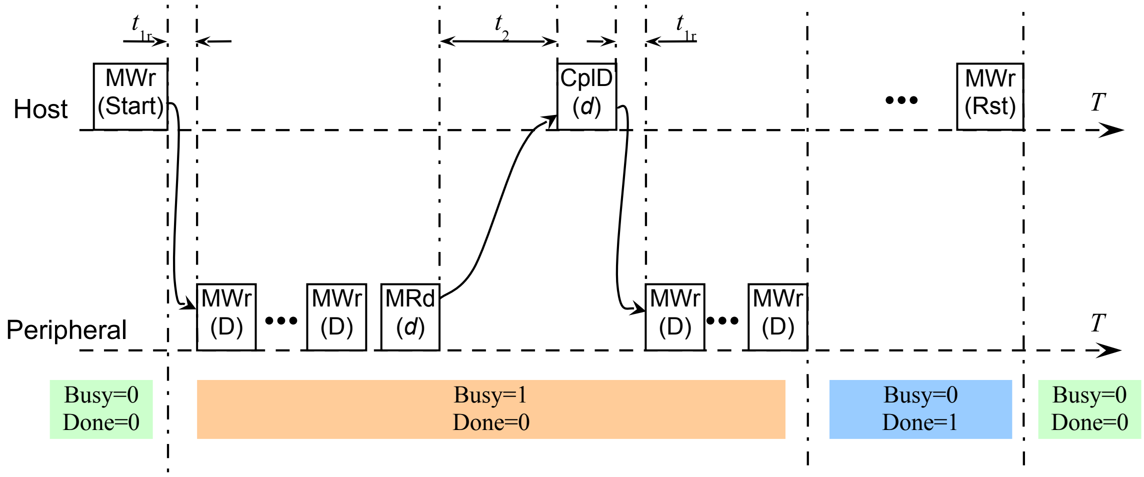


Figure 3-13 Upstream DMA (read) in 2 descriptors

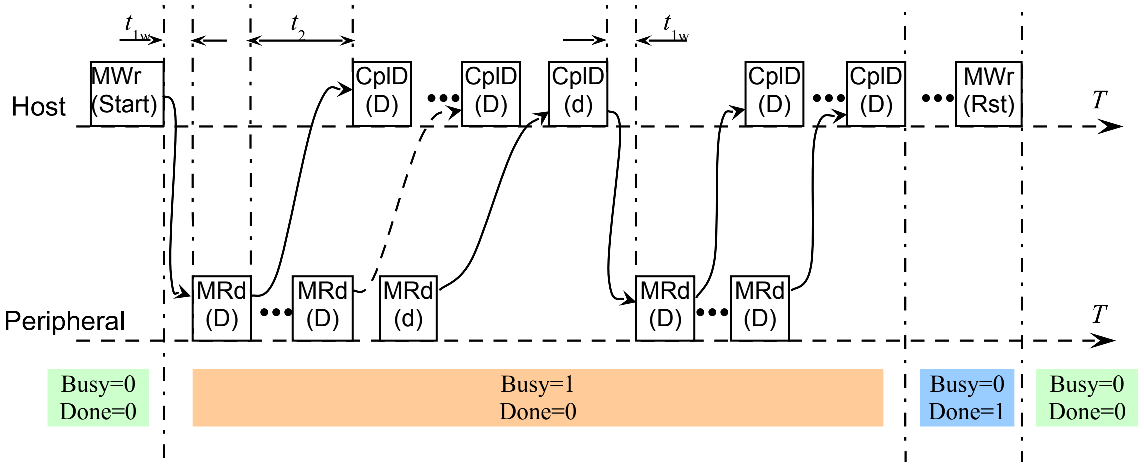


Figure 3-14 Downstream DMA (write) in 2 descriptors

(D) means the TLP carries payload for data. (*d*) means the TLP is related to a descriptor.

t_{1r}: latency from DMA read start command to DMA action at Tx, 296 ns.

t_{1w}: latency from DMA write start command to DMA action at Tx, 160 ns.

t_2: latency from Tx MRd to Rx CplD, around 1136 ns in our test system.

If a DMA transaction is started after another finished DMA but without Reset command before the Start command, both Busy and Done bits will be ‘1’, which indicates an illegal state. Done bit can only be cleared by a Reset command.

3.4.2 DMA packet division

Every DMA must be done in one or more TLPs. Read-request for DMA write and write-request for DMA read. For usual DMA size larger than the MAX_READ_REQUEST_SIZE or MAX_PAYLOAD_SIZE, DMA transaction will be broken down into several TLPs. And these TLPs should not violate the maximum size parameters as well as RCB. So for an arbitrary DMA with arbitrary size and start offset, the partitioning of TLPs is deliberated for the first and the last, dependent upon the start address and the DMA length.

The RCB affects the data receiving for DMA write. If the possible splitting of data payload into two or more TLPs (CplDs) is not correctly handled, data loss can happen.

Concerning the address-length combination (ALC), there are a few different possibilities, which should be treated in different ways. The DMA state machine mainly cuts the whole DMA, usually more than 4 KB, into some units and sends them out in corresponding TLPs (MRds for downstream DMA and MWrs for upstream DMA). Figure 3-15 shows typical cases for ALC. We name the first block of non-integral of Max_TLP_Size a Head and the non-integral last block a Tail. Correspondingly the middle block is named Body. Max_TLP_Size here can indicate the MAX_READ_REQUEST_SIZE (for downstream DMA) or MAX_PAYLOAD_SIZE (for upstream DMA). For example, case 3-15(a) has no Head, case 3-15(e) has no Tail, and case 3-15(d) has no Body.

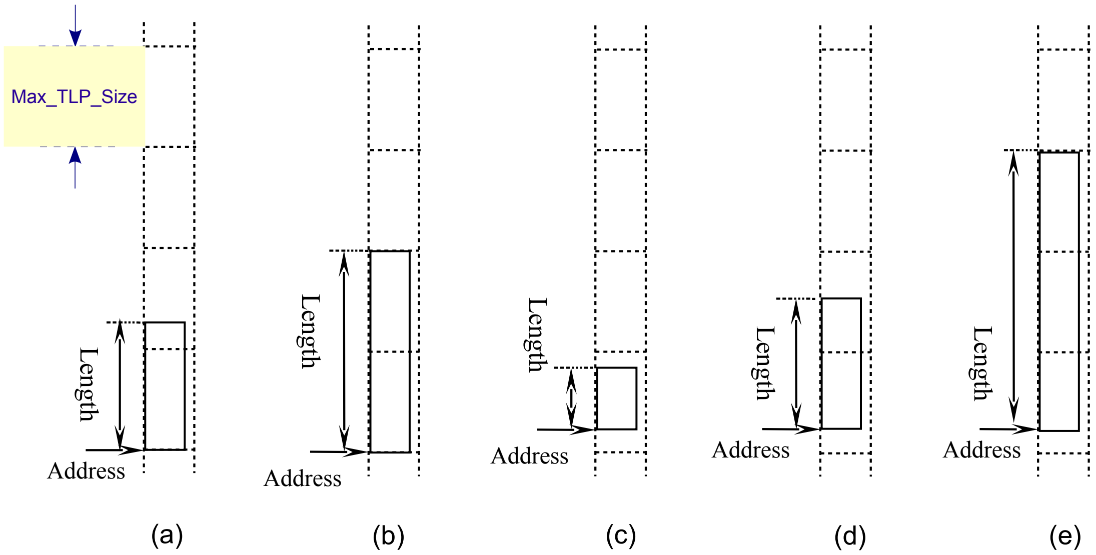


Figure 3-15 Some typical cases of address-length combinations

If we let

$$N = [DMA\_Size/Max\_TLP\_Size],$$

which is rounded result of the division, the legal number of tasks can be

N, N+1 or N+2.

For example, a DMA size of `0x108` for a Max_TLP_Size = `0x80`, N = 2. So possible division patterns can be 2, 3 or 4, provided the ALC does not straddle the 4KB boundary.

However for DMA size of `0x50`, N = 0, possible division patterns can be 1 or 2, because a none-zero DMA size can not be done in 0 packet transaction.

Concerning the possibilities for the fitting, altogether there are 8 possibilities for the ALC, as listed in table 3-6, the ALC truth table. 0 means no and 1 means yes.

Table 3-6 DMA division in terms of ALC

Leng_M:Leng_U:Leng_N	ALC_B: ALC_T	Head-Body-Tail
0 0 0	x x	0 0 0
0 1 0	0 0	0 1 0
	0 1	1 0 1
	1 0	- - -
	1 1	- - -
0 0 1	0 0	- - -
	0 1	1 0 0
	1 0	1 0 0
	1 1	1 0 1
0 1 1	0 0	- - -
	0 1	1 0 1
	1 0	1 1 0
	1 1	1 1 1
1 x 0	0 0	0 1 0
	0 1	1 1 1
	1 0	- - -
	1 1	- - -
1 x 1	0 0	- - -
	0 1	1 1 1
	1 0	1 1 0
	1 1	1 1 1

We can see in table 3-6, Head-Body-Tail can not be ‶001″ or ‶011″.

Leng_M: DMA size ≥ 2×Max_TLP_Size.

Leng_U: DMA size ≥ Max_TLP_Size, subordinate to Leng_M.

Leng_N: DMA size is not integral of Max_TLP_Size.

ALC_B: Extra body block is carried in.

ALC_T: There is tail block in ALC.

Head: the TLP size of the first memory request, can be of non-2^n-aligned, e.g. `0x13` DW.

Body: the TLP size of memory requests in the middle, must be of 2^n-aligned, e.g. 0x40 DW for Max_TLP_Size = 0x100.

Tail: the TLP size of the last remained memory request, can be of non-2^n-aligned, e.g. 0x0B DW.

Based on table 3-6, the logic cuts the DMA transaction defined by the current descriptor into proper packets.

3.4.3 TAG management for DMA write

DMA write is implemented by issuing read request TLPs (MRds) to the root and expecting the completion TLP (CplD). This kind of transactions is non-posted. For the pipelined read requests, the completions must be correctly associated to their read requests. Such mechanism in PCI Express is realized with TAG field. TAG is 8-bit wide.

TAG RAM, built on block RAM resource in FPGA, is for referencing the requests and acknowledgements. Later on, for 64-bit version, TAG RAM is replaced with FF pseudo RAM, which has faster output under high clock rate, e.g. 125 MHz.

The association is actually a table data structure. So we use a TAG RAM as the memory element. 8 bits of TAG field can provide 256 TAGs, and we use 7 bits for the DMA write data request, altogether 128 TAGs. Another bit is used for distinguishing descriptor requests and payload data. There is possibility in PCI Express to extend the TAGs volume, but for simplicity we do not take that feature. Practical tests show that 128 TAGs is sufficient for our applications. This TAG RAM has one port only for write (update) and the other port, both for read and write. For DMA write, a TAG RAM is set with 128 entries, each entry corresponds to a data TAG. By issuing a data request, the corresponding entry is filled with the local address information, to which the returned payload in CplD should be written. Usually the the completion for a request (MRd) is done in multiple CplDs. So after every data CplD, the TAG RAM is updated with the new address information.

The update write to the TAG RAM should be carefully processed to guarantee correct content in the TAG RAM entries. Special logic is implemented when two consecutive CplDs associated with the same TAG come as back-to-back transactions and the first one has little data payload (1 DW or 2 DWs), as figure 3-16 presents. The difficulty lies in insufficient cycles to make the TAG RAM content prepared for the second CplD. Such situation is a frequent case for 64-bit interface DMA. An additional register as wide as the TAG RAM entry is used to temporarily save the address information when a CplD has little payload and is not the last CplD for the request. This information is directly used if the next CplD is associated with the same TAG and comes back-to-back, so that the TAG RAM is bypassed.

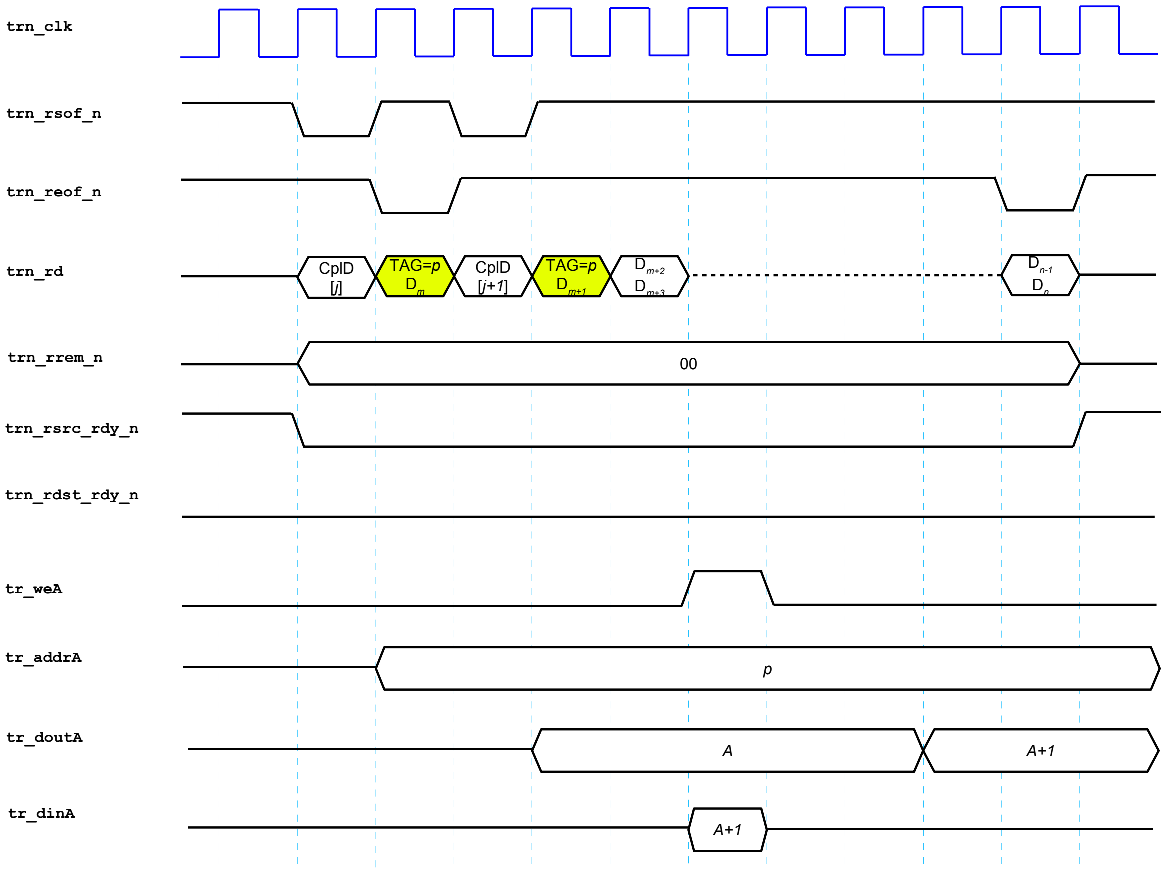


Figure 3-16 Tight TAG RAM update

For high-speed design (250 MHz transaction clock rate), the output of this BRAM must be registered. And in 64-bit interface DMA, this BRAM is replaced with register-RAM, which can sustain the 125 MHz clock rate as well as decrease one cycle for the TAG reading. 64-bit interface needs extra processing because the address information (TAG) and data come in the same cycle for a CplD.

3.4.4 Channel buffers and Tx arbitration

Different TLPs and user commands go to different channels for processing, because they need the Tx output resource. The channel buffer is 128 bits in width and maximum 16 entries in depth. Every request is compressed into a 128-bit abstract, which includes all the necessary information of the TLP being built and issued. The channel buffers are aggregated at Tx port for a fair and dead-lock-free arbitration.

Tx arbitration uses weighted LRSF (**L**east **R**ecently **S**erviced **F**irst) policy. For every requester one or more priority registers are set. Multiple priority registers are for the case of weighted arbitration. The more priority registers one requester has, the higher weight it possesses. We have 8 priority registers and they are assigned to the five channels as table 3-7.

Table 3-7 Channel weights for Tx arbitration

Channel	INT	PIO read	DMA write	DMA read	Marker
Priority register number	3	2	1	1	1

Suppose there are N priority registers, each register being N-bit. Maybe some of them belong to the same requester for weight grading. These N registers, numbered 0 to N-1, are initialized to values

$$p_i = 2^i,\ \ 0 \le i \le \text{N-1}.$$

In this way they build up an N-dimension upper-triangle matrix

$$\begin{pmatrix} 1 & 1 & 1 & . & . & . & 1 & 1 \\ 0 & 1 & 1 & . & . & . & 1 & 1 \\ & . & . & . & . & . & . & \\ & . & . & . & . & . & . & \\ 0 & 0 & 0 & . & . & . & 0 & 1 \end{pmatrix}$$

If a requester has request to compete, its competition value is equal to its priority register value; otherwise, 0. To decide the winner if there are at least one request, all the competition values are OR'ed, and the winner has the same value as the OR'ed value. After a winner is decided, its priority register is put to the lowest (`000...01`), meanwhile the priority registers of all other partners with lower priority than the winner is increased. The increasing is done just by logic left shifting the priority register. After these two cycles, new arbitration is available again. In our case, N = 8.

This arbitration is easy to implement in logic and has very good performance, because the arbitration takes only one cycle, and the priority register updating can be paralleled, also taking only one cycle. And as we can see, the space cost is $O(N^2)$, which is acceptable in our FPGA development.

The weighted LRSF fulfils the fundamental requirements for a fair arbitration, no dead-lock and no starvation. The system has maximum payload size limitation (`MAX_PAYLOAD_SIZE`) and the time-out protection, so the Tx channel will not be occupied by a single requester. Dead-lock is avoided because there will never be two requesters with the same priority level. Starvation is avoided by means of priority degrading of the serviced requester and so at most N-1 arbitrations have to be waited by the channel with the lowest priority before it is serviced.

There are quite a lot of arbitration strategies being studied. [66] We choose this arbitration for its easy scalability because during the Active Buffer development, probably more channels are to be appended to take part in the competition for the Tx. In the initial version for example, Epoch Marker was not assigned a channel.

Arbitration exception may happen when a channel is incorrectly reset while it is just granted. In this case, the arbitration will let the Tx module issue an invalid output and the PCI Express interface will crash. Such situation should be avoided by the software, but the hardware design should reserve tolerance to it. We establish a small vigilance module that detects the unexpected cancelling of the arbitration requests. The exception leads to an arbitration reset and the software can read the global error register to check if it has happened.

The disadvantage for weighted LRSF arbitration is the poor real-time capability. Obviously even the most urgent request with very heavy weight cannot be guaranteed to be granted the resource at once. All its priority registers are sure sooner or later sinking down to the lowest values. Real-time and dead-lock-free are commonly a paradox for arbitration. Our system has lower requirement for real-time response and the chosen arbitration (LRSF) is proved efficient.

3.4.5 DMA descriptor and DMA commands

A DMA descriptor consists of parameters such as the source address, destination address, DMA size in bytes, address to the next descriptor and the DMA mode control. The address for next descriptor points to the next DMA parameters resident in host memory, so that the DMA engine fetches the next DMA command after current DMA is DONE. In this way, the DMA transfer is still automatic and meanwhile providing more flexibility to the user.

Table 3-8 lists the descriptor definition for a DMA channel.

Table 3-8 Descriptor definition

Name	R/W	Address Offset	+3								+2								+1								+0							
			31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Peri_Addr_H	R+W	+0x00	Higher 32-bit of DMA Peripheral address																															
Peri_Addr_L	R+W	+0x04	Lower 32-bit of DMA Peripheral address																															
Host_Addr_H	R+W	+0x08	Higher 32-bit of DMA Host address																															
Host_Addr_L	R+W	+0x0C	Lower 32-bit of DMA Host address																															
Next_BDA_H	R+W	+0x10	Higher 32-bit of next descriptor address																															
Next_BDA_L	R+W	+0x14	Lower 32-bit of next descriptor address																															
Length	R+W	+0x18	DMA transaction byte count																															
Control	W	+0x1C							V	Last				UPA		BN			AInc							End	Rst (0x0A)							
Status	R	+0x1C							V	Last				UPA		BN			AInc							End				TO			Busy	Done

BN[2:0]: Encoded BAR Number. Default is “000”.

V: Valid Control. A special bit for DMA channels. If set, the DMA engine uses this Control word; else, the DMA engine takes the previous Control word of that channel for current transaction. Even the reset command must also set this bit. Default value is ‘0’. Note: for the first descriptor this bit must be asserted.

Last: If set, the current descriptor is the last one for the current DMA chain. After the descriptor is processed, the DMA engine stops. Default value is '0'.

UPA: Use PA (**P**eripheral **A**ddress) from each descriptor, otherwise, use calculated PA. Default value is '0'. But for the initial descriptor, PA is always used, therefore this bit can be of any value for the first descriptor.

AInc: Address increments. If set, the address of the endpoint side increases by four for every double word transfer. Default value is '1'.

End: If set, the DMA engine is paused after this descriptor is processed. This means, the engine can be resumed afterwards. Default value is '0'.

Rst: Channel can be reset by write "0x0200000A" to the corresponding Channel Control register. These bits return always 0x0 by reading.

TO: Time-out signal. If asserted, this bit indicates that a time-out event has occurred during the DMA operation. Default value is '0'.

Busy: DMA engine is running. Default value is '0'.

Done: DMA engine is already finished. Default value is '0'. Once asserted, this bit never changes until a channel reset command comes.

A write to a DMA channel control register with the valid bit asserted is a DMA command. Bits like Last, UPA and AInc are the parameters of the command. Rst bits and End bit are the type of the command. Concerning the bits V, End and Rst, possible types of commands are listed in table 3-9.

Table 3-9 DMA commands

Command	V	End	Rst[7:0]	Description
Reset	1	x	0x0A	Reset the DMA channel state.
Start	1	0	0	Start/Resume DMA transaction.
Stop	1	1	0	Pause the current running DMA transaction, if any.
Redo	0	x	x	Repeat the last DMA command, which had the V bit set.

x*: don't care.*

3.4.6 DMA Status

For a DMA operation, the DONE status identification is important for efficiency and stability. In our DMA design, DONE status can be detected by polling the DMA status register or by servicing interrupts.

We count the outgoing TLPs for calculating the DMA DONE status. If it is not DONE, it is in BUSY or in IDLE status.

Such counting is done differently for DMA read and DMA write. For DMA read, if all the request abstracts are issued, the DONE status can be asserted. For DMA write, we have a bit map, corresponding to the read requests (MRd) that are issued. Every time all data is completed for one request, the very bit is filled with `1`. After all bits are `1`, the DONE status can be asserted. Using the bit map for DMA write saves the resource for adding and comparing of 32-bit vectors.

Table 3-10 lists the possible states for the DMA engine.

Table 3-10 DMA statuses

State	Busy	Done	Description
Idle	0	0	DMA channel is idle, ready to start new DMA transaction.
Busy	1	0	DMA is running.
Done	0	1	A DMA is already finished.
Unreset	1	1	Abnormal state. A previous DMA is finished but the state is not reset before a second DMA is started.

3.4.7 Transaction layer interface: 32-bit vs. 64-bit

Besides the lane number difference, the transaction layer interface can have 32-bit bus width or 64-bit bus width from Xilinx PCI Express core. The initial DMA design was with 32-bit interface, as in ABB1, MPRACE2 (**M**ulti-**P**urpose **R**econfigurable **A**ccelerator/**C**omputing **E**ngine - version **2**). Later on, for a looser timing constraint, we transported to 64-bit interface, as in AVNET PCIE board and ML605 board.

The major difficulty to make such transport lies between the 3rd and the 4th DW of the TLP. For TLPs with payload, 32-bit addressing MWr and CplD, the 3rd DW contains the address information and the 4th contains the first DW payload. In the 32-bit interface, these two DW come in two consecutive cycles. But in 64-bit interface, they two come in the same cycle, which costs extra effort in redesign the pipelines and in crossing the DWs. Tables 3-11 and 3-12 illustrate the difference with tables 3-2 and 3-3. 'R' denotes reserved and those bits must be zero.

Table 3-11 Memory write request TLP format for 32-bit addressing in 32-bit bus interface

<table>
<tr><td colspan="8">Byte +0</td><td colspan="8">Byte +1</td><td colspan="8">Byte +2</td><td colspan="8">Byte +3</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="2">format
X0</td><td colspan="5">type</td><td>R</td><td colspan="3">TC</td><td colspan="4">R</td><td>T
D</td><td>E
P</td><td colspan="2">Attr</td><td colspan="2">R</td><td colspan="10">length</td></tr>
<tr><td colspan="16">requester ID</td><td colspan="8">tag</td><td colspan="4">last DW BE</td><td colspan="4">1st DW BE</td></tr>
<tr><td colspan="30">address [31:2]</td><td colspan="2">R</td></tr>
<tr><td colspan="32">data #0</td></tr>
<tr><td colspan="32">(<i>further payload data or no more data</i>)</td></tr>
</table>

Table 3-12 Completion TLP format in 32-bit bus interface

<table>
<tr><td colspan="8">Byte +0</td><td colspan="8">Byte +1</td><td colspan="8">Byte +2</td><td colspan="8">Byte +3</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>R</td><td colspan="2">format
X0</td><td colspan="5">type</td><td>R</td><td colspan="3">TC</td><td colspan="4">R</td><td>T
D</td><td>E
P</td><td colspan="2">Attr</td><td colspan="2">R</td><td colspan="10">length</td></tr>
<tr><td colspan="16">completer ID</td><td colspan="3">CS</td><td>R</td><td colspan="12">byte count</td></tr>
<tr><td colspan="16">requester ID</td><td colspan="8">tag</td><td>R</td><td colspan="7">lower address</td></tr>
<tr><td colspan="32">data #0</td></tr>
<tr><td colspan="32">(<i>further payload data or no more data</i>)</td></tr>
</table>

The difference between 32-bit and 64-bit interfaces brings noticeable modification for the TLP resolution modules, MWr and CplD. Especially for CplD resolution, which is critical for DMA write, the TAG resolution/target address fetching requires faster processing, because the TAG and the first payload DW come in the same cycle. More pipelines are inserted and the TAG RAM is changed to FF style to win one cycle ahead than the BRAM style.

3.5 Transaction-layer verification

For complex data path logic, the verification, or feed-back simulation, plays an important role in searching the hidden hazards in the early phase of the development. For our PCI Express DMA project, the simulation environment is built to emulate the behaviour of the PCI Express transaction layer with respect to the user's intention. Such environment is supposed to have no knowledge about the fabric logic, treating it as a black-box. It pours TLP soup to the DMA logic and checks the TLPs flowing out. The output TLPs are compared with the expectation. So the atomic unit on the transaction layer interface is TLP. The user intention is represented in form of packets. The simulation is initiated to the idle state, in which no transactions take place in Rx or Tx bus. Before this state, some basic read (HW version) and write (INT enable) are done. Figure 3-17 shows the procedure of the simulation bench.

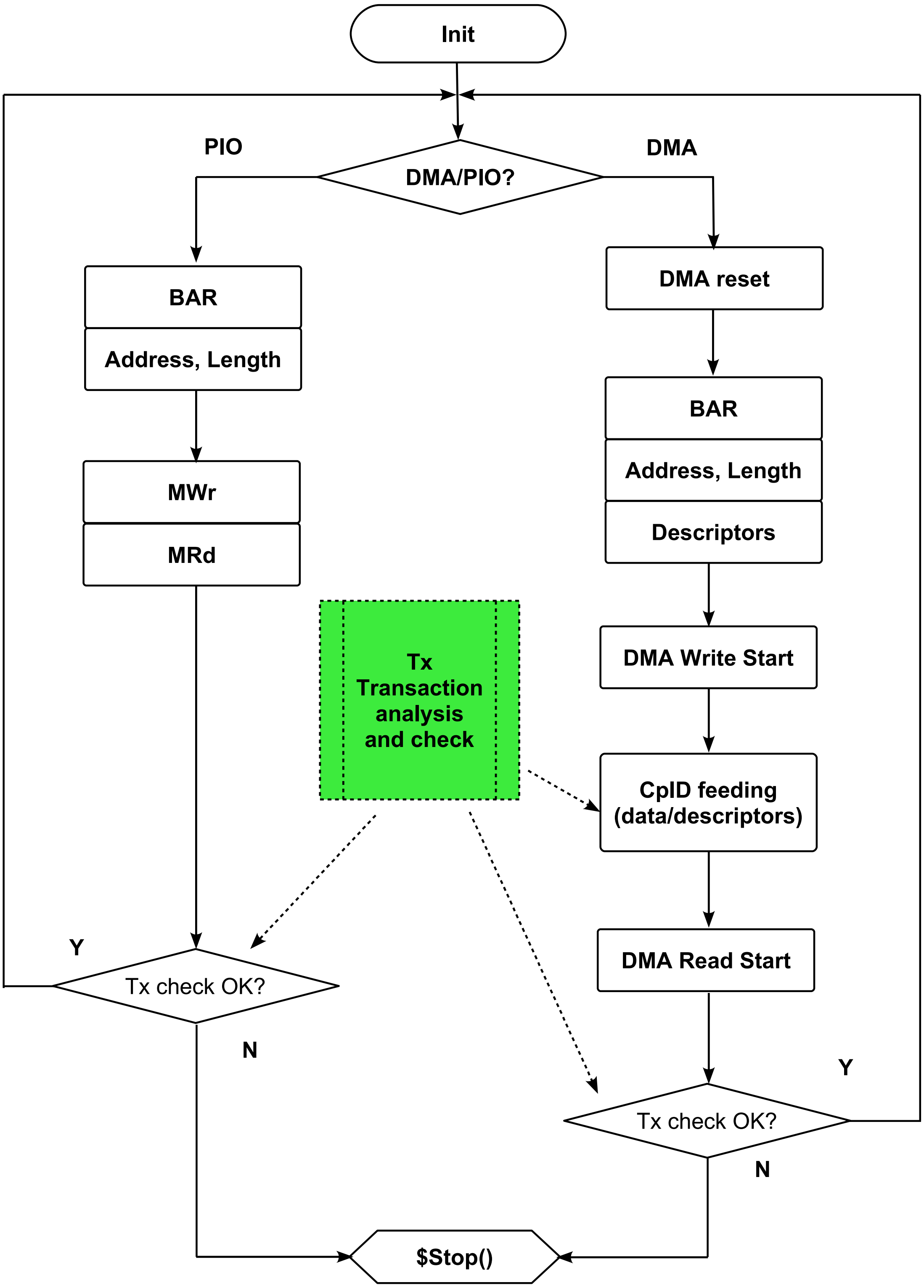


Figure 3-17 Simulation flow

Tx transaction analysis is a set of parallel processes concurrent with Rx procedures.

The verification is in unit of loops. A verification loop can be PIO or DMA. A PIO loop may target on registers or event memory. A DMA loop can only target on the bulk memory spaces. Because the DMA engine is used on different systems, the memory spaces are categorized into two types, RAM type and FIFO type. RAM type is implemented with BRAM primitives. FIFO type includes the large-size SDRAM event buffer and the small-size built-in FIFO in Xilinx FPGAs. The FIFO type memory is different with the RAM type in the way that it has full and empty states and the expected DMA logic should take care of these states to avoid buffer overwrite and invalid data reading (under-read), or dead-lock waiting.

This simulation environment is intended to simulate the behaviour of PCI Express transaction layer interface in the Active Buffer, as figure 3-18 illustrates. The simulation environment generates TLPs for the Rx port of the DMA logic. These TLPs are related to DMA or PIO transactions. At Tx port, the TLP checker knows what TLPs should come out, exactly of the format and the content (if a TLP is payloaded). And, if the TLP coming out of Tx port is requesting data, the TLP generator provides the Rx data in proper and flexible way according to PCI Express specification. Both Rx and Tx TLPs are sometimes out of order, due to the arbitration mechanism behind. And to test the robustness of the logic, some arbitrary harmless MWr and MRd TLPs are inserted by the TLP generator during loops.

The stimulus is depicted in figure 3-18. The transaction can be data transfer (PIO/DMA), status polling, or parameter configuration.

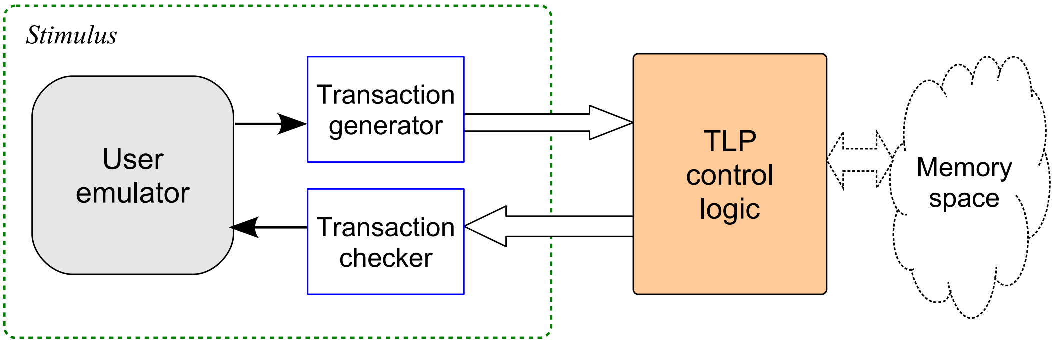


Figure 3-18 Simulation environment on the transaction layer of PCI Express

Because the correctness of the PCI Express core is commercially verified, our verification work does not include it for a shorter verification time. The simulation level is on the transaction layer of the PCI Express and it deals with the TLP as the packet unit.

The basic approach is to read the same memory section after writing it with specific data sequence. The data identification is carefully checked. Also the TLP format is checked, e.g. the HEADER field, the SOF and EOF signals, the agreement between the TLP Length field and the actual length, the address-length combination (crossing 4KB memory boundary is illegal in PCI Express specification), etc.

Table 3-13 lists major test purposes in the verification.

Table 3-13 Simulation objectives

Test type	Description	Reason or purpose
PIO simulation amount equality	Read data amount should be the same as the write in PIO simulation.	Missing data at the end of read transaction might be ignored by the integrity checker.
DMA simulation amount equality	Read data amount should be the same as the write in DMA simulation.	Missing data at the end of read transaction might be ignored by the integrity checker.
PIO with 64-bit address	PIO MWr and MRd can be 3-DW header or 4-DW header.	Support for addressing systems other than 32-bit.
DMA with 64-bit address	DMA PA and HA can be 32-bit or 64-bit wide.	Support for addressing system other than 32-bit.
Random/increasing payload data sequence	Data content randomized or sequenced.	Different data sequences have different simulation effects.
4KB boundary straddling checking - Tx	Checks the violation of 4KB straddling for the Tx outgoing TLPs.	4KB boundary violation is not permitted in PCI Express.
4KB boundary straddling checking - Rx	Checks the violation of 4KB straddling for the Rx outgoing TLPs.	Verification for verification.
Tx TLP format checking	Every field of the Tx outgoing TLP must be legal and the coherence should be always maintained.	Ill-formatted TLPs to the host often causes system crash.
Rx TLP format checking	Checks that the simulation generates correct TLPs.	Verification for verification.
PIO simulation BAR: BAR0, BAR1, BAR2	PIO target memory region.	All BARs should be covered in PIO transactions.
DMA simulation BAR: BAR1, BAR2	DMA target memory region.	BAR1 and BAR2 should be covered in DMA transactions.
Back-to-back transactions	CplD and MRd (status) are mixed and back-to-back sometimes.	Used to be problem maker for CplD in DMA write.
TLP details: Requester ID, TAG, completer ID, etc.	Correct requester-ID and TAG should be used in the TLP.	All details of the TLP should be coherent.
Zero-length PIO	Zero-length memory read by PIO is supported and simulated.	Essential in memory flushing.
Zero-length DMA	Zero-length memory read by DMA is supported and simulated.	Exception test. DMA engine should be robust for zero-length DMA request.
Max-Read-Request-Size: 128, 256, 512, 1024, 2048, 4096 bytes	Maximum thresholds for read.	Generalized for different system configurations.
Max-Payload-Size: 128, 256, 512, 1024, 2048, 4096 bytes	Maximum thresholds for write.	Generalized for different system configurations.

(*Table 3-13 to be continued on the next page*)

Table 3-13 Simulation objectives (*continued*)

Test type	Description	Reason or purpose
`Request-Completion-Boundary: 64, 128 bytes`	The completions in DMA write payload are allowed to be returned in several TLPs.	To test that the DMA engine can deal with different RCB parameters.
`DMA simulation with single descriptor`	Single-descriptor DMA.	Most commonly in KERNEL memory mode with small-size DMA.
`DMA simulation with multiple descriptors`	Multiple-descriptor DMA.	Most commonly in USER memory mode with large-size DMA.
`DMA descriptor MRd completed in 1 CplD`	Subsequent descriptor completed in a single CplD.	Most normal case.
`DMA descriptor MRd completed in 2 CplDs`	Subsequent descriptor completed in multiple CplDs.	Abnormal case but must be considered.
`DMA pause and resume`	DMA can be paused and then, resumed again or reset.	For a more flexible control upon the DMA procedure.
`DMA status checking - Polling`	Check the DMA status by MRd.	Status must be accessible via PIO read and present correct values.
`DMA status checking - INT`	If the DMA DONE INT is enabled, Msg will be sent to the root when the DMA is done.	For a higher performance.
`DMA reset`	Simulate the reset effect of DMA reset. All status should return to the initial values.	DMA reset should put state to the correct one.
`INT enable and disable`	Enables or disables interrupts.	Interrupt control.
`Interrupt equality checking between INT and` $\overline{\texttt{INT}}$	INTA and $\overline{\text{INTA}}$ numbers are counted every time they change. If the difference is greater than 1, error is reported.	The assert and deassert numbers must be balanced to avoid the system crash.
`Death detection: outputs`	Output should not die.	Check the Tx output dead-lock.
`Death detection: loops`	Loop counting-up should not die.	DMA hanging cannot be detected by output death detection because the status polling is regularly sent and acknowledged.
`Rx source throttling`	trn_rsrc_rdy_n asserted and deasserted during a TLP.	Standard supporting.
`Tx destination throttling`	trn_tdst_rdy_n asserted and deasserted during a TLP.	Standard supporting.
`Rx flow control and no flow control`	Large time-scale with and without flow control for Rx.	Simulate the Rx source throttling.
`Tx flow control and no flow control`	Large time-scale with and without flow control for Tx.	Simulate the Tx destination throttling.

PIO loop covers zero-length write and read. For DMA loop, zero-size DMA is also covered, in which

DMA engine should transfer no data and assert the DONE status bit at once.

For DMA loop, downstream transaction (DMA write) is executed before upstream transaction (DMA read). The data checking happens at the upstream process. In addition, the downstream and upstream DMA sizes are compared to see if they agree, which can not be discovered by the data integrity checking.

It is not practical, even not possible, to simulate the entire Event Buffer for its large size 256 MB or 512 MB. What we do is to use a smaller built-in FIFO in the FPGA, about 128 KB in size, for a behavioural replacement in simulation. And as we have discussed in chapter 2, the large-size FIFO behaves exactly like the standard FIFO. Therefore, such replacement in the simulation is sufficient and plausible.

Death detection is set to prevent the logic halt somewhere. This is necessary because when the logic is in a dead loop, no output is available for check and therefore, no data errors are reported.

The content of the data sequence can be random or sequential, random for a strengthened testing and the sequential for easier tracing the errors, if any. The totally random content gives faster convergence for simulation.

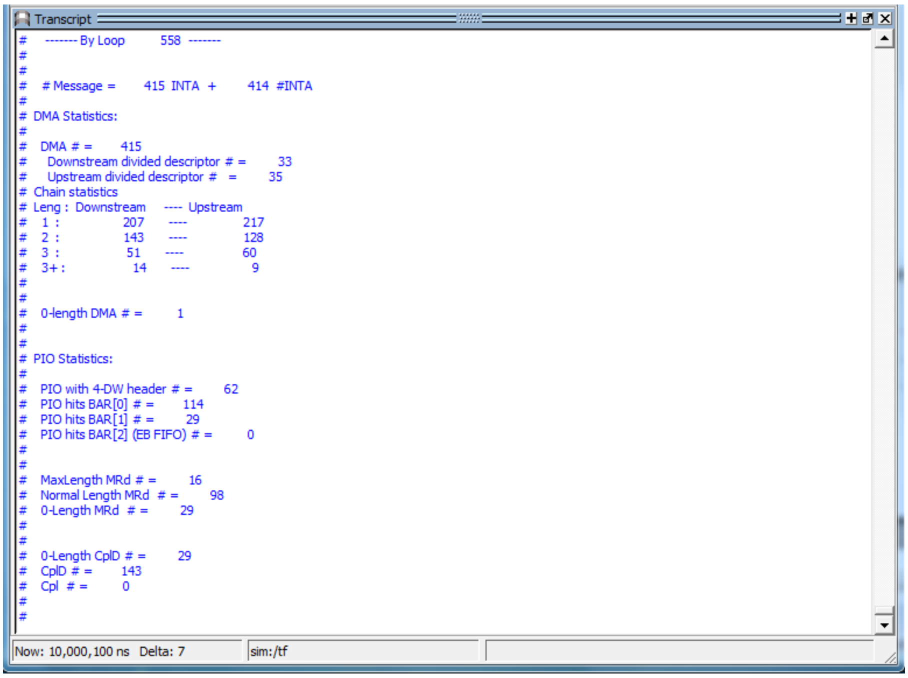
The simulation environment emphasizes the maximum behavioural coverage as well as fast convergence. Possibly more parameters, such as TLP length, back-to-back transactions or not, data flow throttling of Rx and Tx, and so on, are randomly varied. This also helps in fast finding of hidden logic bugs.

The current limitation for this simulation environment is that two or more loops can not be overlapped, because that introduces more complexity for the data verification. Also, the concurrent DMA transaction is not logically verified. That is directly tested by the software programs. It works without data errors and without dead-lock.

Source throttling at Rx port and destination throttling at Tx port are both covered. They are complied with the transaction layer specification from Xilinx PCI Express core.

During the simulation, statistics are displayed, such as DMA loop number, PIO DMA number, INT pairs, zero-length transaction number, multiple-descriptor DMA number, etc., as figure 3-19 shows.

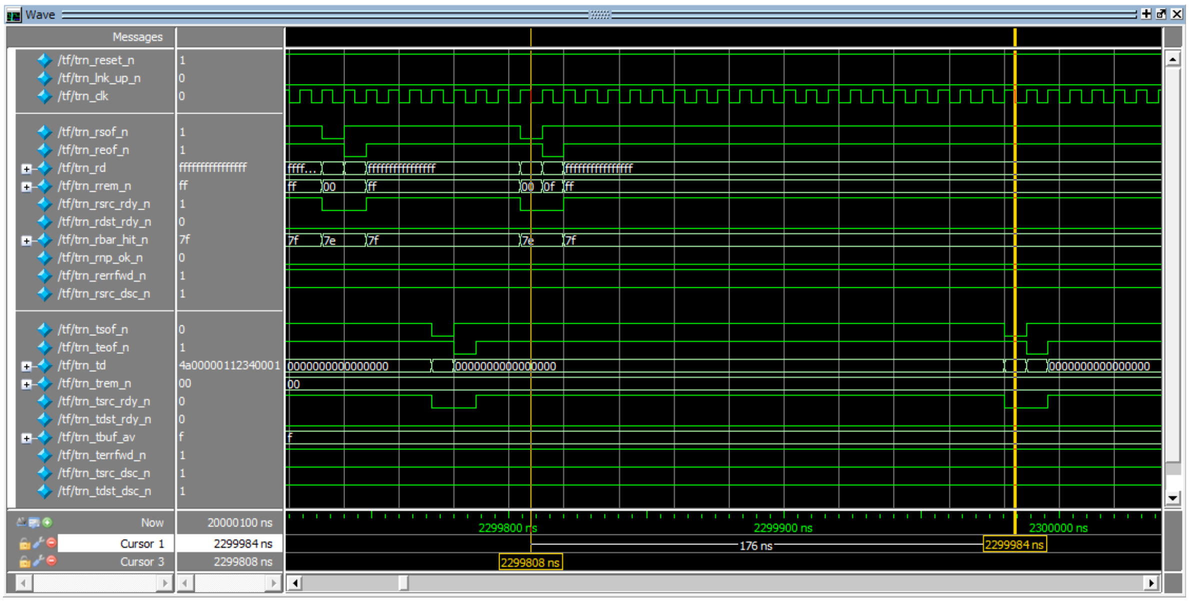
Figure 3-20 to 3-28 shows typical waveforms of the simulation under ModelSim SE 6.5.



```
#    ------- By Loop        558  -------
#
#
#   # Message =       415  INTA  +       414  #INTA
#
#  DMA Statistics:
#
#   DMA # =       415
#     Downstream divided descriptor # =       33
#     Upstream divided descriptor #   =       35
#  Chain statistics
#  Leng :  Downstream    ---- Upstream
#   1  :              207     ----             217
#   2  :              143     ----             128
#   3  :               51     ----              60
#   3+ :                14     ----               9
#
#
#   0-length DMA # =        1
#
#
#  PIO Statistics:
#
#   PIO with 4-DW header # =       62
#   PIO hits BAR[0] # =       114
#   PIO hits BAR[1] # =        29
#   PIO hits BAR[2] (EB FIFO) # =        0
#
#
#   MaxLength MRd # =        16
#   Normal Length MRd  # =       98
#   0-Length MRd  # =       29
#
#
#   0-Length CplD # =       29
#   CplD # =       143
#   Cpl  # =        0
#
#
Now: 10,000,100 ns   Delta: 7        sim:/tf
```

Figure 3-19 Transaction layer simulation statistics output

Figure 3-20 shows the PIO write and PIO read, as well as the completion (CplD) that comes 176 ns later.



```
Messages
/tf/trn_reset_n        1
/tf/trn_lnk_up_n       0
/tf/trn_clk            0

/tf/trn_rsof_n         1
/tf/trn_reof_n         1
/tf/trn_rd             ffffffffffffffff
/tf/trn_rrem_n         ff
/tf/trn_rsrc_rdy_n     1
/tf/trn_rdst_rdy_n     0
/tf/trn_rbar_hit_n     7f
/tf/trn_rnp_ok_n       0
/tf/trn_rerrfwd_n      1
/tf/trn_rsrc_dsc_n     1

/tf/trn_tsof_n         0
/tf/trn_teof_n         1
/tf/trn_td             4a00000112340001
/tf/trn_trem_n         00
/tf/trn_tsrc_rdy_n     0
/tf/trn_tdst_rdy_n     0
/tf/trn_tbuf_av        f
/tf/trn_terrfwd_n      1
/tf/trn_tsrc_dsc_n     1
/tf/trn_tdst_dsc_n     1

Now                    20000100 ns
Cursor 1               2299984 ns
Cursor 3               2299808 ns
```

Figure 3-20 PIO write, PIO read and its completion (CplD)

Figure 3-21 shows the DMA read starting. From the DMA start command to the the first payload TLP

(MWr) from the Tx side is 296 ns.

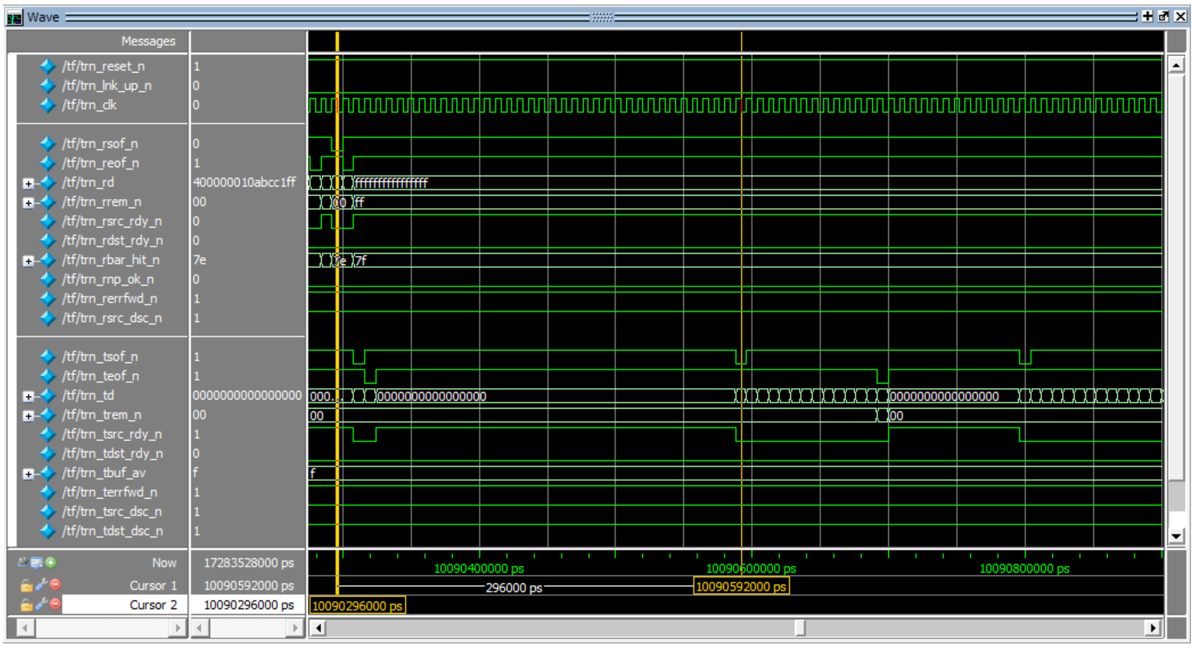


Figure 3-21 DMA read: initial descriptor and payload

Figure 3-22 shows the DMA write starting. From the DMA start command to the the first request TLP (MRd) from the Tx side is 160 ns.

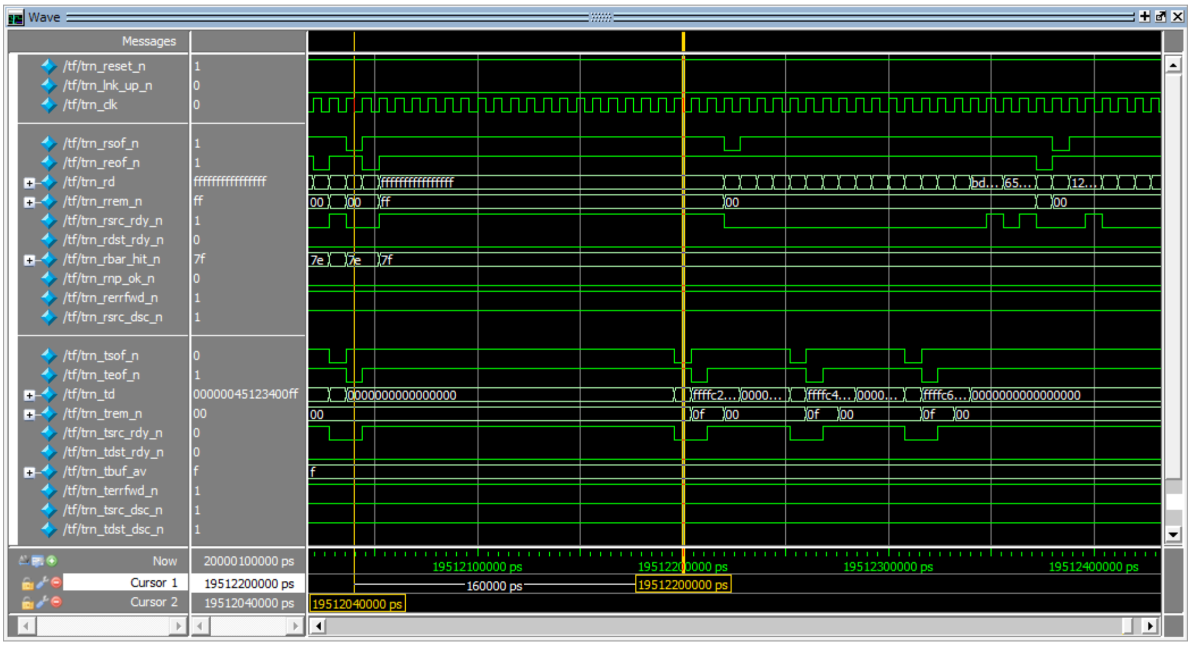
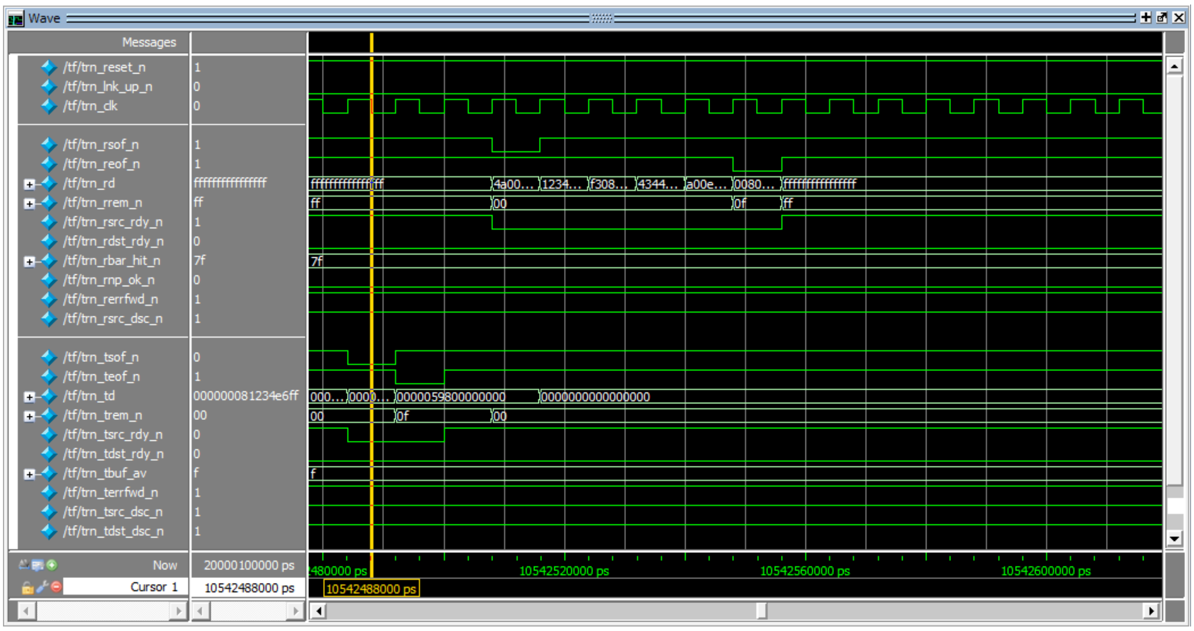


Figure 3-22 DMA write: initial descriptor and payload

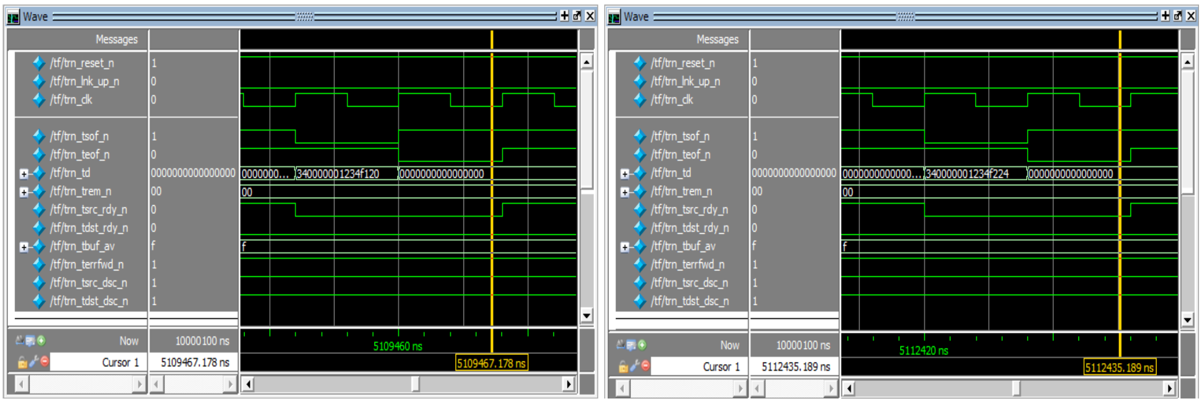
Figure 3-23 shows the subsequent DMA descriptor fetching. For simplicity, the simulation completes the Tx MRd almost at once. However in real tests, the latency between is around 1136 ns.



Messages	
/tf/trn_reset_n	1
/tf/trn_lnk_up_n	0
/tf/trn_clk	0
/tf/trn_rsof_n	1
/tf/trn_reof_n	1
/tf/trn_rd	ffffffffffffffff
/tf/trn_rrem_n	ff
/tf/trn_rsrc_rdy_n	1
/tf/trn_rdst_rdy_n	0
/tf/trn_rbar_hit_n	7f
/tf/trn_rnp_ok_n	0
/tf/trn_rerrfwd_n	1
/tf/trn_rsrc_dsc_n	1
/tf/trn_tsof_n	0
/tf/trn_teof_n	1
/tf/trn_td	000000081234e6ff
/tf/trn_trem_n	00
/tf/trn_tsrc_rdy_n	0
/tf/trn_tdst_rdy_n	0
/tf/trn_tbuf_av	f
/tf/trn_terrfwd_n	1
/tf/trn_tsrc_dsc_n	1
/tf/trn_tdst_dsc_n	1
Now	20000100000 ps
Cursor 1	10542488000 ps

Figure 3-23 DMA: derivative descriptor request

Figure 3-24 shows the Interrupt messages. The left is INTA and the right is $\overline{\text{INTA}}$.



Messages	
/tf/trn_reset_n	1
/tf/trn_lnk_up_n	0
/tf/trn_clk	0
/tf/trn_tsof_n	1
/tf/trn_teof_n	0
/tf/trn_td	0000000000000000
/tf/trn_trem_n	00
/tf/trn_tsrc_rdy_n	0
/tf/trn_tdst_rdy_n	0
/tf/trn_tbuf_av	f
/tf/trn_terrfwd_n	1
/tf/trn_tsrc_dsc_n	1
/tf/trn_tdst_dsc_n	1
Now	10000100 ns
Cursor 1	5109467.178 ns

Messages	
/tf/trn_reset_n	1
/tf/trn_lnk_up_n	0
/tf/trn_clk	0
/tf/trn_tsof_n	1
/tf/trn_teof_n	0
/tf/trn_td	0000000000000000
/tf/trn_trem_n	00
/tf/trn_tsrc_rdy_n	0
/tf/trn_tdst_rdy_n	0
/tf/trn_tbuf_av	f
/tf/trn_terrfwd_n	1
/tf/trn_tsrc_dsc_n	1
/tf/trn_tdst_dsc_n	1
Now	10000100 ns
Cursor 1	5112435.189 ns

Figure 3-24 Interrupts ON and OFF

left: ON; right: OFF.

Figure 3-25 shows the source throttling at the Rx side.

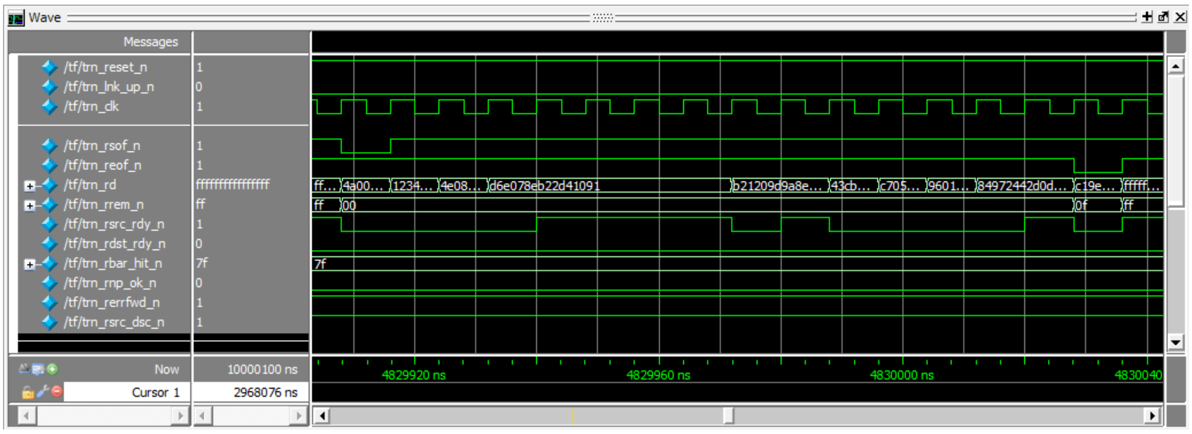


Figure 3-25 Rx source throttling

Figure 3-26 shows the destination throttling at the Tx side.

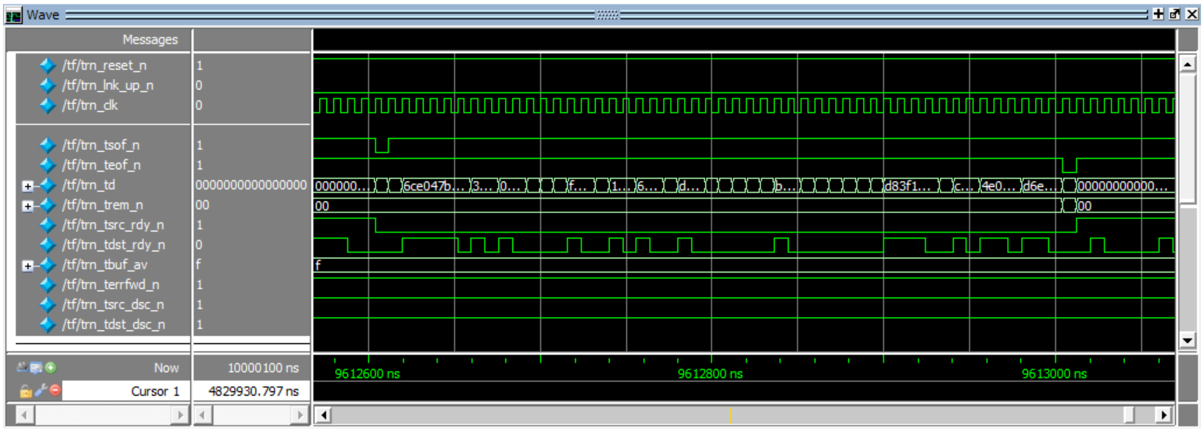
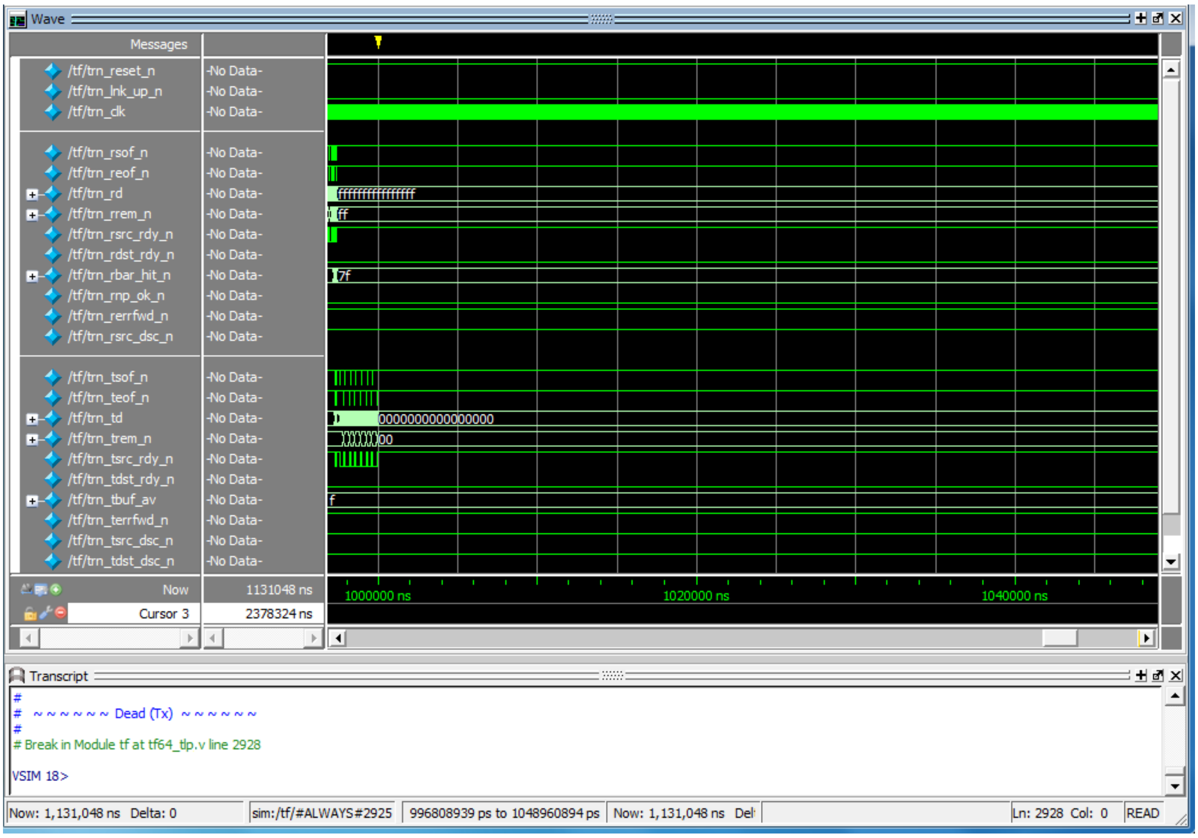


Figure 3-26 Tx destination throttling

Figure 3-27 is an example with the Tx death status report. Similar death report also includes the loop death, where the current simulation loop halts but the Tx is not "dead" because the Rx keeps sending MRd TLPs for status polling and the Tx outputs CplD TLPs.

Figure 3-28 is a successful verification specified to run 100 ms. It ran about 3 hours on our simulation PC. Actually a 30ms-run can already cover all discovered logic bugs.



```
/tf/trn_reset_n      -No Data-
/tf/trn_lnk_up_n     -No Data-
/tf/trn_clk          -No Data-

/tf/trn_rsof_n       -No Data-
/tf/trn_reof_n       -No Data-
/tf/trn_rd           -No Data-
/tf/trn_rrem_n       -No Data-
/tf/trn_rsrc_rdy_n   -No Data-
/tf/trn_rdst_rdy_n   -No Data-
/tf/trn_rbar_hit_n   -No Data-
/tf/trn_rnp_ok_n     -No Data-
/tf/trn_rerrfwd_n    -No Data-
/tf/trn_rsrc_dsc_n   -No Data-

/tf/trn_tsof_n       -No Data-
/tf/trn_teof_n       -No Data-
/tf/trn_td           -No Data-
/tf/trn_trem_n       -No Data-
/tf/trn_tsrc_rdy_n   -No Data-
/tf/trn_tdst_rdy_n   -No Data-
/tf/trn_tbuf_av      -No Data-
/tf/trn_terrfwd_n    -No Data-
/tf/trn_tsrc_dsc_n   -No Data-
/tf/trn_tdst_dsc_n   -No Data-

Now        1131048 ns
Cursor 3   2378324 ns
```

```
#
#  ~ ~ ~ ~ ~ ~  Dead (Tx)  ~ ~ ~ ~ ~ ~
#
# Break in Module tf at tf64_tlp.v line 2928

VSIM 18>
```

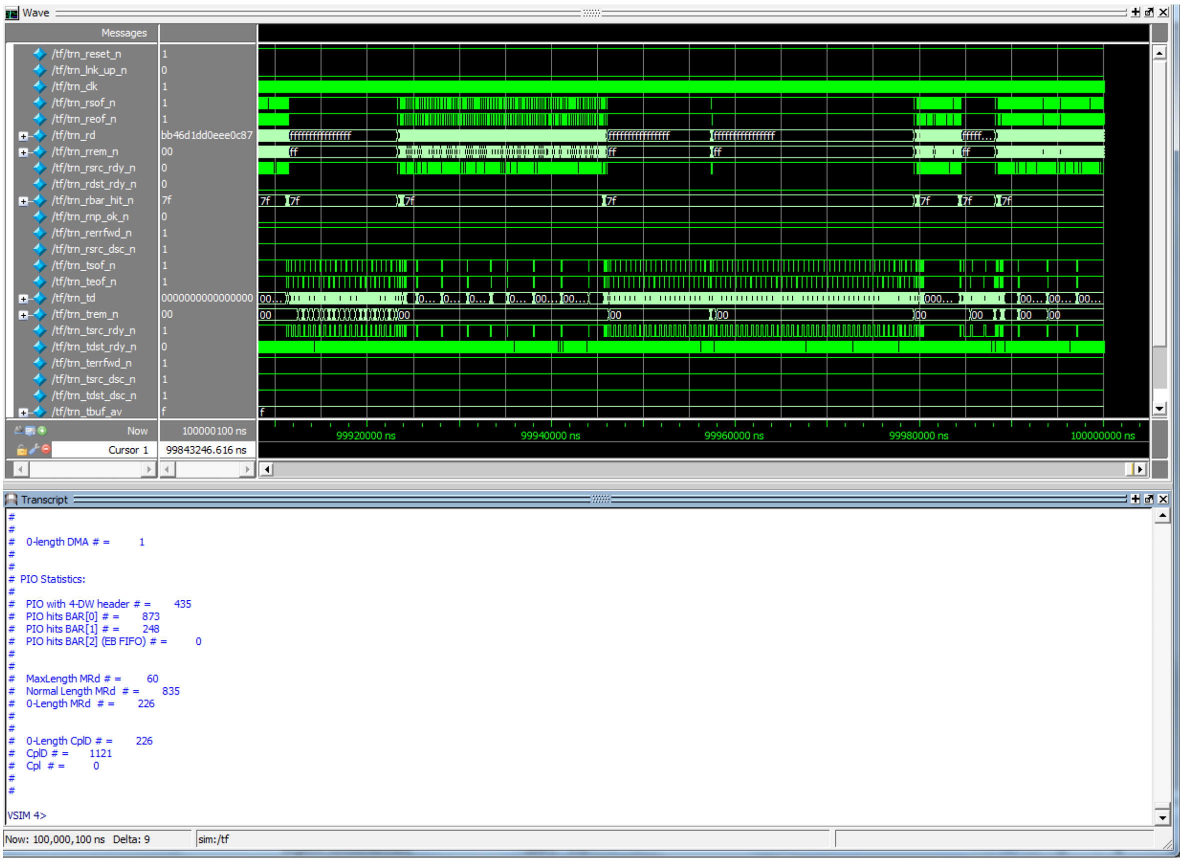
```
Now: 1,131,048 ns  Delta: 0 | sim:/tf/#ALWAYS#2925 | 996808939 ps to 1048960894 ps | Now: 1,131,048 ns  Del | Ln: 2928  Col: 0 | READ
```

Figure 3-27   Simulation (Tx) death state report

For the current version, the minimization upon the simulation time is not too much considered because the randomized test patterns can reach the satisfaction fast. Our experience shows that if the simulation time reaches 30 ms without error, it will also be able to sustain 100 ms. And if 100 ms simulation time is over and no errors happen, the logic can be released to configuration because no error has been found after 100 ms simulation time. The longest simulation time we have done to the PCI Express DMA simulation is 300 ms.

Later in this chapter, an approach to determine the end of simulation, transition-level verification (TLV), will be discussed. TLV is a preliminary research and just begins to be applied to our verification, so most of our verification work was done with estimated termination. The idea of TLV is that, after the expected coverage is reached, the simulation can be stopped and verification is supposed to be successful. TLV is a kind of partial full-coverage. Simulators, such as ModelSim, provide coverage interface.

The machine to run the simulation is an Intel 4-core, 3.0 GHz, with 4GB DDR2 memory. The simulator we use is ModelSim SE 6 from Mentor Graphic Inc.



Messages	
/tf/trn_reset_n	1
/tf/trn_lnk_up_n	0
/tf/trn_clk	1
/tf/trn_rsof_n	1
/tf/trn_reof_n	1
/tf/trn_rd	bb46d1dd0eee0c87
/tf/trn_rrem_n	00
/tf/trn_rsrc_rdy_n	0
/tf/trn_rdst_rdy_n	0
/tf/trn_rbar_hit_n	7f
/tf/trn_rnp_ok_n	0
/tf/trn_rerrfwd_n	1
/tf/trn_rsrc_dsc_n	1
/tf/trn_tsof_n	1
/tf/trn_teof_n	1
/tf/trn_td	0000000000000000
/tf/trn_trem_n	00
/tf/trn_tsrc_rdy_n	1
/tf/trn_tdst_rdy_n	0
/tf/trn_terrfwd_n	1
/tf/trn_tsrc_dsc_n	1
/tf/trn_tdst_dsc_n	1
/tf/trn_tbuf_av	f
Now	100000100 ns
Cursor 1	99843246.616 ns

```
#
#
#   0-length DMA # =        1
#
#
# PIO Statistics:
#
#   PIO with 4-DW header # =       435
#   PIO hits BAR[0] # =       873
#   PIO hits BAR[1] # =       248
#   PIO hits BAR[2] (EB FIFO) # =        0
#
#
#   MaxLength MRd # =       60
#   Normal Length MRd  # =       835
#   0-Length MRd  # =       226
#
#
#   0-Length CplD # =       226
#   CplD # =      1121
#   Cpl  # =        0
#
#

VSIM 4>
```

Now: 100,000,100 ns   Delta: 9   sim:/tf

Figure 3-28   Simulation successfully finished

The limitation for the verification up to now is that two or more loops can not be crossed, because that introduces complexity for the data verification. In terms of simulation time, it is hard to predict. We can try to simulate as long as possible, but the quantitative decision is more or less a matter of experience. And for this reason we are studying transition-level verification approach.

3.6 Transition-level verification

Simulation grows as the logic does. Such a simulation was not completed at one time. It was improved step by step from the practical test feedback. When test errors occurred in bench test and it was sure that the problem was in the logic design, the corresponding situation was captured and integrated to the simulation in form of a new stimulus situation. In this way, the number of logic bugs is getting less and less until a very reliable DMA logic is released. A direct benefit of this verification is that it allows big version update, for example, registers appending or removing, new transactions supporting, logic simplification due to FPGA area or timing constraints, etc. The verification guarantees the fairly big logic update does not injure the original design robustness. Only upon this base can a logic design have a healthy development. For example, in ML605 DMA project, the DMA engine used to experience large change to aggregate multiple requests into one to improve the performance. The simulation found some hidden bugs and the release version logic met the resource constraints and introduced no new bugs.

More attention is being paid to verification research in digital logic design. It is a trend towards automation. [67]

Formal verification is powerful to prove a design's correctness, but it normally requires large effort to develop the proper method. Due to its high cost, it is mostly limited in the life-critical fields or security systems. [68]

A thorough verification involves all gates toggling on and off. However, for FPGA design, such verification is sometimes impractical and unrealistic. A complex design usually consists of thousands of FFs. Of course a FF can still be broken down to gates. But we would like to stay on FFs and try to test their toggling. Furthermore, for state machines, every state is a combination of FFs. So a simplification of FF traversing can also be expressed as state traversing. Vice versa, if we focus on the state machines in a design, FFs/registers can be seen as 2-state machines, entry at initialization value, as in figure 3-29.

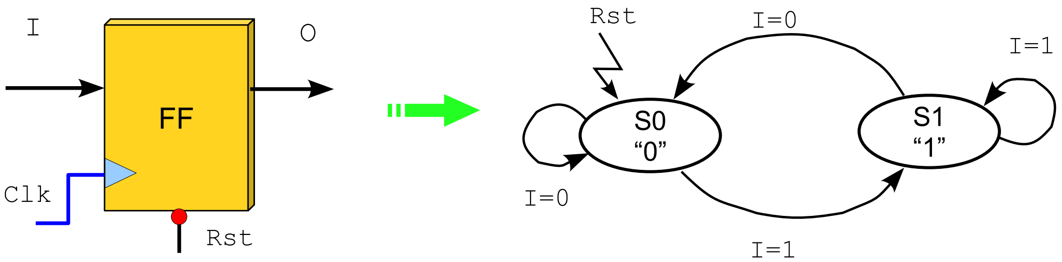


Figure 3-29 FF seen as a 2-state machine (output omitted)

Thereby we have a weaker definition for verification, namely behavioural traverse. A behavioural traverse in digital logic verification is the method in which all transitions (arcs) in a state machine are walked by the stimulus.

We take the bold assumption that the logic is guaranteed to run along the trajectory of its behavioural description (state machines, counters, …), without unexpected electrical diverges, for example meta-stability. The non-logic failure or unexpected divergence is covered by analogue emulators and therefore, out of the scope of this thesis.

Combinatorial logic is not emphasized because the pipeline behaviour is the focus for most large designs. This is also a simplification for high-speed synchronous digital designs. Of course, if the combinatorial logic switches the transition of a state machine, it is sure to be traversed.

Transition-level verification is to maximize the coverage by means of state-traverse of FSMs. FSM is widely used in today HDL designs and the synthesizers are getting more powerful in synthesizing the state machines. For state-coding as example, the optimization can even choose among all possible coding styles (one-hot, binary, …)

In the previous part of this chapter, we have built up the infinite verification environment. However, we have neither quantitative estimation nor logical decision to end the verification as it reaches the goal. With the transition-level traverse method, it is easier to decide when to stop the verification. In this way, the coverage is well controlled, whereas it reaches the proposed level and does not introduce too much redundancy in simulation time.

Figure 3-30 is a simplified model of state machine with 3 states. Regardless of its state coding, the behavioural description is as the graph.

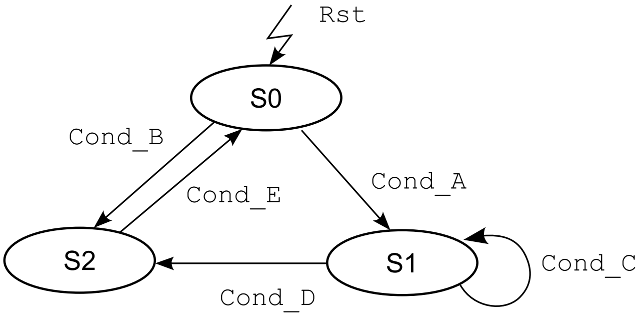


Figure 3-30 An example of general finite state machine for TLV

Some states of a state machine cannot be reached because of design defect and most synthesis tools can report such states, for example, Xilinx XST. If we have simulation route S0 → S1 → S2 → S0, every state is covered. However, this route does not guarantee the transitions S1 → S1 and S0 → S2 can work properly, that is, the expected outputs. Here comes the concept of n-traverse of state machine. All n-step combinations of possible state transitions is the set of n-traverse. For instance, 2-traverse of the state machine in figure 3-30 includes,

S0 → S1,

S0 → S2,

S1 → S1,

S1 → S2 and

S2 → S0.

If all these 5 transitions are traversed, we can assume that all entries and all exits of all states are covered and the logic is behaviourally reliable.

And we have seen, state-traverse, or 1-traverse, is insufficient for state machine verification. Similarly we can have n-traverse, which can be theoretically divided into 2-traverses.

FSM CAD tools are also booming. For example, Frank Lemke has contributed the FSMDesigner to the open source community, which provides versatile functions and great flexibility in developing

FSMs. [69]

To execute the transition traverse verification, internal state signals should be feasible for the test bench. First, the verification fills out a transition matrix, as the graph algorithms. For example, the state machine in figure 3-30 has the transition matrix in figure 3-31.

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 0 \end{array}$$

Figure 3-31 A transition matrix example

Then, the stimulus is exerted upon the tested unit and the same matrix is filled. After the two matrices are exactly equal, the verification is finished. And if no error is found, the logic is reliable. Some note here is, when some state cannot be reached, the corresponding elements in the filled matrix should be initially set to 1, which means, such transitions do not have to be traversed.

TLV needs a sound support from simulation as earlier discussed in this chapter. The randomized stimulus is able to drive the logic to every state and over every transition. If not, the stimulus should be improved.

If after a long time of simulation/verification, the traverse is still not done, there might be some states or registers unreachable. When this happens, investigation into the specific units is needed. If some states or FFs are surely unreachable, their markers in the traversing matrix should be set 1.

3.7 DMA engine variations

We have tried four versions of development boards for the Active Buffer project. Figures 3-32 to 3-35 list the relevant boards. On all the boards, we have implemented the DMA engine running on the transaction layer of PCI Express. The PCI Express core from Xilinx takes care of transactions on the physical layer and the data link layer. In Virtex4 FPGA, the PCI Express core is a “soft” one. From Virtex5 on, the PCI Express core is hardware primitive. But for the transaction layer user, the implementation is no difference.

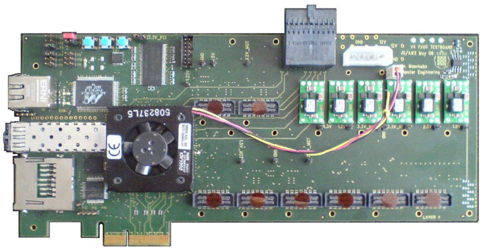


Figure 3-32 ABB1 (ZITI)

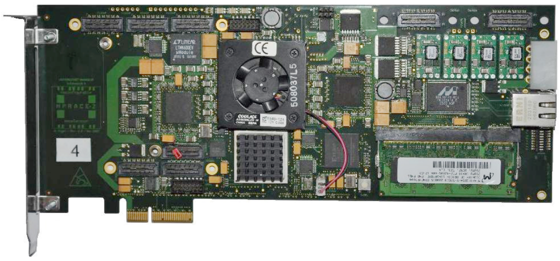


Figure 3-33 MPRACE2 (ZITI)

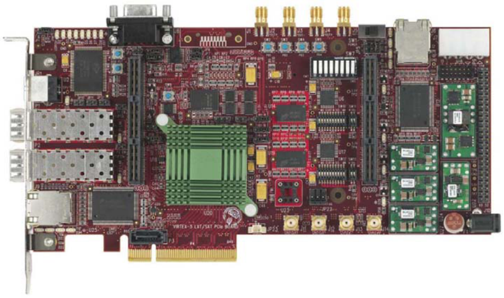


Figure 3-34 ABB2 (AVNET)

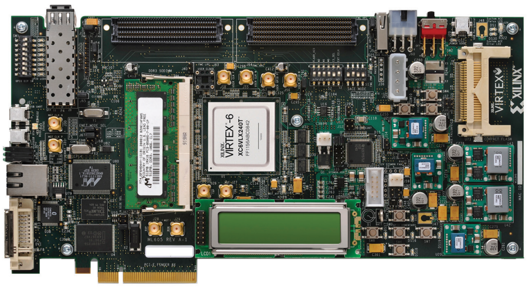


Figure 3-35 ML605 (Xilinx)

Xilinx provides general DMA engine reference designs for their FPGAs. These designs do not fit well in our applications, so we develop DMA engine of our own.

3.7.1 ABB1 – Virtex4 FX20/FX60

The first one has a Xilinx Virtex4 FPGA as the center device, with soft PCI Express core, 4 lanes. We used to use this board with Virtex4 FX20 FPGA, but due to tight resource limitation, we upgraded it to FX60, which is pin-out compatible to FX20. However, the resource limitation is still not yet totally away. That is why the PCI Express core in this version is 32-bit wide in data bus, not 64-bit. On this board, 32 MB DDR SDRAM is available, and possibly more memory chips can be plugged into the mezzanine slots. With this board, we have experience of DPR (**D**ynamic **P**artial **R**econfiguration) practice. [70]

The resource consumption can be found in *Appendix **A.1***.

3.7.2 MPRACE2

MPRACE2 is another Virtex4 board, it is equipped with two Virtex4 FPGAs, one is FX20 for PCI Express interface, focused on data transfer; the other is FX60, larger and focused on scientific computing acceleration, e.g. SPH (**S**moothed **P**article **H**ydrodynamics). The FX20 is labelled BRIDGE and the FX60 is labelled MAIN. DDR2 modules are attached to the MAIN as the work storage. Between these two FPGAs, 10Gbps Aurora links are implemented, because the data rate is high. As illustrated in figure 3-36. [71] [72] [73]

Ideally the communication between should be like with one FPGA. However, the PCI Express module should be always alive and the reconfiguration of the MAIN is through the BRIDGE. In the design phase, DPR was not so powerful.

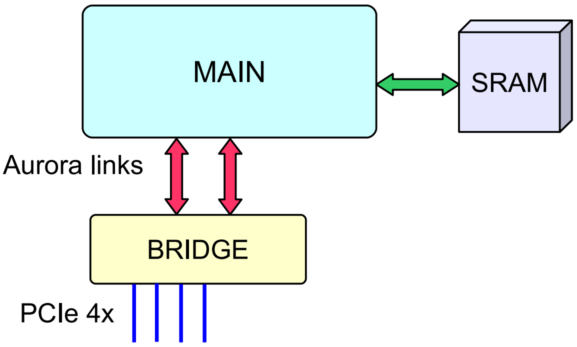


Figure 3-36 MPRACE2 structure

The DMA engine is ported to the smaller FPGA on the MPRACE2 board, namely BRIDGE FPGA. BRIDGE FPGA manages all the PCI Express interface transactions, including PIO and DMA. The host machine has access to the MAIN via the BRIDGE, over the Aurora links (5Gbps × 2). We have 4 BARs for this system,

BAR[0] is the system registers in BRIDGE, 64KB;

BAR[1] is BRAM space in BRIDGE, 1MB.

BAR[2] and BAR[3] is resident on the larger FPGA, namely MAIN.

BAR[2] is register space in MAIN, 64KB;

BAR[3] is memory space in MAIN, 1MB.

For real operation implementation, BAR[1] is set to dummy to save resource in the BRIDGE. The

DMA write transfer data from the host to the MAIN over the BRIDGE, and DMA read is the opposite direction.

DMA write moves data to the Aurora link buffer, then into the MAIN RAM space. After SPH computation, the host software gets the status by polling, and then issues a GET-DATA command to the MAIN. The MAIN puts the requested data into the Aurora link, and the data arrive in a link buffer in the BRIDGE. At the same time, the host initiates a DMA read, which will wait until the BRIGE link buffer has had the amount of data for the first packet, and then move them into the host memory, and waits for the next packet of data to be ready.

MPRACE2 uses separate FPGAs to have the flexible reconfigurability. However, the high-speed links between the two FPGAs caused some problem in debugging and development. The link does not always work in the proper way when flow control happens. The external high-speed link has more difficulty into stable operation than the internal one. As a conclusion, the external high-speed link takes more effort in debugging. So, for a preparation of the later discussion, the DPR technology is going to be a better solution for quite similar purposes.

The resource consumption can be found in *Appendix* ***A.2***.

3.7.3 ABB2 – AVNET Virtex5 PCIE development board

The third development board is from AVNET Inc., namely AES-XLX-V5LXT-PCIE110-G, with a Virtex5 LX110T FPGA, 8-lane hardware PCI Express core, 64-bit transaction layer interface, larger DDR2 SDRAM memory. [74]

We use it in 4-lane configuration. DMA read performance is about 543 MB/s and DMA write performance is about 790 MB/s. DPR is also practised on this board. It is widely used in recent CBM DAQ beam tests. A generalized HDL design of this project has been committed to OpenCores.org. [75]

The resource consumption can be found in *Appendix* ***A.4***.

3.7.4 Viroquant application – ML605

We also transport our DMA engine to a Virtex 6 FPGA, XC6VLX240T on ML605 board, sketched in figure 3-37. On that board, the memory interface is upgraded to DDR3 SDRAM modules. Other than in the MPRACE2 project, the data for DMA read is not commanded by the host PIO command. Instead, the data are requested by the read DMA engine itself. [76]

Because the DDR3 memory controller has larger overhead to process a read request, the `MAX_PAYLOAD_SIZE` parameter is too small for it. So for this project, we adapt the DMA read module to aggregate the multiple smaller read requests into one for every DMA descriptor.

Accordingly, the DMA read DONE status is redesigned to give a correct END signal because the data fetching mechanism in this project is different with the others.

The resource consumption can be found in *Appendix* ***A.5***.

3.8 PCI driver

We do the performance tests under Linux kernel version 2.6 and develop the PCI driver and the MPRACE library for CBM DAQ. The driver and the library are developed by Dr. Guillermo Marcus Martinez. He also contributes most of the test programs. [77]

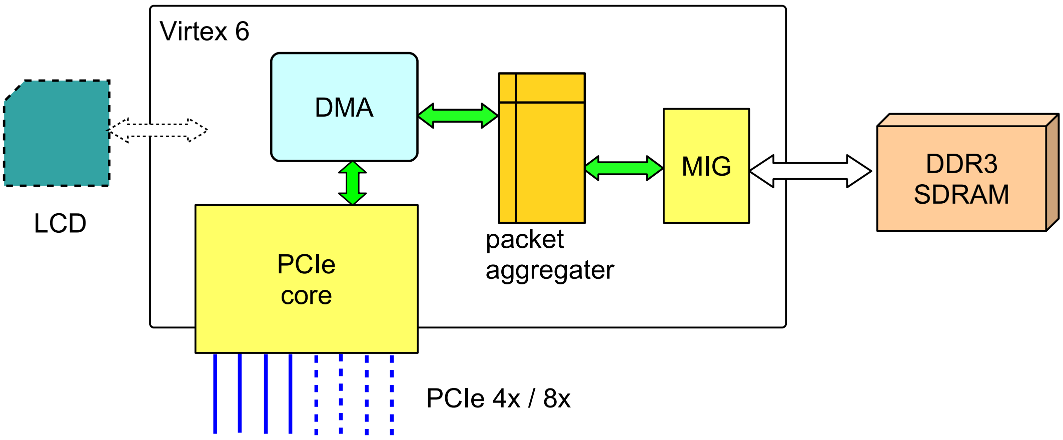


Figure 3-37 Block diagram of ML605

The test programs support KERNEL memory space and USER memory space. In KERNEL test mode, DMA transaction is done in one descriptor; in USER mode, because of larger and segmented memory space, DMA is usually done in multiple descriptors. USER test mode usually involves larger memory space and it will be more acceptable for our final DAQ application. In terms of performance, these two modes have little difference.

During the test initialization, a buffer space is allocated in the host memory before the DMA test is started. This buffer can be filled by PIO or DMA write and then read out with the same two options. The acquired data are compared with the data filled in, for data integrity checking. After the DMA correctness is verified, the performance tests are executed, without data comparison, to get rid of unnecessary overhead during performance measurement. The software tests use the same driver and library as of the applications in real CBM DAQ system.

Method to aggregate multiple PIO write (MWr) into one TLP from the software layer is not yet founded. So the initial DMA descriptor has to be issued in separately 8 MWr PIO commands and this makes up overhead for a DMA start.

3.9 Summary

In terms of the Active Buffer, we have built up a prototype DAQ environment for CBM experiment. This data buffering and transporting system is implemented in form of SG DMA channels over PCI

Express. The DMA channels are robust interface between the network and the host. Such design has general meaning to DAQ systems for high-energy physics, especially for those with high data flow density. We have an SG DMA controller in stable operation and in the future, it can be upgraded to the master mode to free the host load further. To upgrade it to master mode needs effort both in hardware design and software cooperation but it is worthwhile.

A verification environment is built up to avoid the discovered bugs being repeated during the logic modification. It is very important to have such verification method for complex logic design in data path applications. This verification environment has grown to the comparable size as the implemented design in terms of HDL file size.

FPGA vendors, such as Xilinx, are supposed to provide PCI Express Core regularly for their commercial and marketing consideration. PCI Express Gen2 is also in sight now. The transaction layer interface has minimal difference since the one in Virtex5. So the DMA engine design should be no difficulty to reuse for future applications.

In the next chapter, DMA performance test results will be presented.

Chapter 4 DMA performance tests

4.1 Introduction

In this chapter, we present typical performance test results with the DMA engine of chapter 3. As we know, 4-lane PCI Express Gen1 has 10 GT/s transfer rate, with 8b/10b encoding. Therefore on the transaction layer, the peak data rate is 1 GB/s, which can never be acquired by the application due to the TLP packetizing overhead. Ideally, the DMA performance reaches its maximum when the TLPs are issued back-to-back and every TLP contains maximum count of data payload that is allowed by the `MAX_PAYLOAD_SIZE`, both for RX and TX of PCI Express. If the DMA initialization overhead is ignored, we have the peak performance values with respect to the `MAX_PAYLOAD_SIZE` parameter in table 4-1 for 4-lane Gen1 PCI Express configuration. Note that in table 4-1, 32-bit or 64-bit addressing does not make difference, but the bus width does.

Table 4-1 Theoretical peak DMA performance for 4-lane PCI Express Gen1

`MAX_PAYLOAD_SIZE` (byte) M	Peak performance with 32-bit bus (MB/s) P_{32}	Peak performance with 64-bit bus (MB/s) P_{64}
128	914.28	888.89
256	955.22	941.18
512	977.10	969.70
1024	988.42	984.62
2048	994.17	992.25
4096	997.08	996.11

In table 4-1, the results are calculated with

$$P_{32} = M/(M+12) \quad \text{[GB/s]}$$

and

$$P_{64} = M/(M+16) \quad \text{[GB/s].}$$

There are 12 bytes (3-DW header) overhead per TLP with 32-bit bus width and 16 bytes (3-DW header plus 1-DW tail) overhead with 64-bit bus width. Actually the back-to-back transactions can not always be guaranteed either by the source or by the destination.

We have a few computers to make such performance tests and we have found out that the results are strongly dependent upon the chip-set types of the host machines. For example, the test chart in figure 4-1 was made with a machine of AMD chip-set, whose DMA read performance (484 MB/s) is lower than that on another test machine, a machine of Intel chip-set, where the DMA read can reach over 640 MB/s, not included here. In the CBM DAQ, the DMA read performance is more important because that channel transports DAQ data to the host. And for the DMA write channel, it will be used to transfer CTL (**c**o**nt**ro**l**) messages or DLM (**D**eterministic **L**atency **M**essages), which do not need high bandwidth. So we emphasize on the improvement of the DMA read performance.

The DMA transactions are for the present time initiated by the Active Buffer board. The data transfer in DMA write (downstream) involves a 2-step process,

1) card issues read requests to the host,

2) the host sends the requested payload in completions (CplD) to the card.

Data transfer in DMA read (upstream) is done in one step after the DMA is started. The board just sends payloaded TLP (MWr) from the event buffer to the host. In our system, the DMA write has higher performance than that of DMA read, as figure 4-1 and figure 4-2 show.

The test result values are averaged out of long stable operations, e.g. 50 loops every value. The performance values fluctuate strongly, and a smaller number of loops gives relatively too much variance. Infinite test loops are also made to verify the stability and robustness.

The performance tests are mainly targeted to the BRAM (BAR[1]) as the memory type. Because the Virtex5 (AVNET) board is the major CBM DAQ appliance, FIFO performance test is made.

In performance tests, all data content is filled with zeros.

To find out the quantitative principles behind the performance measurement, linear regression is made to the DMA execution time in the performance tests in this chapter. The performance is correlated with DMA size. The DMA initialization costs quasi fixed amount of CPU time, so that larger sizes of DMA has higher performance. A simplified modelling for the DMA time, can be

$$t = t_0 + s/p_0$$

where

t: total time of the DMA,

t_0: fixed initialization time to start the DMA, including buffer allocation, descriptor set-up and issuing, etc.

s: DMA size, and

p_0: the steady speed for the DMA payload transfer, equivalent to the peak bandwidth if the DMA size were infinite. Note that p_0 is neither P_{32} nor P_{64} in table 4-1.

In the following performance tests, t is calculated as

$$t = s/p$$

where p denotes the measured DMA performance.

In this way, the DMA time is a linear function of the DMA size. In following performance charts, the measured DMA parameters (t_0 and p_0) are deduced with linear regression, as well as the standard error σ of performance regression and the correlation coefficient r.

4.2 Virtex4 performance test

Figure 4-1 shows the BRAM DMA performance tests with ABB-1. The test was taken about two years ago, so the DMA size range was narrow, from 4 kB to 512 kB.

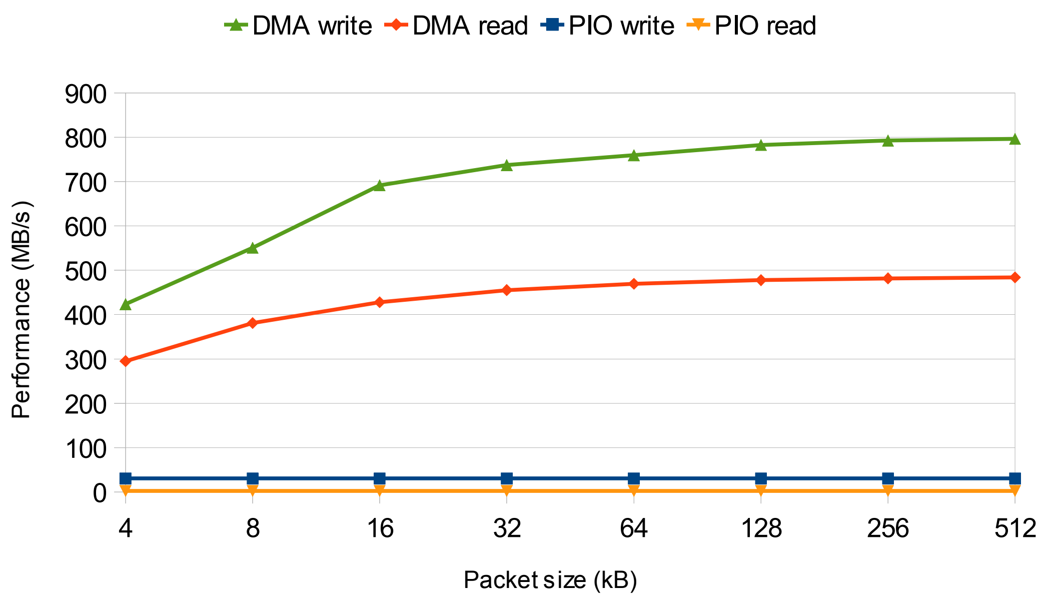


Figure 4-1 BRAM DMA performance test with Virtex4 FPGA (ABB-1)

4-lane PCI Express Gen1. AMD chip-set

DMA read: t_0 = 4.050639 μs, p_0 = 801.2957 MB/s, r = 0.9999973, σ = 14.57986.

DMA write: t_0 = 4.767245 μs, p_0 = 485.8838 MB/s, r = 0.9999997, σ = 5.535199.

4.3 Virtex5 DMA performance test and analysis

Figure 4-2 shows the BRAM performance tests with the ABB-2 board. The performance is a little bit higher than that with the Virtex4 board because we have made improvement upon the DMA transaction pipelines between these two versions.

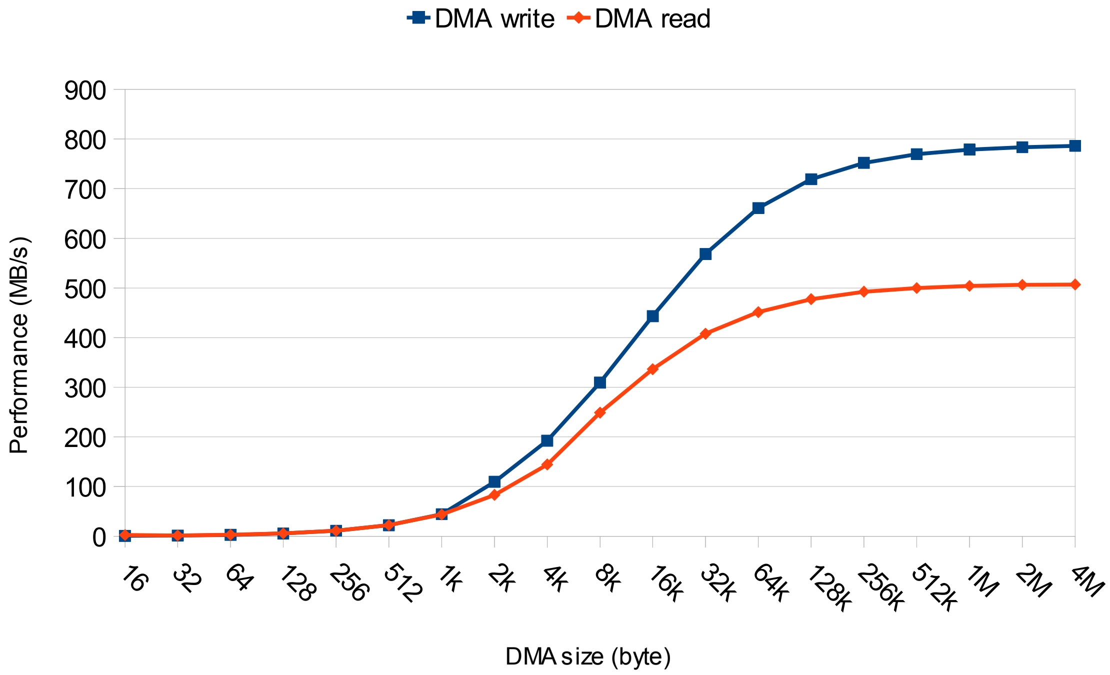


Figure 4-2 BRAM DMA performance test with Virtex5 FPGA (ABB-2)

4-lane PCI Express Gen1. Intel chip-set

DMA read: t_0 = 18.37311 μs, p_0 = 508.3438 MB/s, r = 0.9999983, σ = 6.241044.

DMA write: t_0 = 18.97955 μs, p_0 = 788.9628 MB/s, r = 0.9999974, σ = 14.36144.

Besides the BRAM (BAR[1]) DMA performance test on the Virtex5 board, the FIFO (BAR[2]) DMA performance is also tested. For FIFO DMA write, an internal loop-back is made with the FIFO, which disables the network incoming data. On the output port of the FIFO, the read enable signal is always asserted so that the FIFO never becomes almost full, available for maximum write speed. For FIFO DMA read test, the input port has the always asserted write enable signal and the FIFO never becomes empty, which allows a maximum read speed.

The FIFO DMA performance test on Virtex5 board is shown in figure 4-3.

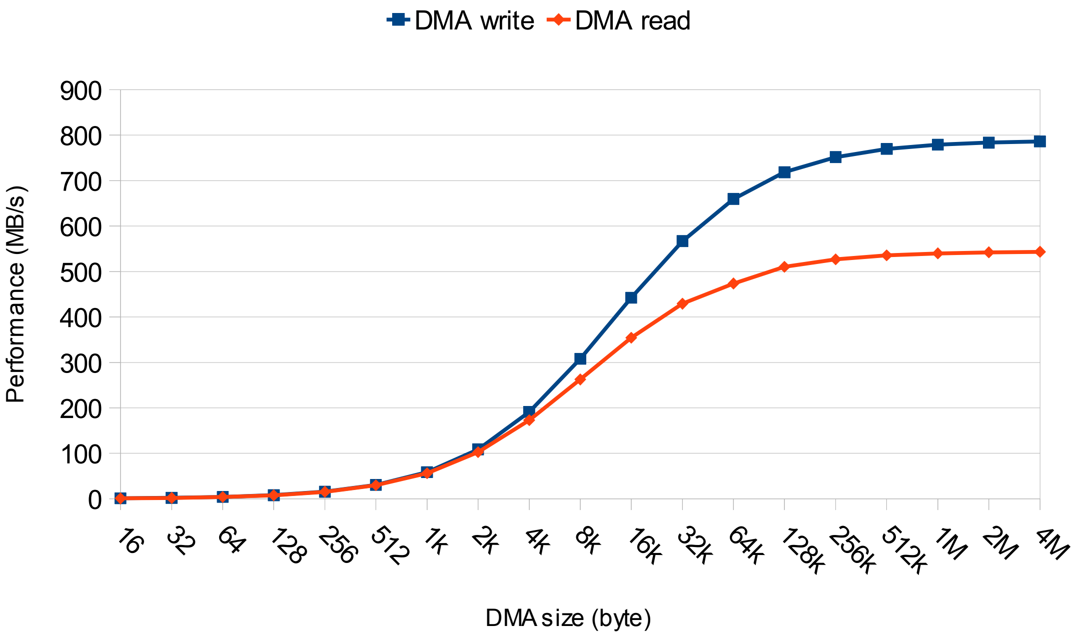


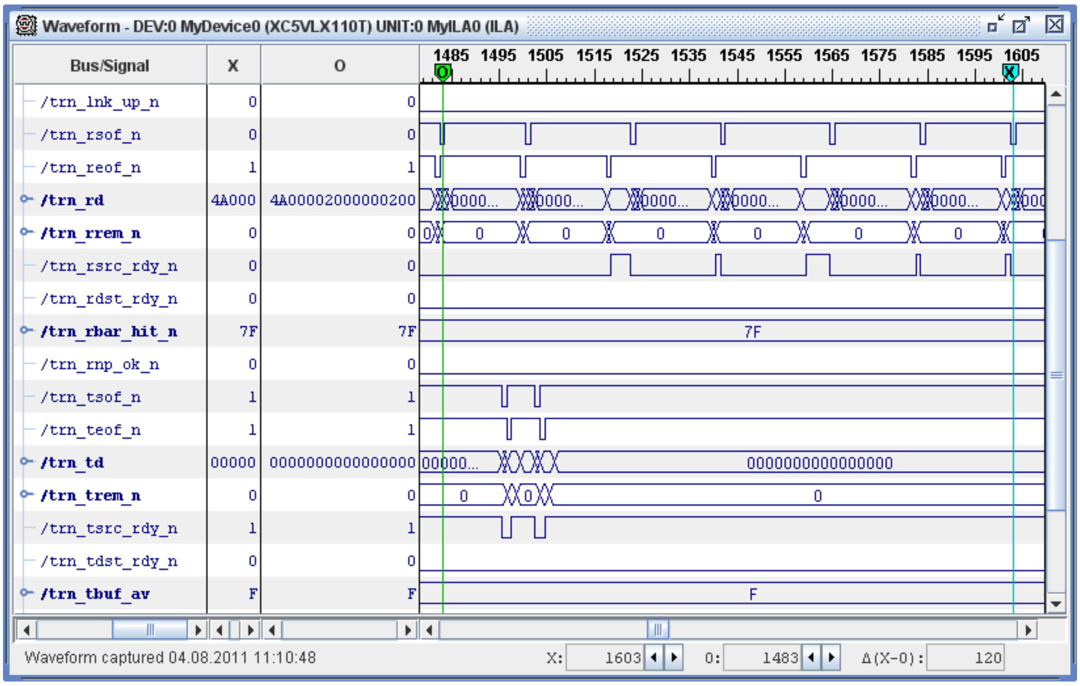
Figure 4-3 FIFO DMA performance test with Virtex5 FPGA (ABB-2)

4-lane PCI Express Gen1. Intel chip-set

DMA read: t_0 = 16.32730 μs, p_0 = 544.4987 MB/s, r = 0.9999999, σ = 1.469778.

DMA write: t_0 = 16.16106 μs, p_0 = 788.3250 MB/s, r = 0.9999999, σ = 0.4544296.

In terms of DMA write, the FIFO performance is almost the same as the BRAM performance test, because the bottleneck lies in the host and the DMA write performance depends upon how fast the root can provide the data. Ideally, if the root could sustain the back-to-back transaction (CplD), the peak DMA write performance with `MAX_PAYLOAD_SIZE = 0x80 = 128` bytes would be 888.89 MB/s, as table 4-1. However, as we can see (from the deasserted `trn_rsrc_rdy_n`) in figure 4-4 and figure 4-5, back-to-back transaction is not always possible from the root, which prevents the DMA running at the peak performance (888.89 MB/s). Comparing the DMA write performance test results (above 780 MB/s) with the calculated peak performance values (both 800 MB/s) in these two figures suggest that there is not much possibility in improving the DMA write performance. Coincidently figure 4-4 and figure 4-5 have the same performance estimation value. Actually they can have variation with each other.

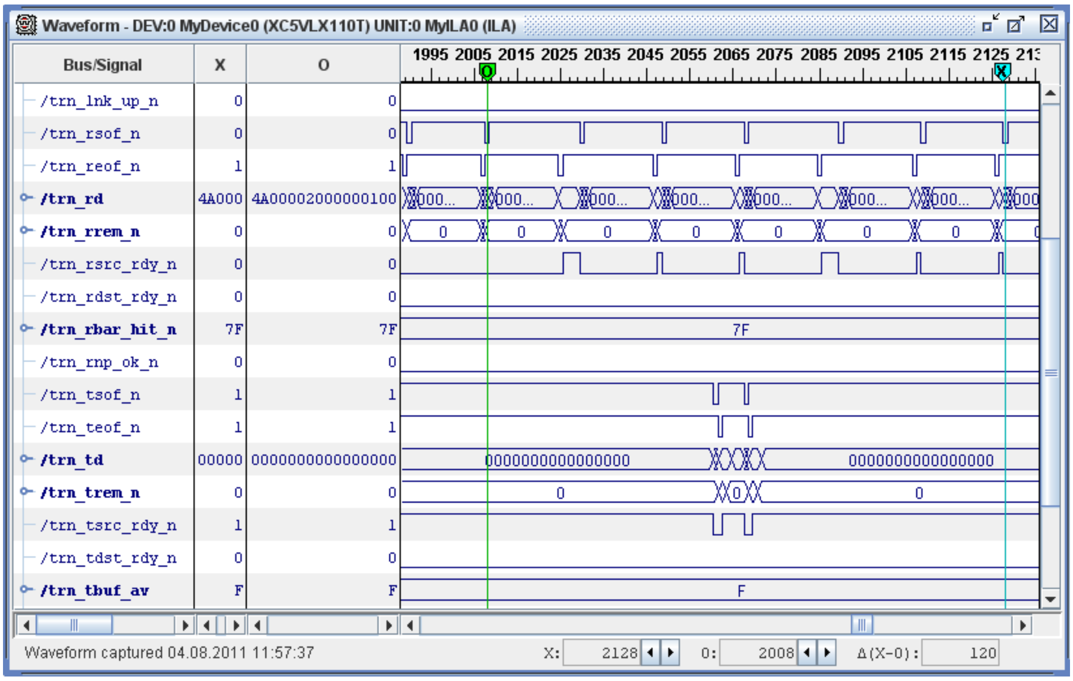


Bus/Signal	X	O
/trn_lnk_up_n	0	0
/trn_rsof_n	0	0
/trn_reof_n	1	1
/trn_rd	4A000	4A00002000000200
/trn_rrem_n	0	0
/trn_rsrc_rdy_n	0	0
/trn_rdst_rdy_n	0	0
/trn_rbar_hit_n	7F	7F
/trn_rnp_ok_n	0	0
/trn_tsof_n	1	1
/trn_teof_n	1	1
/trn_td	00000	0000000000000000
/trn_trem_n	0	0
/trn_tsrc_rdy_n	1	1
/trn_tdst_rdy_n	0	0
/trn_tbuf_av	F	F

Waveform captured 04.08.2011 11:10:48 X: 1603 O: 1483 Δ(X-O): 120

Figure 4-4 ChipScope snapshot of the BRAM DMA write test with Virtex5 FPGA (ABB-2)

Calculated peak performance $p_{0wrBRAM}$ *= 1 GB/s × 16 × 6 / 120 = 800 MB/s*.



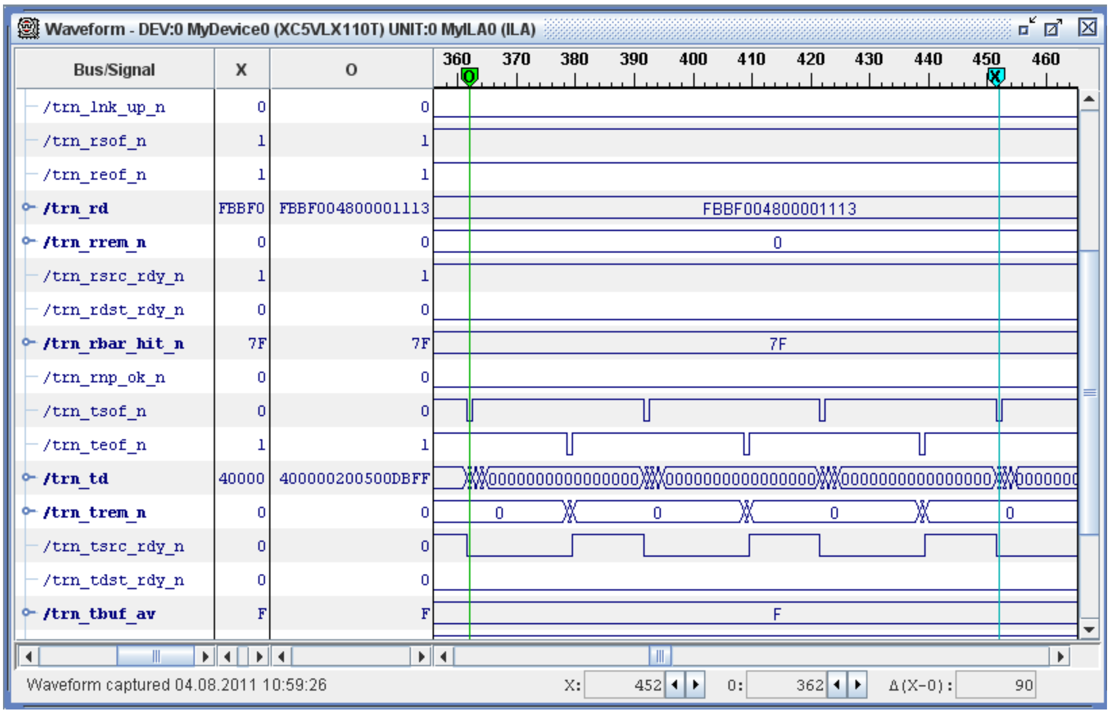
Bus/Signal	X	O
/trn_lnk_up_n	0	0
/trn_rsof_n	0	0
/trn_reof_n	1	1
/trn_rd	4A000	4A00002000000100
/trn_rrem_n	0	0
/trn_rsrc_rdy_n	0	0
/trn_rdst_rdy_n	0	0
/trn_rbar_hit_n	7F	7F
/trn_rnp_ok_n	0	0
/trn_tsof_n	1	1
/trn_teof_n	1	1
/trn_td	00000	0000000000000000
/trn_trem_n	0	0
/trn_tsrc_rdy_n	1	1
/trn_tdst_rdy_n	0	0
/trn_tbuf_av	F	F

Waveform captured 04.08.2011 11:57:37 X: 2128 O: 2008 Δ(X-O): 120

Figure 4-5 ChipScope snapshot of the FIFO DMA write test with Virtex5 FPGA (ABB-2)

Calculated peak performance $p_{0wrFIFO}$ *= 1 GB/s × 16 × 6 / 120 = 800 MB/s*.

Now let us take a look at the DMA read. Figure 4-6 is the BRAM DMA read snapshot and figure 4-7 is the FIFO DMA read. We can see that the throttling comes only from the peripheral (`trn_tsrc_rdy_n`) and this obviously degrades the DMA read performance. The gap cycle number between two payload TLPs is larger than that in the DMA write case because the peripheral cannot provide the data fast enough for back-to-back transactions. Another observation is the FIFO performance is higher than the BRAM and this is due to that the BRAM data path has one more stage buffer than the FIFO path. If we can in the future improve the DMA read pipelines, the same performance as the DMA write, 800 MB/s, may be anticipated. This means, there is much possibility for a better DMA read performance.


Waveform - DEV:0 MyDevice0 (XC5VLX110T) UNIT:0 MyILA0 (ILA)

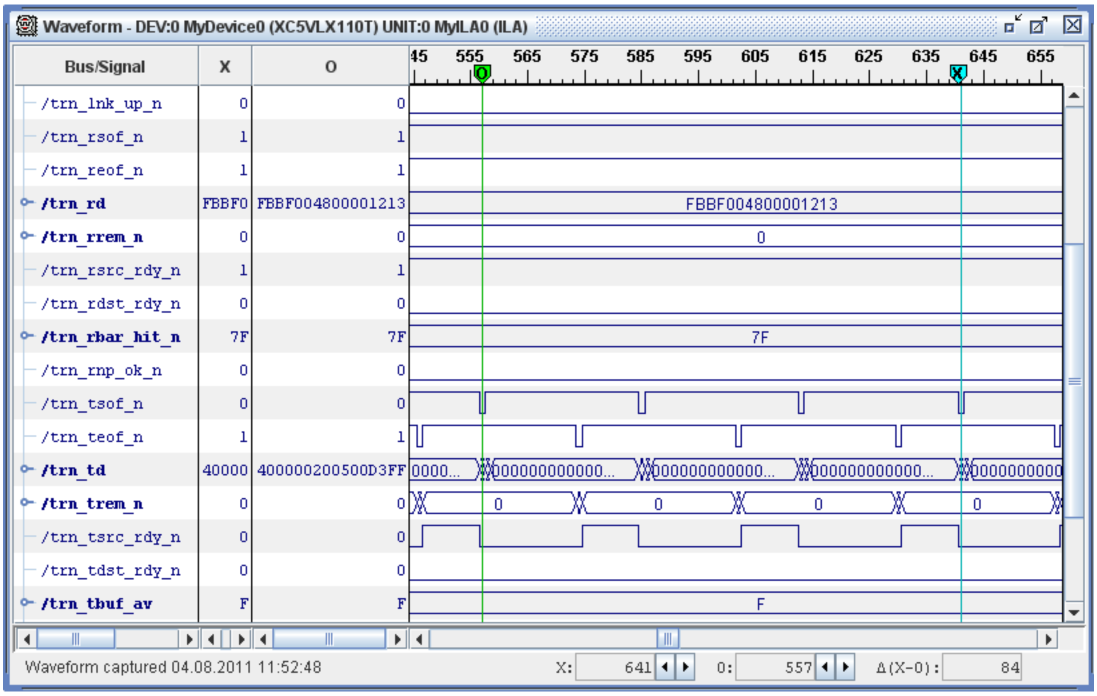
Bus/Signal	X	O
/trn_lnk_up_n	0	0
/trn_rsof_n	1	1
/trn_reof_n	1	1
/trn_rd	FBBF0	FBBF004800001113
/trn_rrem_n	0	0
/trn_rsrc_rdy_n	1	1
/trn_rdst_rdy_n	0	0
/trn_rbar_hit_n	7F	7F
/trn_rnp_ok_n	0	0
/trn_tsof_n	0	0
/trn_teof_n	1	1
/trn_td	40000	400000200500DBFF
/trn_trem_n	0	0
/trn_tsrc_rdy_n	0	0
/trn_tdst_rdy_n	0	0
/trn_tbuf_av	F	F

Waveform captured 04.08.2011 10:59:26

X: 452   O: 362   Δ(X-O): 90

Figure 4-6 ChipScope snapshot of the BRAM DMA read test with Virtex5 FPGA (ABB-2)

Calculated peak performance p_{0rdBRAM} *= 1 GB/s × 16 × 3 / 90 = 533 MB/s.*



Waveform - DEV:0 MyDevice0 (XC5VLX110T) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
/trn_lnk_up_n	0	0
/trn_rsof_n	1	1
/trn_reof_n	1	1
/trn_rd	FBBF0	FBBF004800001213
/trn_rrem_n	0	0
/trn_rsrc_rdy_n	1	1
/trn_rdst_rdy_n	0	0
/trn_rbar_hit_n	7F	7F
/trn_rnp_ok_n	0	0
/trn_tsof_n	0	0
/trn_teof_n	1	1
/trn_td	40000	400000200500D3FF
/trn_trem_n	0	0
/trn_tsrc_rdy_n	0	0
/trn_tdst_rdy_n	0	0
/trn_tbuf_av	F	F

Waveform captured 04.08.2011 11:52:48

X: 641 O: 557 Δ(X-O): 84

Figure 4-7 ChipScope snapshot of the FIFO DMA read test with Virtex5 FPGA (ABB-2)

Calculated peak performance $p_{0rdFIFO}$ = *1 GB/s × 16 × 3 / 84 = 571 MB/s*.

Such measurements are evidenced with the simulation environment in chapter 3. As figure 4-8 shows, the BRAM DMA read simulation counts the same value as the ChipScope snapshot in figure 4-6. Note that 720 ns / 8 ns = 90. And in figure 4-9, the FIFO DMA read simulation agrees with the ChipScope snapshot in figure 4-7. Note that 672 ns / 8 ns = 84.

Therefore, the simulation environment introduced in chapter 3 can be a logic benchmark in the DMA read performance improvement actions. What we measure in this simulation is what we get in the real performance tests.

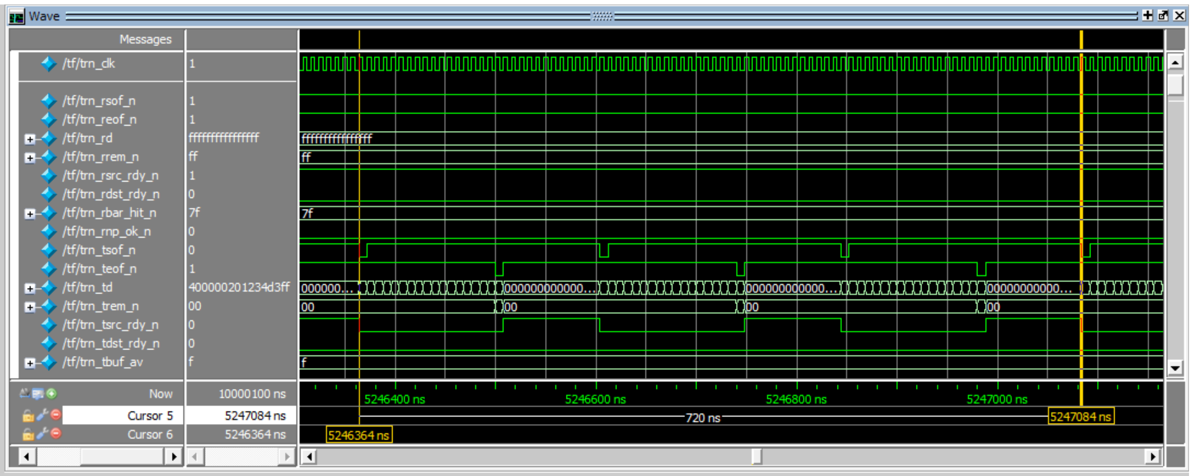


Figure 4-8 ModelSim simulation wave form reproduces the BRAM DMA read test

Calculated peak performance $p_{0rdBRAM\_sim} = 32 \times 4B \times 3\,/\,720\ ns = 533\ MB/s$.

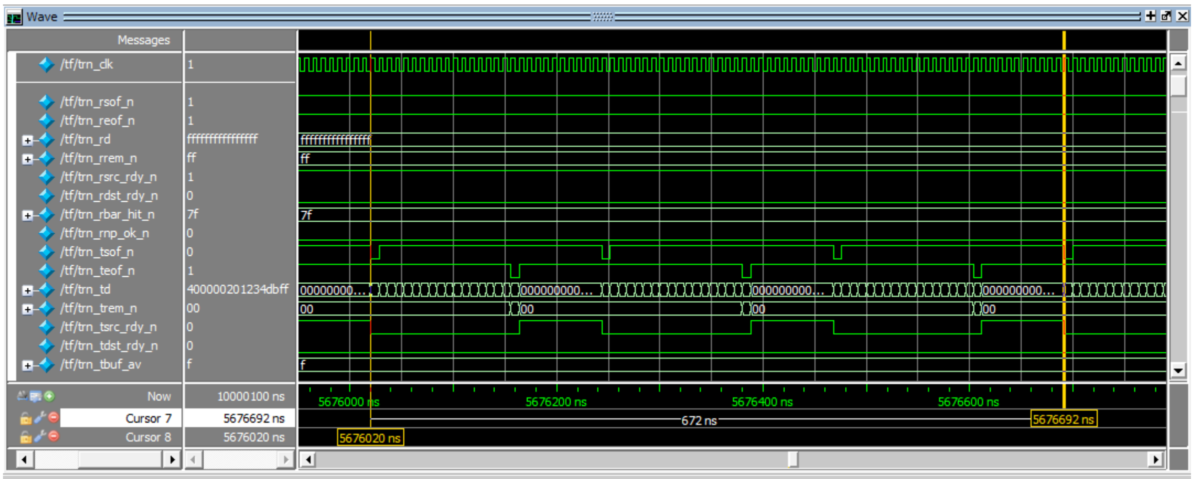


Figure 4-9 ModelSim simulation wave form reproduces the FIFO DMA read test

Calculated peak performance $p_{0rdFIFO\_sim} = 32 \times 4B \times 3\,/\,672\ ns = 571\ MB/s$.

4.4 Virtex6 DMA performance test

Figure 4-10 shows the performance tests with the ML605 board. The logic design and the PCI-Express core have quite the same behaviour with the AVNET Virtex5 board. So the performance values are also almost the same. DMA performance tests over FIFO are not done with ML605 board and they should have the same results as the Virtex5 performance.

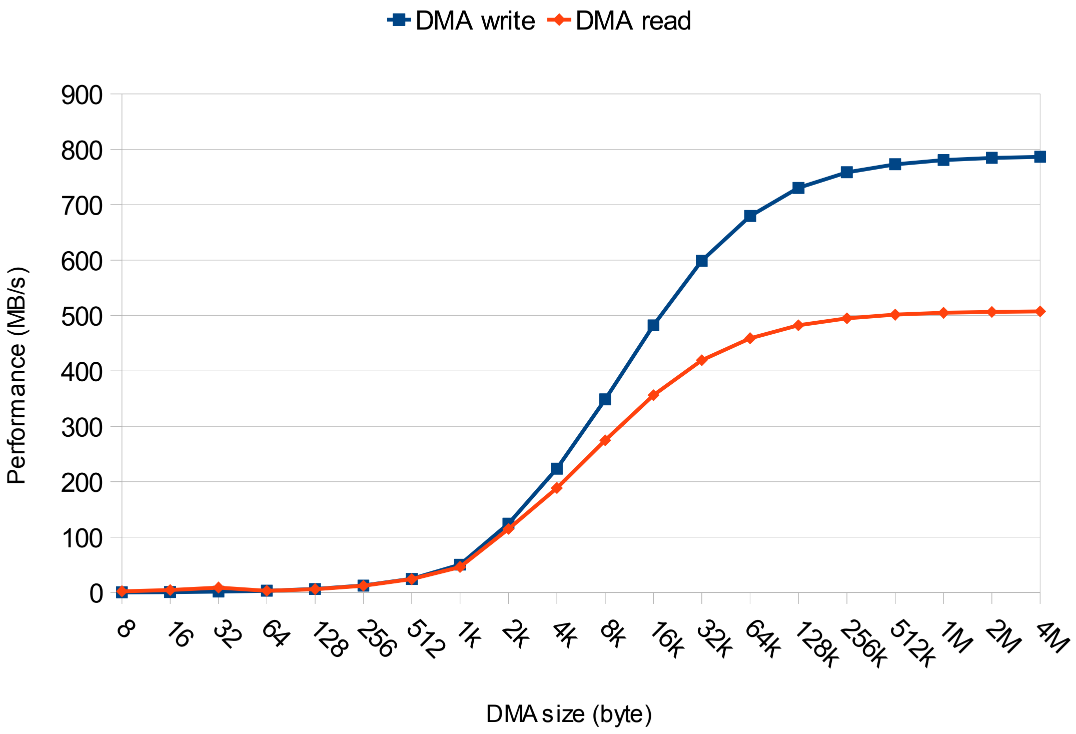


Figure 4-10 BRAM DMA performance test with Virtex6 FPGA (ML605)

4-lane PCI Express Gen1. Intel chip-set

DMA read: t_0 = 13.95407 μs, p_0 = 508.1852 MB/s, r = 0.9999965, σ = 5.217210.

DMA write: t_0 = 16.40552 μs, p_0 = 789.0997 MB/s, r = 0.9999970, σ = 18.42229.

Taking the ML605 Virtex6 test in figure 4-10 as example, the fitting curve for DMA read performance with the regressed parameters is compared with the measured curve in figure 4-11. Majority of measured sample points are well approximated. The disagreed regression points suggest the improvement on this model. For the DMA write performance regression in figure 4-10, the comparison is shown in figure 4-12.

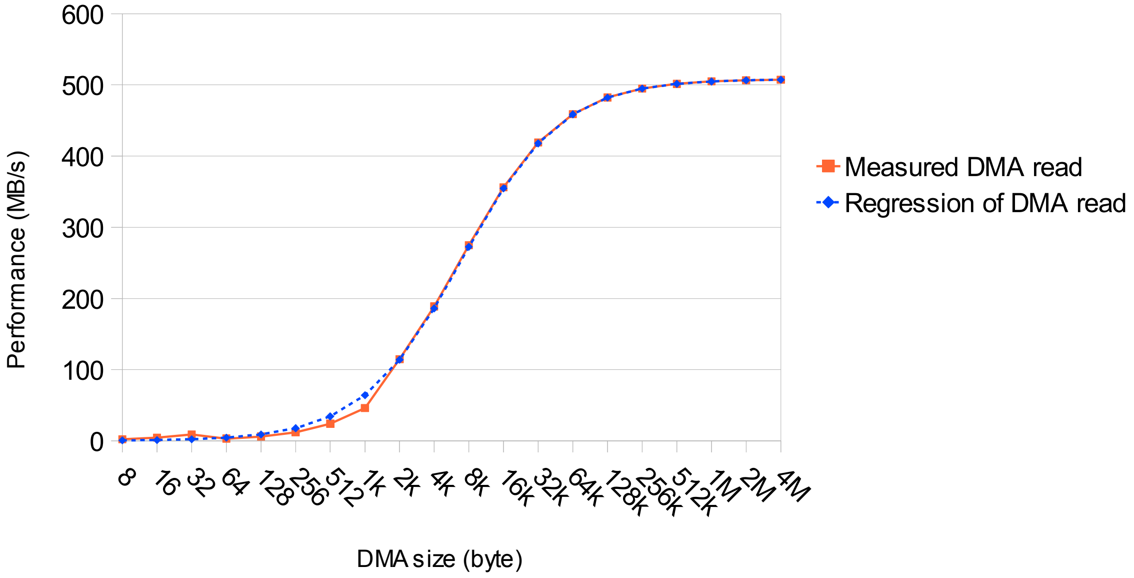


Figure 4-11 Virtex6 BRAM DMA read performance regression ($r = 0.9999965$, $\sigma = 5.217210$)

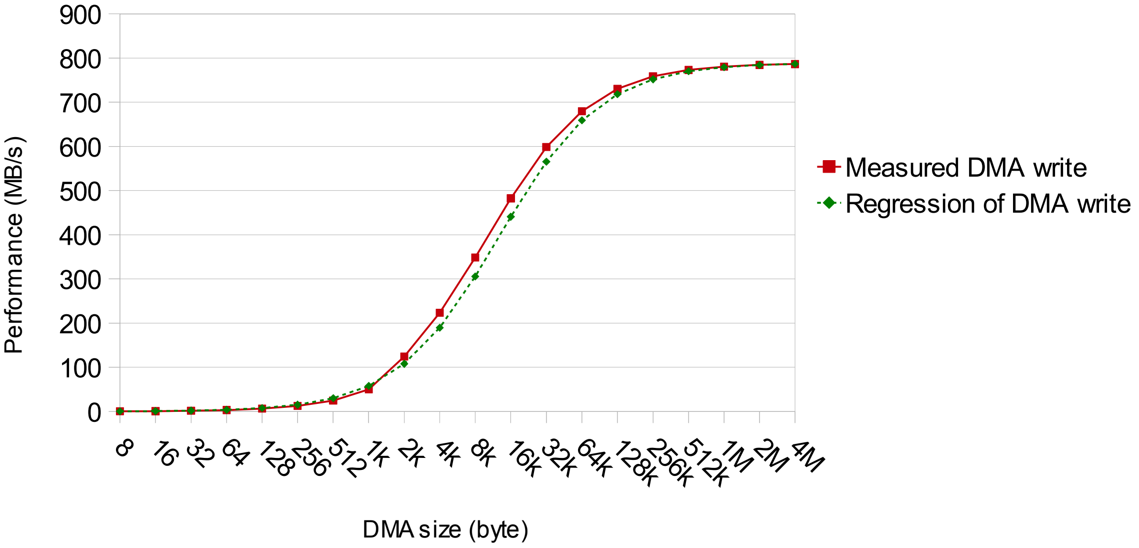


Figure 4-12 Virtex6 BRAM DMA write performance regression ($r = 0.9999970$, $\sigma = 18.42229$)

4.5 Summary

Thanks the PCI Express properties, the DMA engine presents favourable performance results. The performance tests also show that the system is endurably reliable. This is a solid foundation for the future development in CBM DAQ system.

A simplified performance model is built up with linear regression approach. The correlation coefficients are satisfactory. With this model, primary analysis can be done to identifying the system DMA supporting characteristics. It is an attempt to model the system behaviour. We have seen in the performance tests, for a given card, the t_0 parameters for DMA read and DMA write are quite close but the p_0 parameters differ much. t_0 plays an important role for DMA of smaller sizes. For DMA of larger sizes, i.e. over 512 kB, the performance approximates p_0 very well. According to this observation, the performance improvement of small-size DMA depends more upon the software, and the performance improvement of large-size DMA is much word in hardware or logic design.

The DMA read performance is critical for our application, so it should be further improved according to the possibility we just discussed. The DMA write will be used for messages and control to the front-end and its improvement possibility is not so big as the DMA read.

Chapter 5 Dynamic partial reconfiguration

5.1 Motivation

GSI is an industrial entity and it is not allowed to freely use the PCI Express core in Virtex4 licensed to universities. To avoid the license issue and to cut the unnecessary cost for GSI, we have the solution from DPR (**D**ynamic **P**artial **R**econfiguration) that wraps the PCI Express core and provides the final user with the modules (mainly the DMA engine) they really care. Actually the FPGA development is mainly in ZITI, Heidelberg University and GSI colleagues do not have to touch the PCI Express core. Since Virtex5, the PCI Express core is made as a hardware macro and the license problem is no longer there. However, we have benefited from the DPR and therefore continue in exploring its potential.

As we have experienced in ABB1 project, the FX20 finally seemed a little too small for the design. If ChipScope core is needed for debugging, the slice resource got extremely tight. Fortunately we have FX60 version for the upgrade, otherwise we would have to compromise between the abundance of functionalities and the chip area. This has been admitted to be a universal problem for programmable device applications. The fix-sized chip can never be too large for the project. It happens quite often that the proposed device turns out to be under-sized as the project goes into later phases. Unexpected consumption of logic resource should be allowed even for the most strictly planned project. However, if the free space is preserved enough, let us say, 50%, the waste percentage will also go high and the cost evaluation will disappoint the project leader.

Another note from our PCI Express DMA debugging is related to the system reboot overhead. Due to the link training requirement, a system reboot is necessary after a new bit-stream file is loaded in FPGA to get the new test start. This takes sometimes even minutes for a new FPGA configuration, although a very tiny change in the DMA engine is made. There are certain approaches with IOCTL under Linux to rescan the PCI Express bus for an updated enumerating without reboot. However, this software method causes break to the PCI Express core. For communication-focus applications or for situations where the PCI Express configuration space should be preserved, reconfiguring the PCI Express core and introducing operation gap is not a good idea. In this sense, the DPR solution has more general meaning. In the event building debug, there will be multiple version of event building modules with slight difference to test on-line. The DPR solution will provide valuable convenience and save configuration-reboot time in trying different event building firmwares.

Actually, inside the FPGA design, very often is, not all the logic modules work all the time. Some of them take shifts in the time sequence, which gave the programmable logic users many years ago the hope to overlay multiple functions into a smaller wafer area by means of shifting in and out logic modules. For example, most initialization logic works only for the first milliseconds or even nanoseconds and then just sleeps there until next power-up or reboot; at the other hand, the normal logic takes over the charge only after initialization. In some simplex data paths, the transmitting and receiving do not have temporal overlap during the operation.

Space technology, such as space stations, satellites, is an extreme case for DPR applications, where the target system is too far away and is supposed to run a long time service without direct human access. Power supply is budgeted, and hence, the programmable device resource is always one of the top considerations, along with the cosmos radio and failure recovery. [78]

Figure 5-1 illustrates a typical application of DPR in resource-limited environment. An FPGA is designed to inspect the target black-box system, which issues a series of digital signals and checks the outputs of the inspected system so that the internal property can be identified or be verified. Since the action speed to issue the stimulus sequence and to read the response signals is expected high, such application can not be done with a general processor. To snoop the target black-box, DPR is here a suitable solution. Furthermore, if the FPGA is not big enough to hold all the inspection modules in limited size, DPR can be applied to save the space and overlap the multiple functions. The set of inspection series is prepared in multiple reconfigurable modules, and the switching agent loads them one by one dynamically as demanded, without discontinuing other parts of the FPGA. After a cycle of inspection, the target pattern should be able to be identified and then, the host decides the corresponding operations upon it. For those dynamic modules, the cost is a cycling controller, which can be implemented with an MCU or a CPLD.

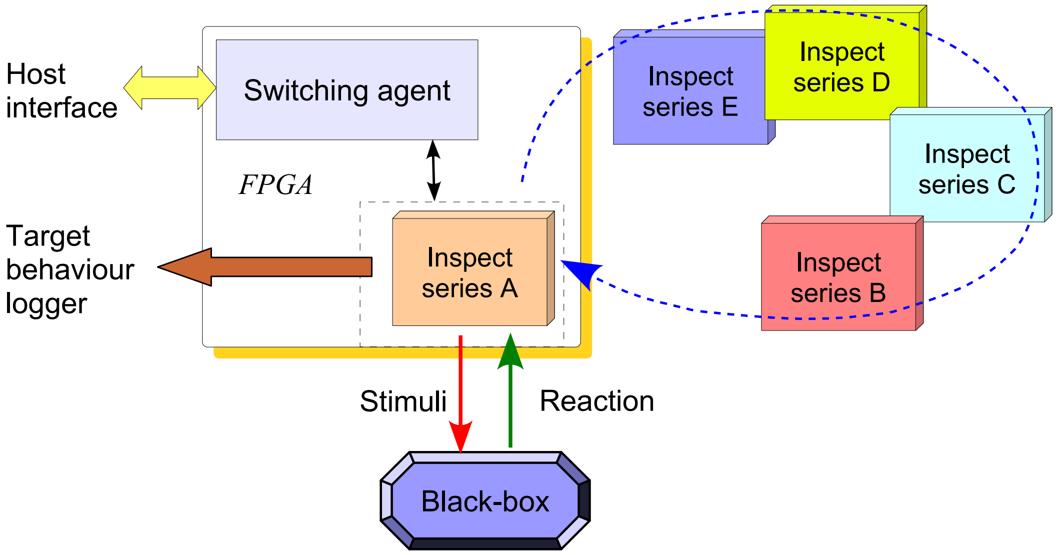


Figure 5-1 A time-slotted FPGA application: black-box snooper

Similar examples can be found in circumstances of loose-time and urgent-resource/-power application. The overhead of a general switching agent can be neglected when the overlap ratio is high enough. To carry out DPR, the common interface/pins should be kept in mind.

In our PCI Express DMA design, as discussed in previous chapters, the different maximum parameters have to be implemented in one module to prepare for all possible system configurations, and only one sixth surely takes effect. This is really a resource extravagance.

Hence, a natural attempt is to overlay the multiple time-scattered function modules into one area, and then to exchange them with specified order. In this way, the hardware modules can be uploaded and offloaded onto the hardware holder (FPGA, or other programmable device), just as the software application into the system space.

Of course the overhead to maintain the hardware flexibility and reconfiguration should be counted, both in extra resource consumption and the cost of development time, especially the increased complexity to use it.

In some other circumstances, the area or resource is not a big problem, but the time is. Still in our PCI Express DMA project, a newly generated and configured bit file with the PCI Express core takes effect only after the system is rebooted, because the link training and enumerating of PCI Express. There is a solution to spare this by software rescan of the PCI Express bus, but it does not always work. So every time we make minor change to the DMA logic, although nothing to do with the PCI Express core interface, a system reboot is needed after configuration. This is a time extravagance during the debugging phase. The partition capability of ISE Foundation can save time for the bit file generation, but the dynamic reconfiguration needs additional advanced support.

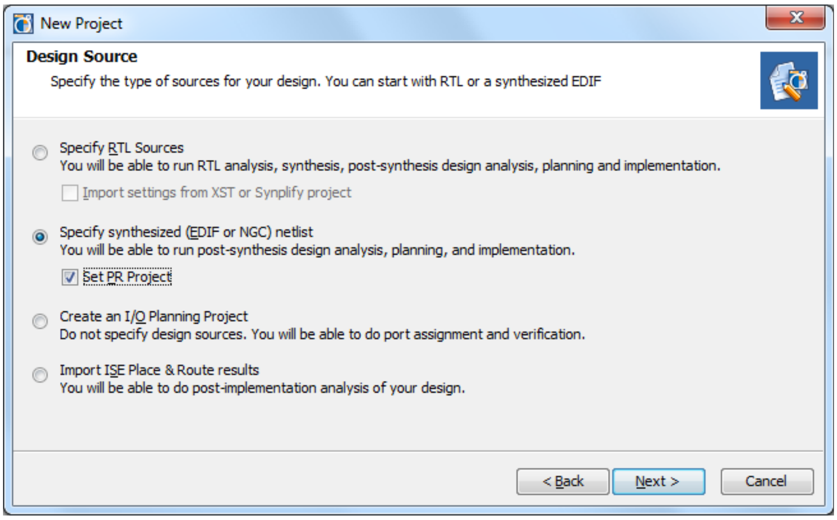
Similar situations take place for some applications with a permanent supervising module, which should never be disabled or shut off. Such modules cannot be placed into a device that is not dynamically reconfigurable. For instance, an FPGA design with real-time monitoring and logging, and some display modules cannot be easily reconfigured, if the monitoring module needs to stay conscious all the time. The conventional solution is to separate the modules into different chips and treated individually. But this tremendously increases the design work and is more error-prone than the single-chip solution. For example, MPRACE2 board has two FPGAs to have the reconfigurable capability with the price of complexity at the communication between those two FX series FPGAs.

Ideally, the modules are fit into one device and the static part stays untouched during the reconfiguration of the dynamic part(s). Such preferable solution is now realistic from DPR technology.

Initially, there were two options to do DPR with Xilinx FPGAs within the general user flow, namely module-based partial reconfiguration and difference-based partial reconfiguration. They both require the developer's great effort to settle all details of DPR, so that they can be called expertise technique. [79] [80] Then Xilinx provided EarlyAccess software suite for a neater DPR implementation. [81] Afterwards, PlanAhead was released, replacing the EarlyAccess software. PlanAhead is a powerful software suite not only for DPR, but also for general Xilinx FPGA implementations. [82] [83]

Partial reconfiguration is an extra special feature of PlanAhead and does not come with standard license. Figure 5-2 shows the validated feature in PlanAhead 12.4, with the check-box of "Set PR

Project”.



New Project

Design Source

Specify the type of sources for your design. You can start with RTL or a synthesized EDIF

○ Specify RTL Sources
You will be able to run RTL analysis, synthesis, post-synthesis design analysis, planning and implementation.

□ Import settings from XST or Synplify project

● Specify synthesized (EDIF or NGC) netlist
You will be able to run post-synthesis design analysis, planning, and implementation.

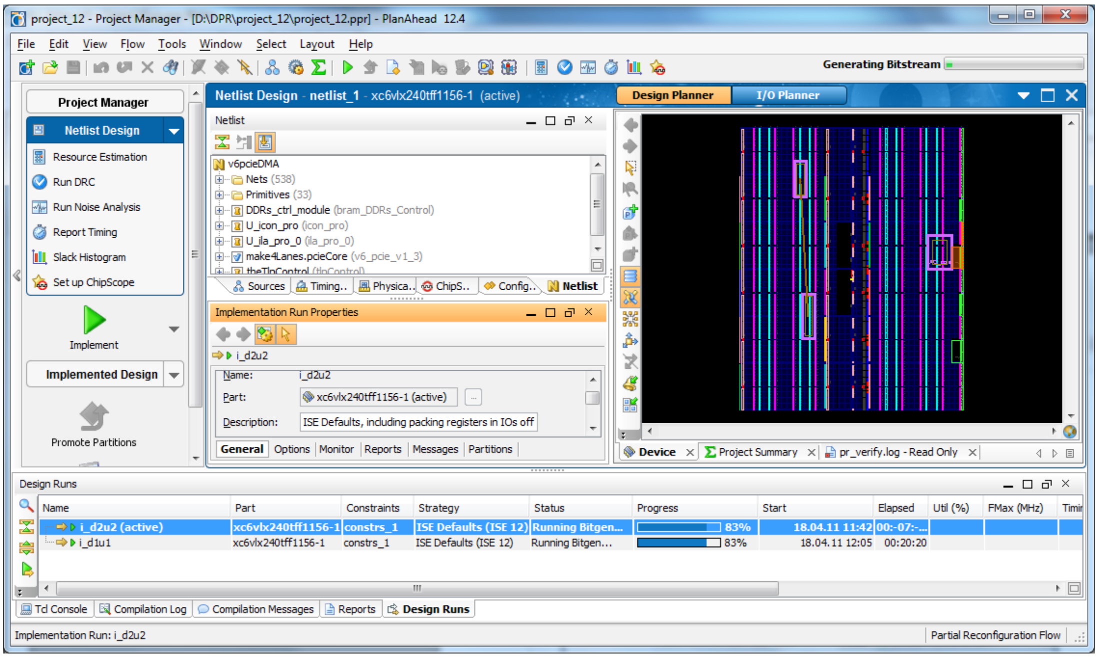
☑ Set PR Project

○ Create an I/O Planning Project
Do not specify design sources. You will be able to do port assignment and verification.

○ Import ISE Place & Route results
You will be able to do post-implementation analysis of your design.

Figure 5-2 PR feature enabled in project creation (PlanAhead 12.4)

Figure 5-3 is a project example GUI of PlanAhead 12.4.



project_12 - Project Manager - [D:\DPR\project_12\project_12.ppr] - PlanAhead 12.4

Netlist Design - netlist_1 - xc6vlx240tff1156-1 (active)

Netlist

- v6pcieDMA
  - Nets (538)
  - Primitives (33)
  - DDRs_ctrl_module (bram_DDRs_Control)
  - U_icon_pro (icon_pro)
  - U_ila_pro_0 (ila_pro_0)
  - make4Lanes.pcieCore (v6_pcie_v1_3)
  - theTlpControl (tlpControl)

Implementation Run Properties

i_d2u2

Name: i_d2u2

Part: xc6vlx240tff1156-1 (active)

Description: ISE Defaults, including packing registers in IOs off

Design Runs

Name	Part	Constraints	Strategy	Status	Progress	Start	Elapsed	Util (%)	FMax (MHz)	Timi
i_d2u2 (active)	xc6vlx240tff1156-1	constrs_1	ISE Defaults (ISE 12)	Running Bitgen...	83%	18.04.11 11:42	00:-07:-...			
i_d1u1	xc6vlx240tff1156-1	constrs_1	ISE Defaults (ISE 12)	Running Bitgen...	83%	18.04.11 12:05	00:20:20			

Implementation Run: i_d2u2

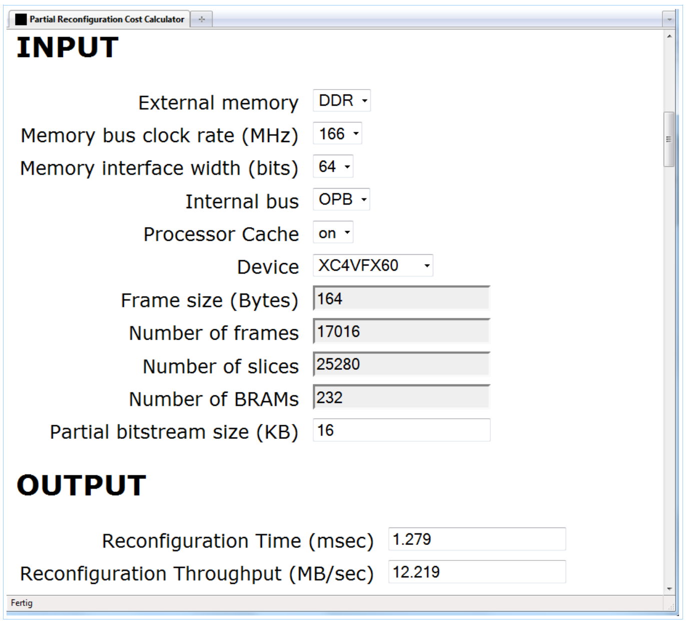
Partial Reconfiguration Flow

Figure 5-3 GUI of PlanAhead 12.4 in DPR implementation

There is also plenty of related work being undertaken both in theory and in practice. People are seeking exciting applications for DPR. [84] [85]

Norbert Abel in KIP of Heidelberg University has developed an object-oriented framework, combining DPR and HLS (**H**igh-**L**evel **S**ynthesis) technologies and keeping the flexibility of instantiation of DPR modules on the fly. [86]

Kyprianos D. Papadimitriou in Technical University of Crete provides his web page to calculate PR cost. An example calculation is shown in figure 5-4. [87]



INPUT

External memory: DDR
Memory bus clock rate (MHz): 166
Memory interface width (bits): 64
Internal bus: OPB
Processor Cache: on
Device: XC4VFX60
Frame size (Bytes): 164
Number of frames: 17016
Number of slices: 25280
Number of BRAMs: 232
Partial bitstream size (KB): 16

OUTPUT

Reconfiguration Time (msec): 1.279
Reconfiguration Throughput (MB/sec): 12.219

Source: http://users.isc.tuc.gr/~kpapadimitriou/prcc.html

Figure 5-4 A PRCC example

5.2 Overview

A dream for most computer scientists is to cancel the fast line between software and hardware. In this sense we have hardware-software co-design, embedded system, system-on-chip, and so on. They are out of the range of this dissertation. We want to focus on the approach from DPR (**D**ynamic **P**artial **R**econfiguration).

DPR is welcome in some application context, such as

(1) Area constraint and power minimizing,

(2) Presence of always conscious modules,

(3) "Mutex" working modules, which never work simultaneously in any time slice.

And due to the DPR tool evolution of Xilinx, PlanAhead, we have experienced two generations of project flows. With PlanAhead 9.2 we tried our DPR on Virtex4 FX60, and the bus-macros were needed. Then, with PlanAhead 12.4, we tried similar implementation on Virtex6 LX240T FPGA, when the bus-macros are obsolete and boundary decoupling is investigated.

5.2.1 Partition

Partition is a good process for modularized DPR projects. It keeps clear boundary for implementation modules, so that one partition can be reimplemented while other partitions' implementation remains untouched. A reconfigurable block should be associated with a partition and the partition should be assigned with an area constraint.

As discussed before, the PCI Express core is put into the static part, as well as the register space. DMA engines are taking 2 dynamic modules.

Partition concept is illustrated in figure 5-5.

With definition of partitions, the FPGA is divided into several parts and each part can be updated without affecting the healthy operations of other modules. It seems like there are multiple separate FPGAs available. Considering MPRACE2 project, this pseudo-multiple-device framework is an option.

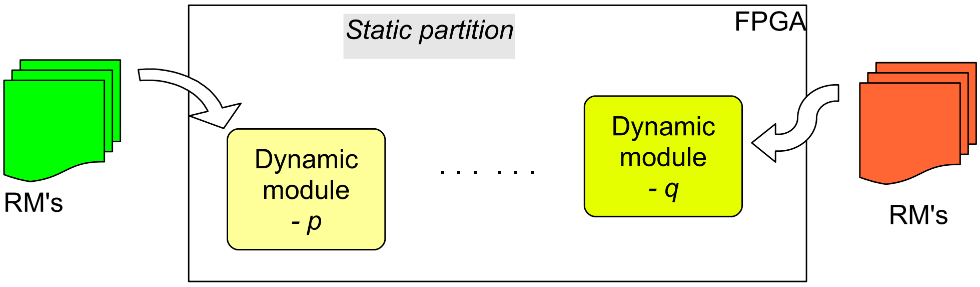


Figure 5-5 DPR partition

5.2.2 Boundary processing

As we have discussed, the modules in DPR project are relatively dynamic. To make dynamics between modules possible, clear boundaries are needed. In earlier PlanAhead versions, or even before PlanAhead was born, the bus-macro (sometimes with enable signal, sometimes with FF) is the conventional component to build up the boundary. And with updated PlanAhead, partitions have weakened such boundaries. However, to have an overall timing convergence, the boundaries between partitions should be decoupled with FF's.

Xilinx Partial Reconfiguration User Guide (V12.3) states, "*it is very important to register the partition boundaries, and to use enables with these registers. During reconfiguration, the activity in these regions is indeterminate and could lead to design corruption if the output of the reconfiguring logic is used. Therefore, you should register boundaries with enables to disable the reconfigurable region during reconfiguration.*" [36]

Static module keeps working during dynamic reconfiguration. In a DPR project, multiple dynamic modules are allowed, but static module is actually one. We can set multiple partitions for static part during development for the sake of easy handling. However in real operation, the static module is running as a whole.

5.2.3 Dynamic module

A dynamic module is the loader site to shift new bit files in or out and geometrically it is usually a rectangle region in the FPGA. Through reconfiguration, a dynamic module is filled with a version of the partial bit file and then it works as a part of the whole design after the boundary is enabled. It can be filled with blank bit-stream to same power. For a clear isolation with the static module during reconfiguration and to operate seamless after reconfiguration, proxy logic is needed around dynamic module(s), which is made of LUT1's.

5.2.4 Reconfiguration methods

The first and simplest option is to use JTAG chain for reconfiguration. However sometimes the application needs to be simplified and SMAP methods are difficult to realize. A familiar approach is to embed a processor into the FPGA, and via ICAP port the DPR is done. Figure 5-6 shows the structure.

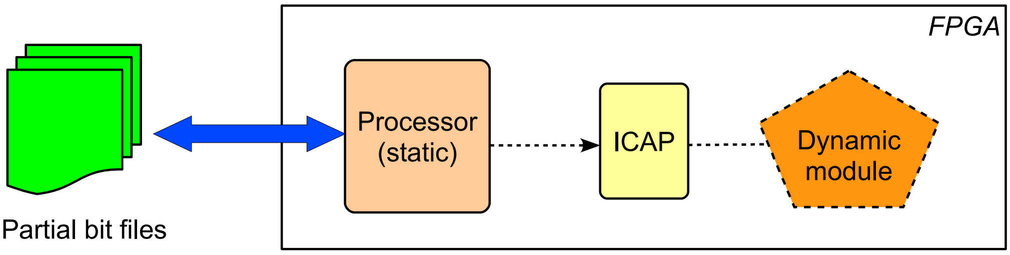


Figure 5-6 DPR via ICAP

5.3 Implementation – Virtex4 FX60

The Active Buffer board, as briefly described in previous chapters, is plugged into the PCI Express slot of its host machine. On the ABB an FPGA (Xilinx XC4VFX60) provides the base functionality that is receiving, buffering and forwarding the hit data. For receiving the data, two MGTs (**M**ulti-**G**igabit **T**ransceivers) of the FPGA are used together with SFPs (**S**mall **F**orm-factor **P**luggables). For the active buffering two DDR SDRAM modules are connected to the FPGA. And for the data forwarding four MGTs are connected to a PCI Express connector. We are using the Xilinx Virtex4 FX60, since this chip comes with the needed functionality (e.g. MGTs), it is big enough to contain all the needed logic and it can be reconfigured. The reconfigurability is used on the one hand for prototyping, on the other hand to provide scalability. In a high energy physics experiment, like CBM in GSI, the surrounding conditions can change due to measured values. Thus it is possible that the ABB functionality has to be changed after the installation of the ABB into the host PC. For this we need the FPGA's reconfigurability.

The ABB provides the DMA functionalities *downstream* and *upstream*, targeting high-speed data transfer between the host memory and the event buffer (DDR SDRAM). Furthermore the ABB supports PCI Express PIO transactions to enable parameter control and status inspection over the board. In this paper, *downstream* DMA names the DMA transfer from the node host memory to the event buffer on the Active Buffer board while *upstream* DMA denotes the opposite direction. A PCI Express interface is much work if starting from the scratch. Thus we build it along with the Endpoint IP Core (V3.6) provided by Xilinx, which takes care of the physical and data link layer transactions of PCI Express. On the transaction layer, we build one PIO channel and two DMA channels, *downstream* and *upstream* respectively. The Endpoint IP core has a data width of 32 bit and 4 lanes, requiring 250MHz operating frequency for the DMA logic. In PCI Express (Base Specification 1.1), the transaction is realized via packets. For example, a write operation uses “posted” TLP, which needs no acknowledge from the target. A read operation uses “non-posted” TLP, which involves a 2-step routine: first a read request is send out to the target, then the target responds with one or more packets, containing the requested data or returning unsuccessful information. Due to such features the PCI Express interface supports full-duplex mode and the bandwidth of both directions can be utilized independently. The overall throughput can be optimized using the according independent thresholds. The Base Specification has a restriction regarding the maximum (non-posted) read request size, namely MAX_READ_REQUEST_SIZE, which has one of the following values (in bytes): 128, 256, 512, 1024, 2048 or 4096. Also the maximum (posted) payload size, namely MAX_PAYLOAD_SIZE, is settable. Possible values are the same as those above. The *downstream* DMA logic has to obey the MAX_READ_REQUEST_SIZE when it is sending read requests to the host. The *upstream* DMA logic has to obey the MAX_PAYLOAD_SIZE when it is sending packets with payload to the host memory. The DMA engines in the Active Buffer system have a proven performance close to the maximum speed of the DDR SDRAM chip, which is about 8 Gbps. The operation pattern of the DMA engines is scatter-gather DMA, which processes chain-like transactions.

5.3.1 The problem

Unfortunately, the special properties of PCI Express do not allow a reconfiguration of the whole FPGA due to link details. During the reconfiguration of the FPGA, its input and output pins are set to high-impedance. This causes the host PC to deactivate the PCI Express slot until the next hard reset. As a consequence, every reconfiguration of the PCI Express core area demands a host reboot. For prototyping this behaviour is very disturbing. For the detector runtime it is absolutely inadmissible, since the host PC is part of a complex cluster network and can’t be restarted at will.

A second problem comes with the DMA thresholds (MAX_READ_REQUEST_SIZE and MAX_PAYLOAD_SIZE). The safest way of implementation is to use the lowest values for both settings (that is 128 bytes). However, this has been proven unable to provide the best performance, which is crucial in our application. To get the best performance, the DMA engines have to be able to calculate the suitable request size and payload size according to the PCI Express configuration space and they have to be able to set these calculated values dynamically. The problem is that such flexibility costs logic resources as well as verification effort. It also causes timing errors, due to the very tight timing constraints of 4 ns (250 MHz). So the situation is that we need to be able to set the thresholds dynamically, but we do not have the logic speed to realize the necessary switch logic inside the FPGA.

5.3.2 The solution

The PCI Express link problems are solvable in two ways. Firstly, one could use an external PCI Express chip, that realizes the PCI logic and will never be reconfigured – and thus never causes any link problems. Secondly, one could divide the FPGA using partial reconfiguration. Using this method, one part (the PCI Express logic) would be defined as *static* and would never be reconfigured, other parts would be defined as *dynamic* and could be reconfigured, while the *static* part continues uninterrupted. Both methods have drawbacks. A separated PCI Express interface chip brings additional costs and additionally occupied IO pins of the FPGA. The one-chip solution uses 8 pairs of differential pins of the FPGA, while the two-chip solution occupies more than 80 pins. Obviously, redundant pin usage not only takes space, but also decreases the reliability of the system. Figure 5-7 explains the idea.

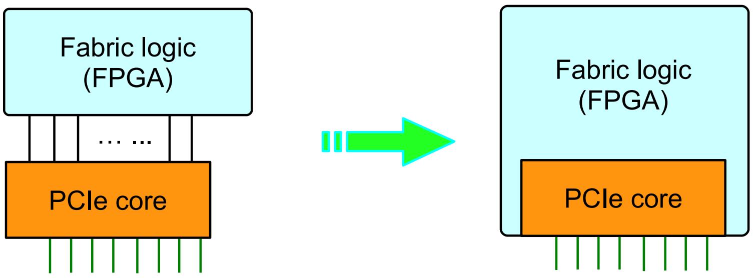


Figure 5-7 Multiple-chip and single-chip solutions for PCI Express

The DPR solution needs more internal FPGA resources than the one with an external PCI Express chip and the DPR tool flow is more complex than the integrated one. Since in our case the internal resource consumption of the FPGA is not critical, but the pin consumption is very critical (besides the memory chips, there are a number of mezzanine connectors on the ABB that utilize most of the IO pins of the FPGA), the one-chip solution using DPR is the better choice. Furthermore DPR also provides a very elegant solution for the threshold problem. It makes it possible to change the thresholds on demand without the need for extra logic inside the DMA controller. For this one can set up two dynamic modules for the two DMA engines. Each has 6 versions of dynamic bit files. Every version is correlated to a MAX-SIZE parameter value and loaded on demand. Hence, the flexibility is implemented, while the design complexity stays low and thus a timing closure becomes possible. It might also be possible to use only one dynamic module, containing both DMA engines, but this way the number of partial bit files would increase to `6 × 6 = 36`, instead of `6 + 6 = 12`. The PIO channel also has settable thresholds controlling its maximum buffer sizes. However, in our application the minimum MAX-SIZE setting (128 bytes) is never exceeded in the PIO channel and thus we do not have to change that parameter dynamically. Thus, the PIO channel logic is placed in the static part.

So, for a flexible structure and scalability, we applied DPR technology to the project. This way, the fixed part, including PCI Express core and DMA register space, is kept static, while the data transfer logic, mainly the DMA engines, is put into dynamic modules. That way we can reconfigure the DMA engines any time during the operation, without having to reboot the host machine. Furthermore DPR helps reducing the design complexity, since the several MAX-SIZE parameter values are realized via according partial bit files.

5.3.3 Implementation

In our project DPR helped solving many problems, but DPR does not only come with advantages. It also has its drawbacks. One of the biggest hindrances regarding DPR is the need to change the tool flow and the resulting need to change the design hierarchy. To realize the dynamical partial reconfiguration we used the Xilinx tools PlanAhead 9.2 and ISE 9.1 (with an additional DPR patch). It is possible but not recommended to do partial reconfiguration on Xilinx FPGAs without PlanAhead. PlanAhead makes the partitioning of the chip and the placement of the bus-macros much easier. Furthermore, PlanAhead introduces a new tool flow, as in figure 5-8, making it possible to generate the static bit file and all needed dynamic bit files with one tool. Using PlanAhead we have to change the design's hierarchy. The bus macros have to be placed in separate components which are instantiated at top level. The dynamic and the static parts have to be placed in separate components which are instantiated at top level, too. Of course, these components can contain sub-components. Also all I/O pins should be placed at top level.

The next step, after generating a suitable VHDL design, is the synthesis. First the static subcomponent and the dynamic sub-components of the VHDL design have to be synthesized. These components are marked as top module and synthesized one after another. For PlanAhead 9 every dynamic module must have the name given in the top level component. It is recommended that the XST does not add any input buffer or output buffer, as these components are internal. After every synthesis the resulting ncd file has to be backed up from the project directory, because the project must be cleaned afterwards. Finally the real top module has to be selected. Any other VHDL component has to be removed from the project. Thus, the top module component handles the dynamic and the static modules as black boxes.

For the further steps with PlanAhead 9, the following files are recommended: the ngc files of the top module, of the static part and of every realization of the dynamic part, an ucf file for the used board and the NMC files of the used bus-macros. PlanAhead is used to place the static part, the dynamic parts and the bus-macros. The next step is to generate the bit files of the static and dynamic parts of the design using the patched *par* of ISE 9.1. After this, PlanAhead is able to merge the different bit files. That means it generates one static bit file with empty dynamic areas, one full bit file containing the static part and a default dynamic module for every dynamic area and partial bit files representing all the dynamic modules.

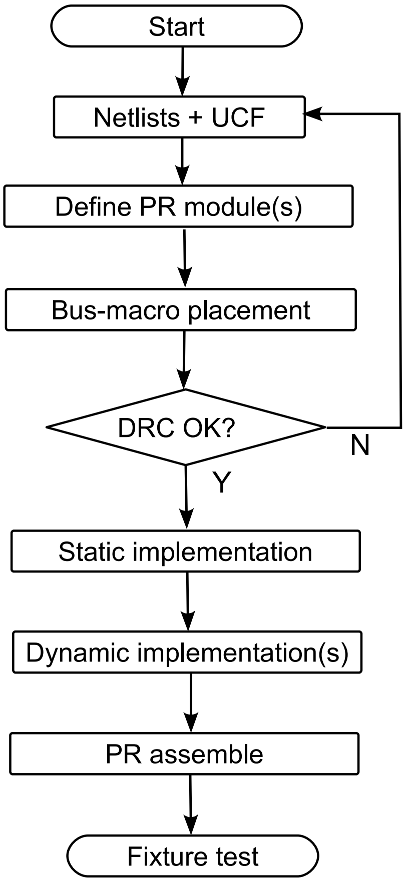


Figure 5-8 DPR procedure in PlanAhead 9

A dynamic module is placed inside a dynamic area. Using PlanAhead, dynamic areas have to be rectangles. In our project the PCI Express Endpoint IP core, the DMA register space and the PIO transactor reside in the static part. The two DMA engines make up two dynamic modules. The interfaces between the static and the dynamic parts are the bus-macros. A bus-macro is a hard macro built on ordinary slices, usually containing an 8-bit wide bus. It can be customized with Xilinx tools such as the FPGA Editor and is also available from the bus-macro library provided by Xilinx. Properties like direction, synchronization, etc. have to be considered by creating or choosing a bus-macro. Regarding the timing there are two versions of bus-macros: synchronous and asynchronous. Synchronous bus-macros have a better timing behaviour, since they refer the data only at the rising edge of a connected clock and thus they separate the timing of the dynamic part from the timing of the static part. The disadvantage is the resulting additional pipeline step, which demands logic changes to keep the correctness of the logic. Using asynchronous bus-macros, the logic does not have to be modified, but the timing is much harder to meet. In fact, for our DMA logic the combinatorial delay is a little bit too high to get timing closure. This is primarily caused by the necessary high clock rate of 250 MHz. Hence, the bus-macros in our project are all synchronous. Furthermore, we use those with enable pins to prevent unpredictable behaviour caused by unexpected signal toggling during the partial reconfiguration. The reconfiguration software holds the enable pins of the bus-macros low until the corresponding dynamic reconfiguration is done.

In the integrated version of DMA logic, the maximum size parameters are variables. While in the partial reconfigurable version, those two parameters are constants, defined via generics (VHDL) or parameters (Verilog HDL). In DPR mode, the FPGA is configured at power-up with the static bit stream, which supports basic PCI Express operations as well as basic PIO operations. After the host system (Linux 2.6) is up and the PCI Express driver is loaded, the software first checks the PCI Express configuration of the host system, especially the maximum value parameters, and accordingly configures the two DMA channels with proper partial bit streams. The dynamic reconfiguration process for every dynamic module is finished within a few milliseconds, since every partial bit stream is small (less than 100 KB in size). Such dynamic reconfiguration is only necessary once at system start-up, and hence, the reconfiguration overhead can be neglected.

Generally the flexibility and scalability of a design is increased by putting more logic into the dynamic parts. However, more dynamic logic will cause more boundaries between the static and the dynamic part. If there are too many bus-macros in a project, not only the available resources are reduced for implementing logic, but also the timing constraints are more and more difficult to meet, no matter whether synchronous or asynchronous bus-macros are used. To have a higher nominal frequency for a DPR project, synchronous bus-macros are strongly suggested. However, as mentioned, the synchronous bus-macros introduce an additional cycle of delay for the signals across it. Hence, to reduce resulting logic changes, it should be attempted to use the synchronous bus-macros together with signals that are not cycle-sensitive. In our project, it is the transaction buffers that contain the bus-macros. When introducing the bus-macros we changed the FIFO controller a little bit, so that the writing FSM (finite state machine) contains the synchronous bus-macros and thus contains one more pipeline step. Due to the already existing FIFO, the additional pipe line step did not change the general behaviour of the design. From an external point of view, only the FIFO depth increased by one. For the FIFO controller it was much better to place the bus-macros at the write port as to put them at the read port, since the read operation is realized via a complex handshaking and the delays are critical for the logic's correctness. By contrast the FIFO write operation is not that critical. We just had to use the ALMOST_FULL signal for flow control instead of the FULL signal, as shown in figure 5-9.

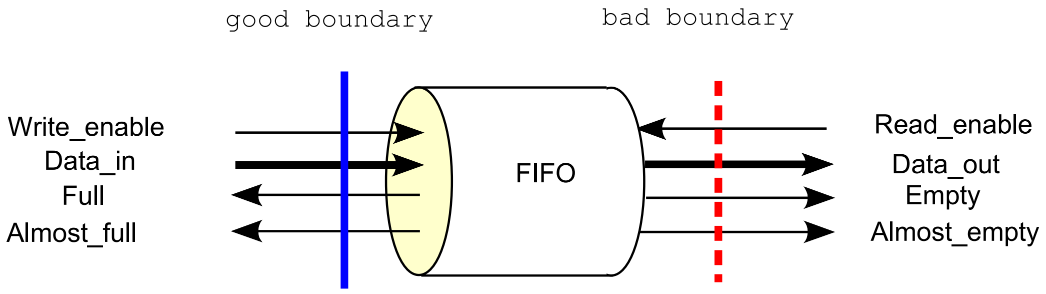


Figure 5-9 Partition boundary examples

5.3.4 Test analysis

A very important concern for a DPR project is that the design still meets its specifications and still meets timing after partitioning it. To reach this, one or more integrated designs, not using DPR but using bus-macros, can be tried before implementing a partitioned one. Thus, introducing DPR, the DMA logic in the Active Buffer system stays unchanged.

Our first tests referred to the performance. Some performance penalty was expected since the synchronous bus-macros come with two additional clock cycles of delay for the DMA engines. However, in the tests, the performance difference between the integrated version and the dynamic version was as small as the measuring error range and thus can be neglected. Closer measurements showed, that this result is caused by the use of the FIFOs inside the transaction buffers, which are almost never empty during the DMA. Table 5-1 shows the typical performance comparisons between integrated and partial designs. The tests were performed 100 times and the values listed are averaged.

Table 5-1 DMA performance comparison

	DMA size (kB)	Integrated (MB/s)	DPR (MB/s)
Downstream DMA write	64	673.9 ± 0.32	673.9 ± 0.78
	256	694.8 ± 0.15	692.0 ± 0.10
	1024	700.1 ± 0.02	696.7 ± 0.73
	4096	701.2 ± 0.02	701.2 ± 0.08
Upstream DMA read	64	430.8 ± 1.75	432.9 ± 0.18
	256	421.2 ± 0.04	437.3 ± 0.04
	1024	433.8 ± 0.02	435.4 ± 0.02
	4096	437.3 ± 0.01	437.7 ± 0.01

We also measured the resource consumption. For this the thresholds `MAX_READ_REQUEST_SIZE` =512 and `MAX_PAYLOAD_SIZE`=128 were used. Table 5-2 shows the resulting resource consumption in detail. We can see that the resource utilization for DPR is not greater than that of the integrated design. Since the integrated design has to implement all possible thresholds it needs more logic inside the DMA modules than the dynamic design. Using DPR only **one** threshold has to be implemented on the chip. The other thresholds are represented by partial bit files that reside in the host system RAM and do not consume any space on the chip. However, using DPR one has to implement the bus macros, which consume LUTs (**L**ook-**U**p **T**ables) and FFs (**F**lip **F**lops). Thus, in our project, the resource consumption of the integrated design and of the dynamic design is almost equal.

Altogether we use 87 (horizontal and vertical) 8-width bus-macros, 41 for upstream module and 46 for downstream. These bus-macros sum up 696 signals. 34 of these signals are open (meaning not used), since one bus-macro always implements 8 signals, whether they are needed or not. A downstream bit file is 81 738 bytes in size and an upstream bit file is 86 848 bytes in size (as reference: the total configuration bit stream for XC4VFX60 is 2 625 438 bytes). The partial bit stream of the upstream module has a larger size than the one of the downstream module although the upstream module occupies fewer slices. The reason is that some RAMB16s and FIFO16s increase the bit stream size. In the downstream area there are no such primitives included.

Table 5-2 Resource consumption in Virtex4 DPR experiment

Resource type	FFs	LUT4s
DPR Static	6901	8742
DPR module 1: upstream DMA engine	460	559
DPR module 2: downstream DMA engine	602	811
Bus-macros	2088	1392
DPR total	10051	11504
Integrated total	10818	12051

5.3.5 Conclusion to the Virtex4 DPR

The tests show that DPR solves the problems very well.

The ABB is part of the CBM high energy physics experiment and realizes the interface between read out elements and a computer cluster. It receives the hit data through an optical port and writes it to the PC's memory via PCI Express. Due to the high data rates in the CBM experiment, new hard- and software solutions such as DPR are needed to meet the demands. Regarding the ABB, DPR solves two major problems. First, no external PCI Express chip has to be used to enable updating reconfigurations. Thus, DPR helps saving costs while offering high scalability. Second, DPR permits a much higher flexibility regarding performance critical thresholds inside the DMA engines. It allows to change thresholds on demand without the need for extra logic inside the DMA controller. Hence, the design complexity stays low and thus a timing closure becomes possible.

Our tests show that DPR is feasible for large and complex designs. The dynamic design has been proven to work correctly. Furthermore, we successfully implemented the synchronous bus-macros (coming with an additional pipeline step) without any performance lost. Due to the higher flexibility of the dynamic system, the performance critical thresholds can be set the best way – and thus the performance of the dynamic system is much better than the performance of an integrated design, which realizes only one value. For resource comparison we used to implement a flexible integrated design, but this design never met the timing constraints under ISE 9.1. The resource consumption of the flexible integrated design is almost the same as the one of the dynamic design. The reason is, that using DPR some elements (like bus-macros) have to be added, but on the other side the flexibility is realized via partial bit files and not via resource-killing multifunctional hardware. Many features are still being exploited. Our next step intends to have a pair of alternative DMA channels, in which the direction of a DMA channel can be reconfigured. Besides the current DMA channel-pair mode, we can have two upstream channels or two downstream channels at the same time, depending on the demand of the transportation situation around the Active Buffer. This structure is sure to give the system more powerful functions which cannot be done without DPR technology.

Difficulty in boundary decoupling and synchronous bus-macro insertion is a considerable part to practice a DPR project. In high-frequency designs as our DAQ projects, the boundaries to do DPR need special treatment to keep the proposed functions as well as to succeed in timing closure. Such work is usually related to pipeline insertion or shifting, and most probably, enable signals appending. In a complex design, such rewrite needs some time.

5.4 Implementation – Virtex6 LX240T

With PlanAhead 12.4, we do DPR to ML605 development board. Similar to the Virtex4 DPR project, we keep the PCI Express core in the static part and the data path logic in the dynamic module in our Virtex6 DPR project. Difference lies in the boundary processing. In PlanAhead 12.4, we use boundary decoupling to replace the deprecated bus-macros. But the effort is the same, additional synchronous pipelines are to be inserted or shifted.

5.4.1 Boundary decoupling and partitions

Without synchronous bus-macros, DPR under PlanAhead 12 needs less effort to be implemented. However, the boundary decoupling is still an issue. The inputs of the reconfigurable module are generally not so critical. But the outputs of the reconfigurable module should be disabled during the partial reconfiguration, to prevent the DPR process causing unpredicted instability. Such disable is done by means of the FFs with enable pin. These FFs should not situate inside reconfigurable module, because during reconfiguration, the enable pins should not be floated. These FFs should formally be in the static module. If there were no disable control, the partial reconfiguration would probably crash the system, as proved in our PCI Express DMA DPR tests. Figure 5-10 shows the location of the tri-state control along the boundary and outside the reconfigurable module.

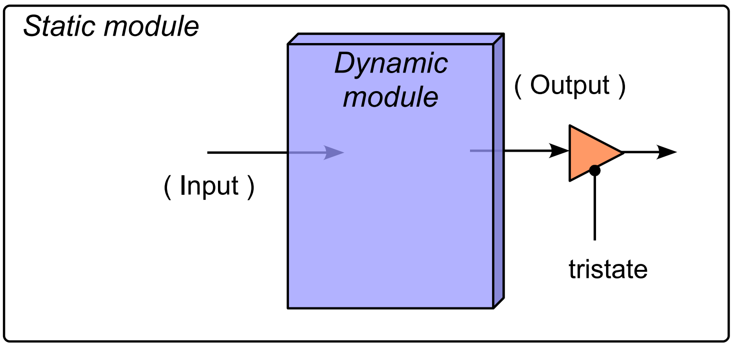


Figure 5-10 Tri-state control position in DPR

Figure 5-11 shows the experimental partitions for the Virtex6 DPR project. The two high-lighted blocks are the DMA_FSM modules, top one for downstream and the bottom one for upstream.

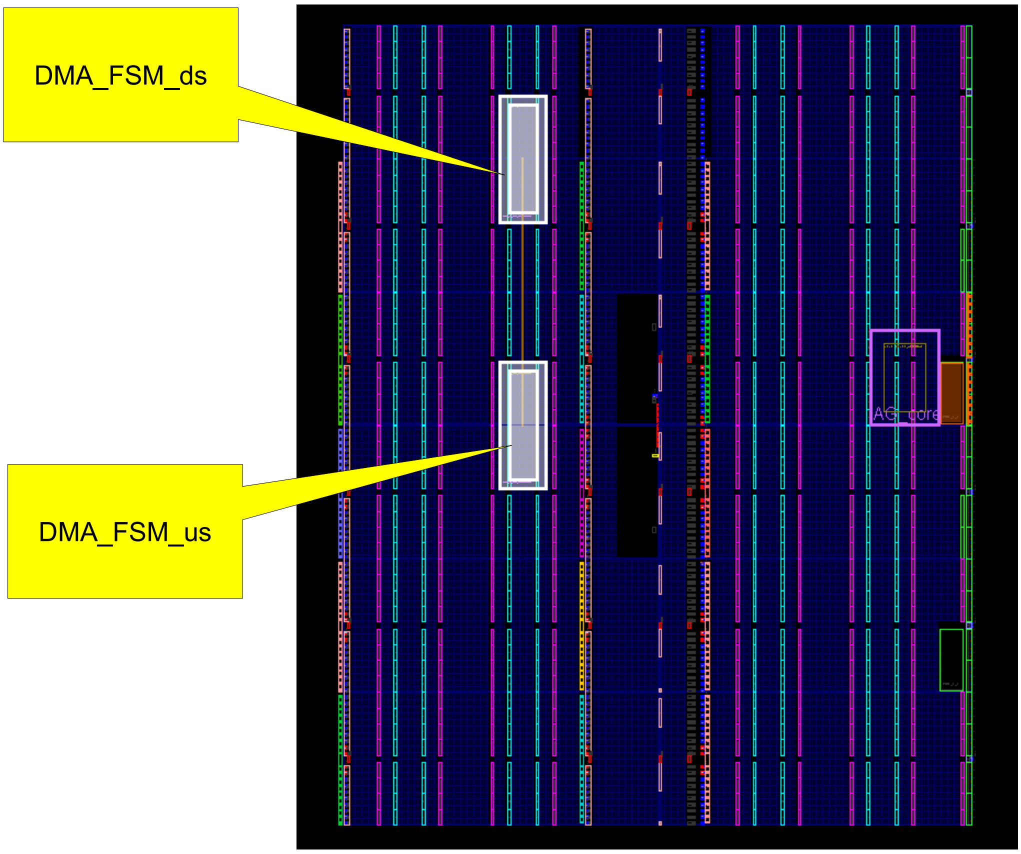
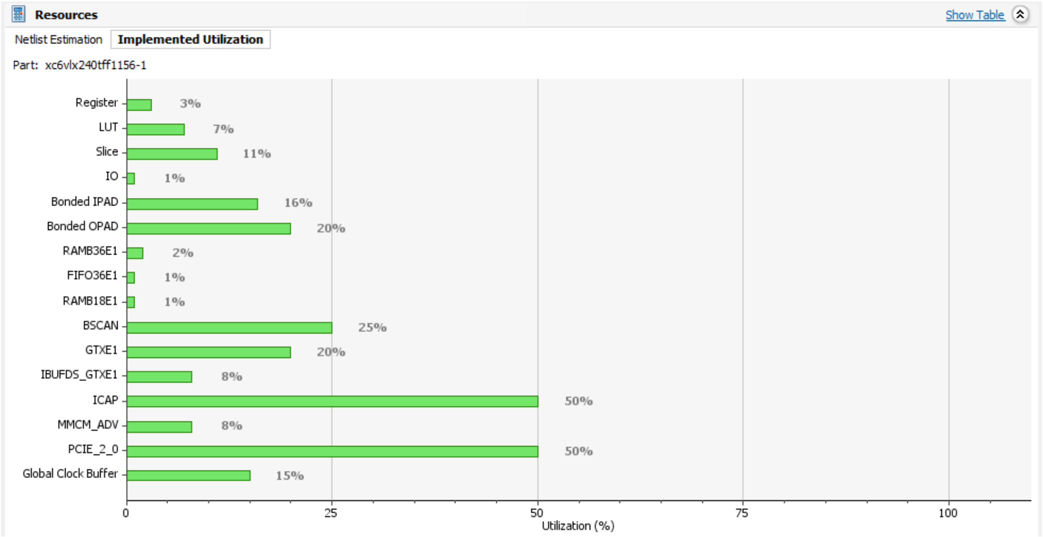


Figure 5-11 Partition snapshot in PlanAhead 12.4 (FPGA: XC6VLX240TFF1156-1)

Figure 5-12 shows the resource consumption statistic in the ML605 DPR experiment.



Part: xc6vlx240tff1156-1

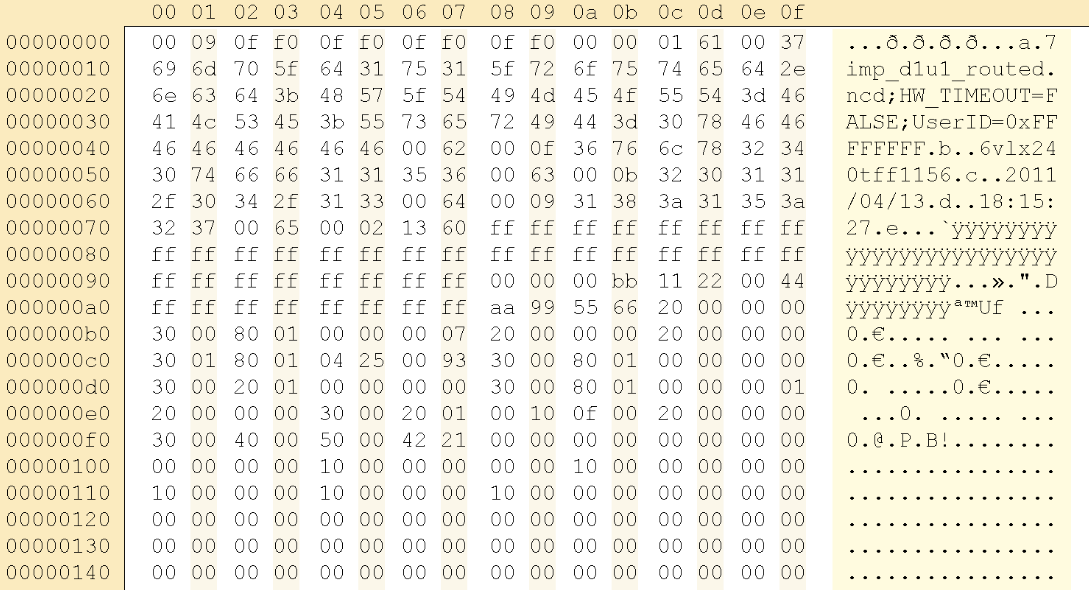
Figure 5-12 Resource utilization of the Virtex6 DPR experiment in PlanAhead 12.4

5.4.2 Bit file format of Xilinx FPGAs

Xilinx bit file format. Xilinx bit file is composed of two parts, header and the data. The header is divided into five sub-sections, a, b, …, e. Sub-sections are separated by the letters 'a', 'b', …, 'e', as shown in figure 5-13. There is a synchronization word “AA 99 55 66” for the bit file loader. There are also words for bus width recognition, “00 00 00 BB” and “11 22 00 44”, but they ignored by ICAP. After letter 'e' and before the F-padding words, is the 32-bit length field. In figure 5-13, “00 02 13 60” means there are 136 032 bytes after the header.

Each partial bit-stream size is 136 150 bytes. Compared to the full-size, 9 232 567 bytes, the configuration time for a dynamic module is less than 2% that of the full configuration.

The bit file configuration sequence can be found in Xilinx configuration guides. [88] [89] [90]



```
          00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f
00000000  00 09 0f f0  0f f0 0f f0  0f f0 00 00  01 61 00 37   ...ð.ð.ð.ð...a.7
00000010  69 6d 70 5f  64 31 75 31  5f 72 6f 75  74 65 64 2e   imp_d1u1_routed.
00000020  6e 63 64 3b  48 57 5f 54  49 4d 45 4f  55 54 3d 46   ncd;HW_TIMEOUT=F
00000030  41 4c 53 45  3b 55 73 65  72 49 44 3d  30 78 46 46   ALSE;UserID=0xFF
00000040  46 46 46 46  46 46 00 62  00 0f 36 76  6c 78 32 34   FFFFFF.b..6vlx24
00000050  30 74 66 66  31 31 35 36  00 63 00 0b  32 30 31 31   0tff1156.c..2011
00000060  2f 30 34 2f  31 33 00 64  00 09 31 38  3a 31 35 3a   /04/13.d..18:15:
00000070  32 37 00 65  00 02 13 60  ff ff ff ff  ff ff ff ff   27.e...`ÿÿÿÿÿÿÿÿ
00000080  ff ff ff ff  ff ff ff ff  ff ff ff ff  ff ff ff ff   ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000090  ff ff ff ff  ff ff ff ff  00 00 00 bb  11 22 00 44   ÿÿÿÿÿÿÿÿ...».".D
000000a0  ff ff ff ff  ff ff ff ff  aa 99 55 66  20 00 00 00   ÿÿÿÿÿÿÿÿª™Uf ...
000000b0  30 00 80 01  00 00 00 07  20 00 00 00  20 00 00 00   0.€..... ... ...
000000c0  30 01 80 01  04 25 00 93  30 00 80 01  00 00 00 00   0.€..%.“0.€.....
000000d0  30 00 20 01  00 00 00 00  30 00 80 01  00 00 00 01   0. .....0.€.....
000000e0  20 00 00 00  30 00 20 01  00 10 0f 00  20 00 00 00    ...0. ..... ...
000000f0  30 00 40 00  50 00 42 21  00 00 00 00  00 00 00 00   0.@.P.B!........
00000100  00 00 00 00  10 00 00 00  00 00 10 00  00 00 00 00   ................
00000110  10 00 00 00  10 00 00 00  10 00 00 00  00 00 00 00   ................
00000120  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000130  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000140  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
```

Figure 5-13 the header format of a partial bit file

5.4.3 ICAP primitive

ICAP in Virtex6 FPGAs is named ICAP_VIRTEX6, illustrated in figure 5-14. The ICAP_VIRTEX6 component has 4 inputs and 2 outputs. For DPR we need read from ICAP to verify the device ID and similar information. `CSb` must be low for ICAP write and during configuration, `RDWRb` must remain valid low, otherwise abort will be triggered and configuration fails. Therefore we have three options to write ICAP,

- CLK-mode: controls the clock edge;
- CSb-mode: uses synchronous clock and controls `CSb`; and
- hybrid mode (both `CLK` and `CSb` are variable).

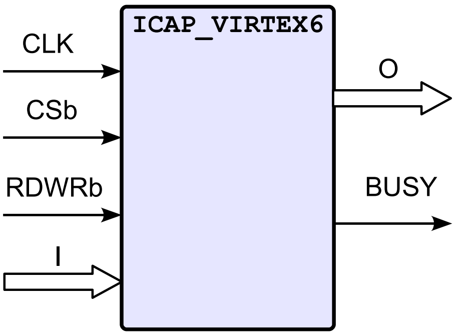


Figure 5-14 ICAP_VIRTEX6

ICAP clock rate cannot exceed 100 MHz. For generality and simplicity, we use the CLK-mode, which controls the CLK rising edge to get the configuration data into ICAP.

Since we have already PCI Express channel constructed between the software and the target FPGA, no more embedded processor is needed in the FPGA to transfer the partial bit file to the ICAP interface. Apart from figure 5-6, our DPR structure is as figure 5-15.

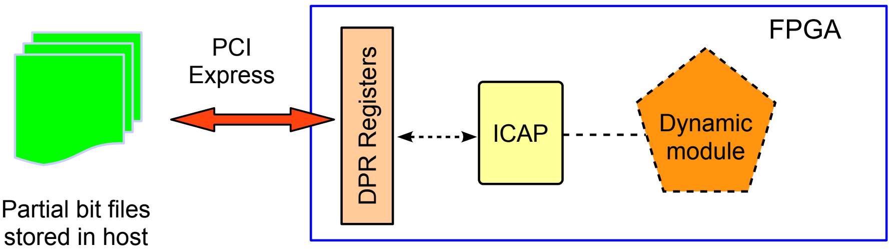
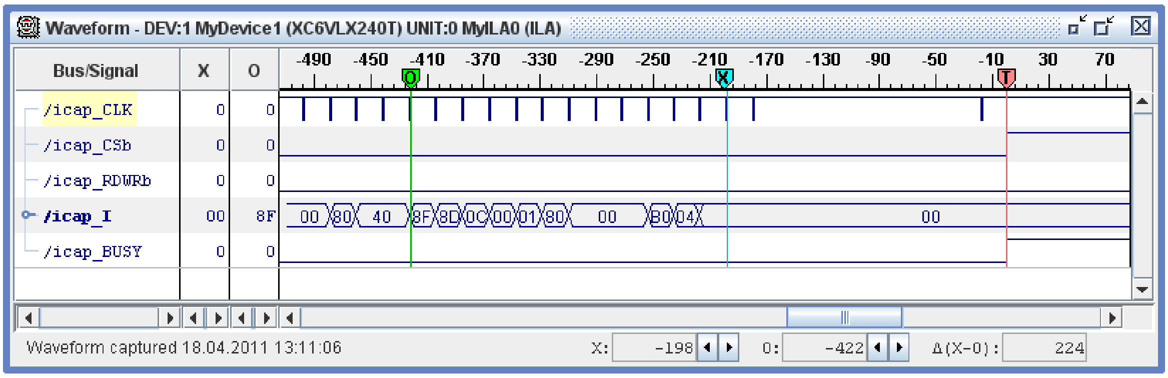


Figure 5-15 DPR over PCI Express

In terms of write, figure 5-16 is an example ChipScope view of ICAP partial reconfiguration. We use 8-bit ICAP data bus and every byte of data write is made by one PIO write to the ICAP control register (resident in BAR[0]). This PIO write draws the ICAP_CLK low and puts the next data to the input data bus (ICAP_I); then after this write, the data bus keeps unchanged and ICAP_CLK rises high, thereby one byte is sent to the ICAP at the rising edge of the ICAP_CLK. During the partial reconfiguration, the CSb and RDWRb are both valid low, until the whole bit file is transferred.


Waveform - DEV:1 MyDevice1 (XC6VLX240T) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
/icap_CLK	0	0
/icap_CSb	0	0
/icap_RDWRb	0	0
/icap_I	00	8F
/icap_BUSY	0	0

Waveform captured 18.04.2011 13:11:06   X: -198   O: -422   Δ(X-O): 224

Figure 5-16 ChipScope view: DPR via ICAP with Virtex6

With figure 5-16, we can estimate the bandwidth of such reconfiguration mode. The clock cycle to drive the ChipScope is 8 ns (125 MHz), we count 224 (from number -422 to number -198) captures altogether 12 `ICAP_CLK` cycles, then we have

$$12\ \mathrm{B} / 8\ \mathrm{ns} / 224 \approx 6.70\ \mathrm{MB/s}.$$

This estimation is out of pure hardware-level maximum rate. Therefore, the real speed should be lower than 6.70 MB/s.

ICAP program under Linux outputs as following.

```
mp-spirit2:/home/dpr# bin/icap_dpr bitfiles/imp_d2u2_us_partial.bit
Device ID verified.
DPR boundary closed.
DPR in process ...
DPR boundary opened.
DPR successfully finished.
  Bit file name : bitfiles/imp_d2u2_us_partial.bit
  Bit file size : 136150 bytes
  Reconfiguration time = 21.1862 ms
  Reconfiguration speed = 6.42634 MB/s

mp-spirit2:/home/dpr# bin/icap_dpr bitfiles/imp_d2u2_ds_partial.bit
Device ID verified.
DPR boundary closed.
DPR in process ...
DPR boundary opened.
DPR successfully finished.
  Bit file name : bitfiles/imp_d2u2_ds_partial.bit
  Bit file size : 136150 bytes
  Reconfiguration time = 22.1401 ms
  Reconfiguration speed = 6.14948 MB/s
```

Here we see the reconfiguration speed is about 6.1 ~ 6.4 MB/s, as we have estimated. Reconfiguration speed would be about four times faster, if we use 32-bit ICAP data bus.

Before and after any of the dynamic reconfigurations, the DMA tests work correctly as the integrated design. The performance test has almost the same result as figure 4-10 shows.

5.4.4 Conclusion and scenario

The DPR experiment on ML605 is preliminary and proves successful. Further applications will be deliberated and benefit. This experiment encourages more general DPR applications in computing acceleration. With the help of high-speed channel PCI Express, the dynamic reconfiguration over the ICAP port can be completed in several milliseconds. Also, the data to and from the reconfigurable area are transferred via high-performance DMA operations, which is an essential factor for acceleration tasks. PlanAhead is now providing a quick and reliable generation of partial bit files. Figure 5-17 shows the scenario.

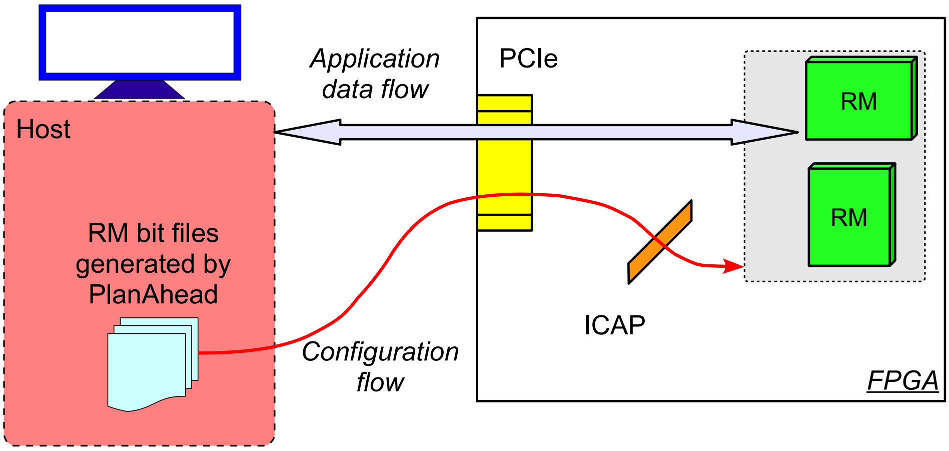


Figure 5-17 DPR scenario for computing acceleration

5.5 Implementation – Virtex5 LX110T

Similar to the Virtex6 project, we also try DPR on the AVNET Virtex5 board with PlanAhead 12.4. The partitioning is quite similar, with two DMA engines into dynamic area and the other parts into the static, as in figure 5-18. The implementation is successful and timing closure is achieved.

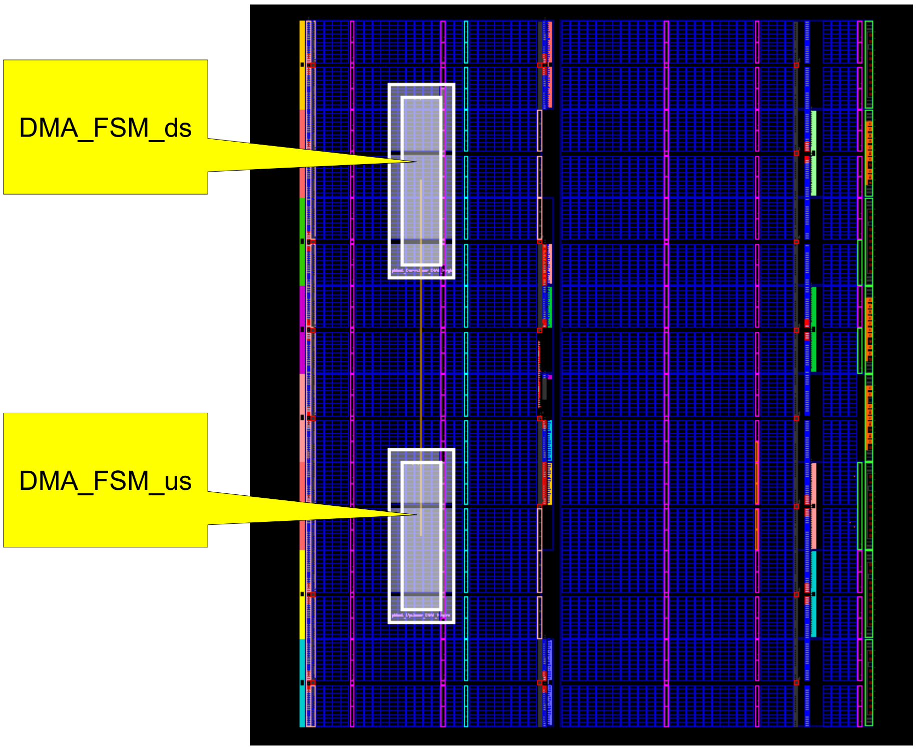


Figure 5-18 DPR Partition (FPGA: XC5VLX110TFF1136-1)

Figure 5-19 is the resource utilization of the Virtex5 DPR.

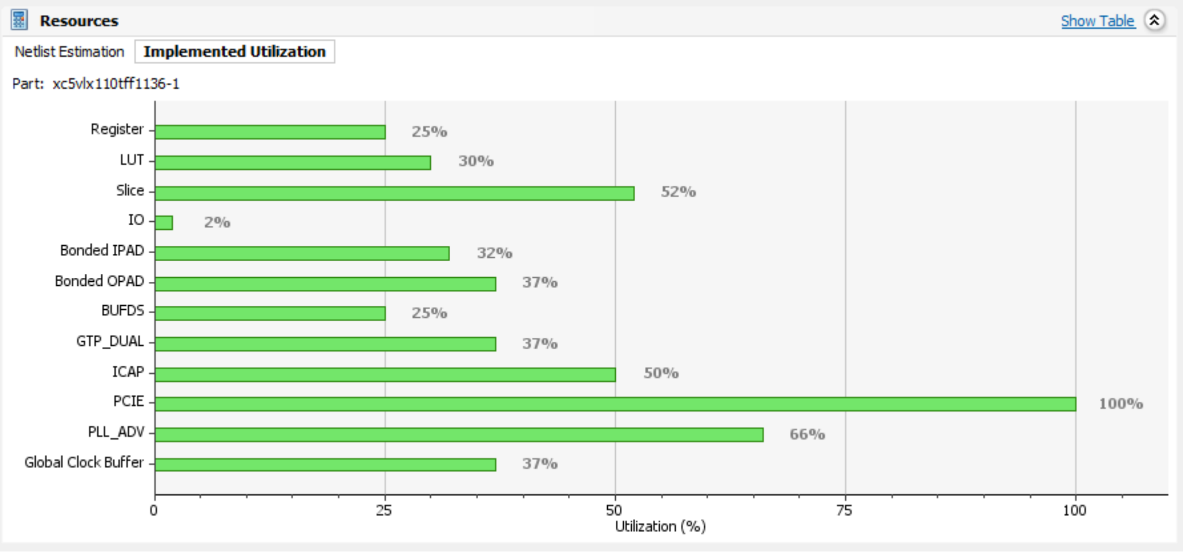


Figure 5-19 Resource utilization of the Virtex5 DPR experiment

The ICAP_Virtex5 works almost the same as ICAP_Virtex6, yet with lower bandwidth in our tests. Figure 5-20 is the ChipScope viewer snapshot of the ICAP_Virtex5.

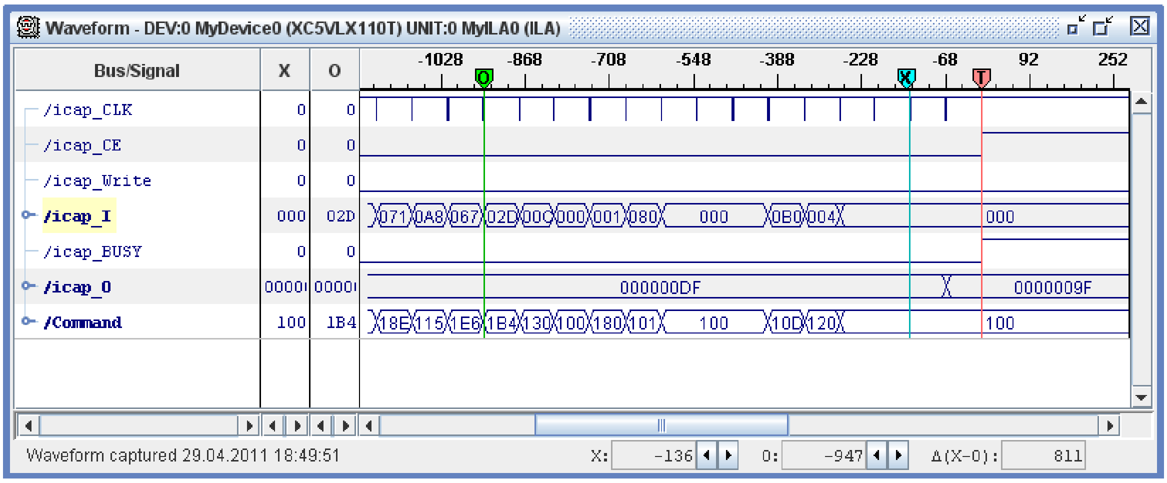


Figure 5-20 ChipScope view: DPR via ICAP with Virtex5

In figure 5-20 we can see the speed of ICAP in Virtex5 is slower than in Virtex6. Similar estimation can be made as

$$12\ \text{B} / 8\ \text{ns} / 811 \approx\ 1.85\ \text{MB/s}.$$

And the console output agrees with this estimation, as follows.

```
mp-spirit2:/home/dpr# bin/icap_dpr bitfiles/imp_u1_partial.bit
DPR boundary closed.
DPR in process ...
DPR boundary opened.
DPR successfully finished.
  Bit file name : bitfiles/imp_u1_partial.bit
  Bit file size : 215919 bytes
  Reconfiguration time = 137.948 ms
  Reconfiguration speed = 1.56522 MB/s
```

```
mp-spirit2:/home/dpr# bin/icap_dpr bitfiles/imp_d1_partial.bit
DPR boundary closed.
DPR in process ...
DPR boundary opened.
DPR successfully finished.
  Bit file name : bitfiles/imp_d1_partial.bit
  Bit file size : 215919 bytes
  Reconfiguration time = 127.749 ms
  Reconfiguration speed = 1.69019 MB/s
```

The software package of ICAP operation is not changed and it works correctly as with the Virtex6 project.

5.6 Limitation of DPR

Not all kinds of components can be put into the dynamic module, for example, the BUFG. Other limitations of DPR include

- Longer development time.
- Tricky treatment along the boundary.
- Boundary overhead.
- Modules/partitions cannot be used arbitrarily. Instead they have to be well planned. A version is limited to be implemented into a pre-defined area.
- Rewrite of HDL with high rate of clock domains.
- PR-User Guide states that in general, the clock rate is expected to have 10% degradation and the packing density is expected to be under 80%. [36]

In a word, the DPR is not as straightforward as it is supposed to be. However, with the effort of the involved scientists and technologists, the difficulty is getting smaller and smaller. For example, the proprietary tool for DPR of Xilinx Inc., PlanAhead is released with version 13.1. [91]

5.7 Summary

Despite of the limitations in DPR, it is still taken as a promising technology in FPGA applications. This is why the major FPGA vendors are investing more effort into this race. The resource overlapping is an advantage in the generalized computing acceleration, where the hardware modules can be loaded just like the software modules. Similar appealing properties encourage vivid research activities in academy and in industry.

In DPR projects with higher global clock rate, the boundary processing is quite an issue during the HDL code rewriting. In the next chapter, an HDL code reuse tool will be introduced, trying to find the solution for similar problems.

Chapter 6 HDL code reuse tool

6.1 DPR review and HDL reuse

As we have seen in the DPR practice, the HDL adaptation is an amount of work, tedious and error-prone. Without synchronous bus-macros or boundary decoupling, the DPR project might suffer from timing closure failure between modules. Manual modification to compensate the boundary requirements takes much time. The tradition of the computer technology shows that such work is better done by the machine.

A DPR project on FPGA sometimes requires HDL code rewrite upon the original HDL design, especially for the heavy-coded and high-frequency modules. This is true when synchronous pipelines are to be used along the boundary between static and dynamic modules, because asynchronous elements make a condensed FPGA design difficult to achieve the timing closure. However, synchronous units bring an additional cycle of delay to the signals across the DPR boundary. For instance, the DPR practice on the Active Buffer board (Xilinx Virtex4 FX60 FPGA, 4-lane PCI Express DMA function, over 12 000 occupied slices) for the CBM experiment, requires a 250 MHz global clock rate and must use synchronous pipeline stages along the reconfiguration boundary between the static module and the dynamic module. [53] The PCI Express core and the global control logic are in the static module while the DMA engines are in the dynamic module. If asynchronous elements were used, there would be no extra cycle of delay to be compensated and little code rewrite would be needed, but the implementation would not fulfil the timing constraints (250 MHz). So it took us much effort to verify the DPR version of the HDL code. [70]

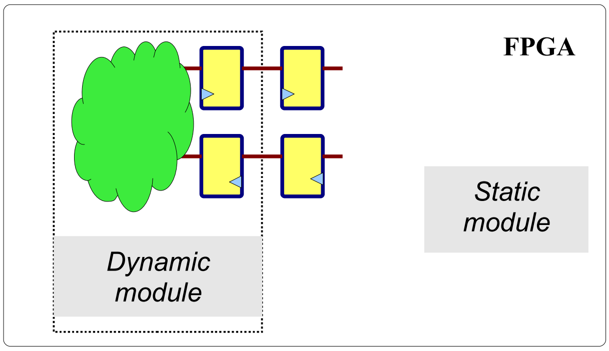


Figure 6-1 Modules and boundary in the DPR partition

Similar requirements come also from other logic design practices. For example, pipeline insertion into the logic needs similar modification. After the functional logic design is done and the logic is ready for implementation, sometimes one or more extra pipelines need to be inserted into a critical path to meet the timing requirement of the operating clock rate. However, code rewriting and re-verification due to pipeline insertion are sometimes too much work for the developer, especially in a large-scale logic project, even though only one cycle delay on some signals is to be considered. For complex logic, a small change somewhere might have a chain effect to a function unit many pipeline stages away, so the designer or rewriter has to trace the change throughout the whole project and to make sure every branch from that changed point still follows the proper timing. For some DPR projects, such rewritten logic may take several weeks for verification to get running properly, as in our Active Buffer DPR project.

Therefore, an automatic tool is needed for such HDL code reuse. And the questions aroused from DPR can be generalized to a larger scope of logic rewrite practice. Such a tool is supposed to provide more flexibility and higher efficiency in using existing HDL codes for new applications. It aims not only at the cycle-delayed application in DPR, but also at a general applicability for the HDL code reuse practice.

As mentioned in the previous chapter, since PlanAhead 10, the bus-macro is no longer needed. However, for high-frequency designs, the boundary is recommended to be proxied with synchronous elements, or in other words, all kinds of FFs. For some proxy logic, enable pins are recommended to have a steady reconfiguration process and to protect the static logic state.

The solution to the previous DPR is done by equivalent removing synchronous stages from the original design, namely the bus-macros. And for the new DPR projects with PlanAhead, we still use the pipeline-stage-remove tool as the first step, to find out the pipeline combination that delays all paths exactly the same cycles. Then a second part is added to the tool, which inserts pipelines to the ports. In this way, we have the total solution with little extra effort, because the second part takes much less time for development. This remove-insert effect is equivalent to pipeline shift.

In an FPGA DPR project, the boundary elements are to isolate the logic regions inside the FPGA during reconfiguration. In figure 6-1, the FF-alike symbols denote the boundary elements. The static module keeps running while one dynamic module is reconfigured. To keep the external behaviour of the dynamic module as it is supposed to be, the input ports have to compensate the extra cycle of delay and the outputs should be correspondingly advanced one cycle ahead due to the insertion of synchronous pipeline stages, so that the outputs of the dynamic module have the same timing sequence to the static module behind the synchronous boundaries. Such compensation can also be done in the static module. Of course there are some signals not so sensitive to cycle delays, but most probably, the designer has to deal with such restructuring for a correct logic with a high-performance and complex design.

Also, as we have introduced for the DMA logic upgrade from 32-bit PCI Express transaction layer interface to 64-bit, a lot of work and effort have been paid to have a working version quite different than the 32-bit version. These two versions can not be simply merged although they seem the same to the upper-level user. Gluing two versions of designs with generics into one HDL file is considered as actual multiplied work. Such kind of bus width variation leads to a half new version of logic design.

Such stories happen in logic development. From personal estimation of the author, about half of the logic development work is not started from scratch. And estimation from the field of FPGA and ASIC design, 70% of HDL work is verification. [92] So, if we have a tool to reuse the HDL code without need to verify, then about one third of hardware designer's work can be saved.

This chapter is an attempt to have machine-aided reuse of HDL design. Currently we deal only with the DPR issue and the bus variation is not yet covered.

Here in figure 6-2(a) we have a section of VHDL code.

```
process (clk)
begin
  if rising_edge(clk) then
    Op <= Ia and Ib;
    Oq <= not Ia or Ib;
  end if;
end process;
```

(a)

```
Op <= Ia and Ib_delayed;
Oq <= not Ia or Ib_delayed;

process (clk)
begin
  if rising_edge(clk) then
    Ib_delayed <= Ib;
  end if;
end process;
```

(b)

Figure 6-2 An FSM with implicit synchronous pipeline stages

Suppose that, in figure 6-2(a), the input signal I*a* is temporally delayed for one synchronous cycle, and the outputs (O*p*, O*q*) keep the same timing relation with respect to I*b*, how should then the new code look like? There is no explicit cycle for I*a* to shift out. We can manually adapt this code as figure 6-2(b) and the topology is obviously modified. For this simple example, the rewrite correctness is still able to be controlled. However with much larger designs in real projects, such rewrite needs a lot of effort both in implementation and verification.

HDL was not invented for reuse, even also not for synthesis and implementation. As the name tells, it was more likely designed for description, or further, archive and simulation. Its advanced descendants such as System Verilog, System C, etc. have been improved in the system verification direction.

To have a universal reuse, the tool is hard to stay on the layer of HDL. This is why we have half-layer-down software structure, which goes down to the syntax tree level and then comes up back to HDL level.

In conventional code reuse, rigid documentation and surplus parameters are used. Of course there are interesting tips for HDL code reuse. [93]

HDL code reuse should also be different than synthesis. For example, synthesis does not have to reserve the `generics` or `generates` in VHDL. But for reuse, these static reuse control structure should be preserved.

Register retiming technology is a prevailing approach to optimize the increasingly complex FPGA designs timing performance. It is commonly used in the synthesis tools, such as ISE, Precision, etc. The pipeline stages can be chosen moved forwards or backwards. Such technologies, although targeted on timing issues, can be referenced to build another way of use in logic design.

However, these tools deal with the netlists instead of readable HDL codes. The retimed netlists do not return back to the HDL level. Or in other words, such retiming is not readable to the language user. And as we have discussed, DPR practice demands a better control over the HDL level. The user needs usually to rewrite his codes and have an explicit pipeline shifted to the design ports.

IP core generators, such as CoreGen or Peripheral Wizard (EDK) from Xilinx, generate surely the HDL codes, with better reusability. However, they do not start from the HDL codes, but from EDIF or netlist sources.

Ideally, if we can concatenate these two kinds of tools to have an integrated HDL reuse suite, the life will be much easier. Unfortunately, Xilinx and other FPGA vendors do not provide such interface among their software.

Therefore, to have such a reuse tool, which starts from HDL codes and ends with HDL codes, the initiating work has to be done by a new manpower. The purpose is to generate HDL code with expected functionality and good readability from existing HDL design.

In terms of machine-human interface for HDL, a high-level language, we have already HDL → binary, function → HDL. HDL → HDL is the concept of code reuse. Seeing from another point, code regeneration means the machine generates human eligible code automatically. It seems like the machine understands the human demands and try to present the understanding in a familiar way.

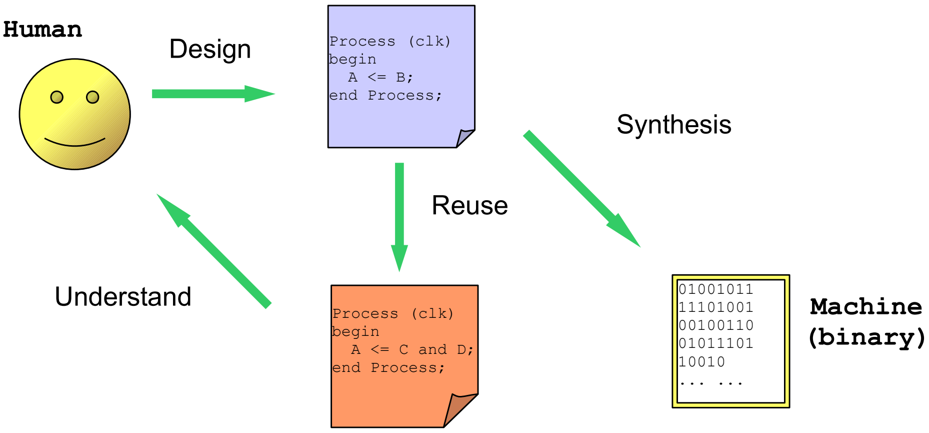


Figure 6-3 Human-machine relationship concerning machine languages

Most directly, the making of such tool can be well referenced from synthesis tools, such as XST, Synplify, ModelSim, and a lot more. There are also some open-source synthesizers/simulators available. Hamburg HDL archive lists most of them. [94]

To have a good control over the expected tree structure, we make AST (**A**bstract **S**yntax **T**ree) of our own. To simplify the routine for the tool to come back to the HDL behavioural layer, existing synthesizers do not fit very well. For example, the `generate` primitives and `generics` should be preserved in the final output HDL design, instead of being replaced with actual values, even when the generates are overlapped.

On the other side, such a tool has looser syntax pressure. The values of some constants do not have to be traced into multiple package files. For example in the next line of assignment,

```
    Buf_WrDin(C_DBUS_AWIDTH-1 downto C_DBUS_AWIDTH/2) <=  DMA_BDA;
```

Here we do not worry about the real information of the constant C_DBUS_AWIDTH, and its original definition does not have to be sought, if it is not locally defined. It can be assumed that it is somewhere else in another file and we simply put it into a pool of picked-up constants and use it if it shows up again. The syntax checking is not a compulsory work of the tool, which is an advantage over making a synthesizer.

In building this kind of tools, there is plenty of existing work on the web to refer, for example, OpenFPGA, ANTLR, LLVM, BlueSpec, and so on. [95] [96] [97] [98] The EDA is in this way being automated. Raphael Njuguna presents major FPGA benchmarks on the web with various HDL source files, which can be used to verify such kind of reuse tools. [99]

6.2 Feasibility

A HDL logic module can be seen as a cause-effect mapping from inputs to outputs, as figure 6-4 shows. A bidirectional signal can be treated as a group of (usually three) inputs and outputs. Every output signal can trace backwards to a number of input signals as its source (unless it is driverless), which builds up a causality tree, like O_{j0} in figure 6-4. The causality tree corresponding to an output port is a general tree structure that represents the logic dependency.

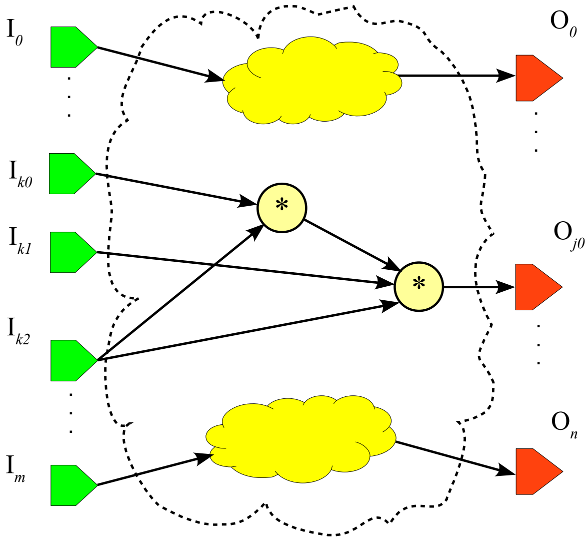


Figure 6-4 General logic and the causality tree

In terms of presence, a tree can be originated or derivative. An originated tree will be explicitly printed in the final rewritten HDL file and a derivative one not. In terms of type, a tree can represent an arithmetic, a logical, a function, or an aggregate. The root index is pointed to the node index in the node/symbol table. For universality, the tree has leaf properties such as slice, aggregate choice, and condition tree information. The trees can be concatenated to build up paths from inputs/drives to outputs/loads. During the path enumerating, if a path has no drive or no load, it will be not calculated, because the rewrite is end-to-end bounded.

The initial purpose of our logic rewrite is to keep the external timing of the dynamic module after synchronous pipelines are attached to it. This involves two parts of change,

a) the input sequence is cycle-delayed (*d*).

b) the output sequence is cycle-advanced (*p*).

In terms of the timing sequence, cycle-delayed means the signal comes a cycle of clock later and cycle-advanced means the signal comes a cycle earlier. Both rewrites are often not so straightforward because the available pipeline stages to be processed may be implicitly scattered in multiple branches of the causality trees. In the DPR practice, we have another alternative, in which both the dynamic module and the static module are rewritten in cycle-delayed-input pattern. We use this approach in this paper, so that the tool needs only to process the input compensation. Of course there are modules, in which no delayed units are found along a causality tree and the cycle delayed rewrite is therefore impossible. Fortunately in most high-speed DPR projects, which need the synchronous pipeline compensation, there are enough pipelines to be shifted in the module. In low-speed DPR projects, the timing closure is not critical along the boundary and therefore, the HDL code can stay untouched.

The automatic HDL reuse tool to fulfil this requirement has “dynamic reuse” property, which is able to make sound modification upon the reused code meanwhile keeping the external behaviour of the whole logic. Correspondingly, the conventional HDL code reuse can be named “static reuse”, in which the code can be reused by slight modification. Such modification might be some parameter (generic) changes, for example, data bus width, switching off ECC, etc. Actually the simplicity of the “static” code reuse is achieved by designer’s extra work to make those parameterized logic run properly. Apart from the static HDL code reuse, the dynamic HDL code reuse discussed here targets on using the existing logic code for a non-copied and non-switched purpose. For example, an SDRAM controller which needs lower latency might probably be generated from an existing SDRAM control logic that has longer latency.

6.2.1 Underlying rules

Basic rules applied to the reuse tool include the extended boolean equations such as,

$$d(\mathrm{A} \bullet \mathrm{B}) = d(\mathrm{A}) \bullet d(\mathrm{B})$$

$$d(\sim\mathrm{A}) = \sim d(\mathrm{A})$$

$$d^{k+1}(\mathrm{A}) = d(d^{k}(\mathrm{A})),\ k > 1$$

Here, A and B are signal names. The operator *d* means a cycle of clock delay is applied to the operand, assuming that all the calculations are in the same clock domain. The tilde ~ denotes a unary inversion (`NOT`). And • represents a binary logic operator like `AND`, `OR`, `XOR`, etc.

A rewrite can be achieved by logical expression transformation. For example, the tree in figure 6-5 is from within a dynamic module, where O_j is the output and I_{k0}, I_{k1}, ... I_{k3} are the inputs of the dynamic module. As discussed before, the inputs should be rewritten to compensate the synchronous pipelines. This original causality tree can be behaviourally expressed as

$O_j = d((d(I_{k0})\ \texttt{OR}\ I_{k1})\ \texttt{OR}\ d(I_{k1}\ \texttt{AND}\ I_{k2}\ \texttt{AND}\ I_{k3}))$.

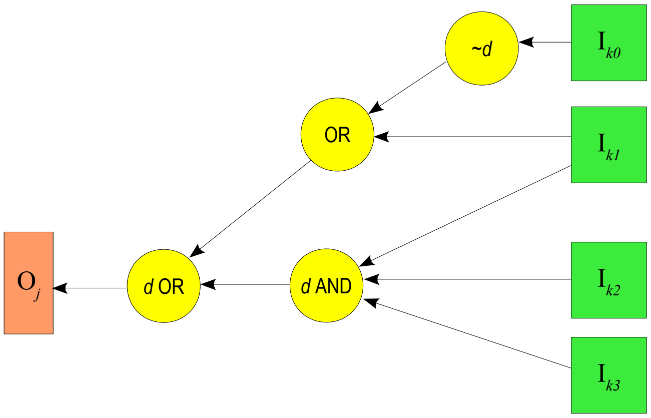


Figure 6-5 Original causality tree

In figure 6-5, *~d* means delayed `NOT`, *d*`AND` means delayed `AND`, and *d*`OR` means delayed `OR`. Similarly we can also have *d*`XOR` for delayed `XOR`, and so forth. To do a cycle-delayed rewrite upon the inputs, the tree in figure 6-5 can be transformed to figure 6-6 and expressed as

$O_j = (d(\ d(I_{k0}))\ \texttt{OR}\ d(I_{k1}))\ \texttt{OR}\ d(d(I_{k1})\ \texttt{AND}\ d(I_{k2})\ \texttt{AND}\ d(I_{k3}))$.

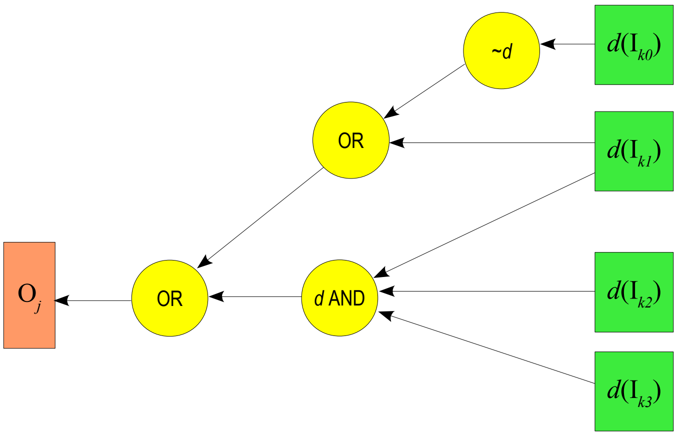


Figure 6-6 Transformed causality tree from figure 6-5

Now the inputs are explicitly synchronous units. The logic is then rebuilt to equip the boundary with synchronous pipelines. This example is quite simplified, just to demonstrate the idea. Rewrite technology is based on expression-matching approach. For example, the cycle-delayed rewrite is equal to the expression replacement

A → *d* (A).

Here the right-side expression is to be replaced by the left-side one. If no match can be found for all branches of any one causality tree, the tool reports the user with impossibility messages. Some other rewrite examples could be

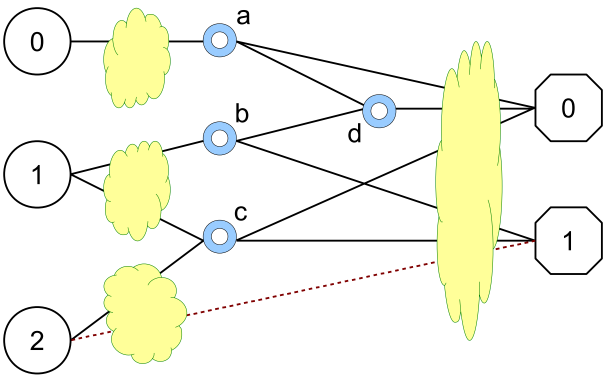
A → A AND *d* (B),

A OR B → A AND *d* (B).

Such replacements, other than the pure cycle-delayed one, are much more complicated to be done manually. They are to be supported by the tool of next version. The reuse tool will try all possible equivalent variations of the target causality trees, search along it, and match the sub-expression with the given one if possible.

6.2.2 Path management

All paths must be firstly enumerated and only then operations upon them are possible. Figure 6-7 shows a simple example of path enumeration.



	O_0	O_1
I_0	2	-
I_1	2	2
I_2	1	**2**

Figure 6-7 path enumeration and association table

The interested units in the design are allowed to be shared among multiple paths, where path coupling happens. If one of the coupled units is used for replacement with respect to one path, the other paths should be carefully treated in the way unexpected changes are avoided. This often involves path decoupling. In current version, because we deal with only the synchronous units (FFs), we duplicate the necessary coupled FFs when path coupling happens. Figure 6-8 shows a case, in which two (actually four) paths couple, where the rings denote pipeline positions and the clouds are asynchronous logic.

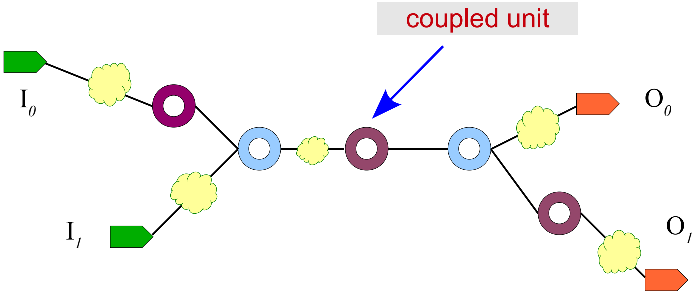


Figure 6-8 Path coupling

6.3 Making of Logro (LOGic RObot)

The logic reuse tool, Logro, works on the AST level. Apart from conventional synthesizers, Logro does not care about what it will look like after implementation.

This tool is actually vendor-independent. But the synthesizable HDL designs in the real world are difficult to be really vendor-independent. Components or parameters specific to the FPGA providers are quite often seen in HDL.

The entire design's functionality is to be essentially preserved. For instances, reset, preset, enable signals should be carefully dealt with, if the design is sensitive to reset timing or similar timing.

Logro is written in C++, compiled with GCC compiler.

Generally, the tool does its work in three steps, HDL → AST → AST ' → HDL ', as the three arrows show in figure 6-9. Accordingly the three steps are numbered as 1, 2 and 3.

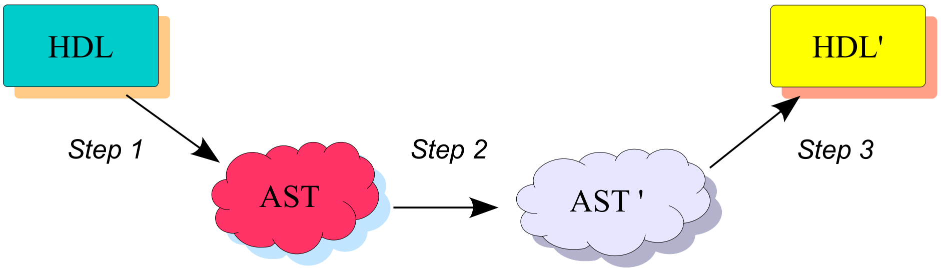


Figure 6-9 Logro flow

Figure 6-10 shows the software components of Logro.

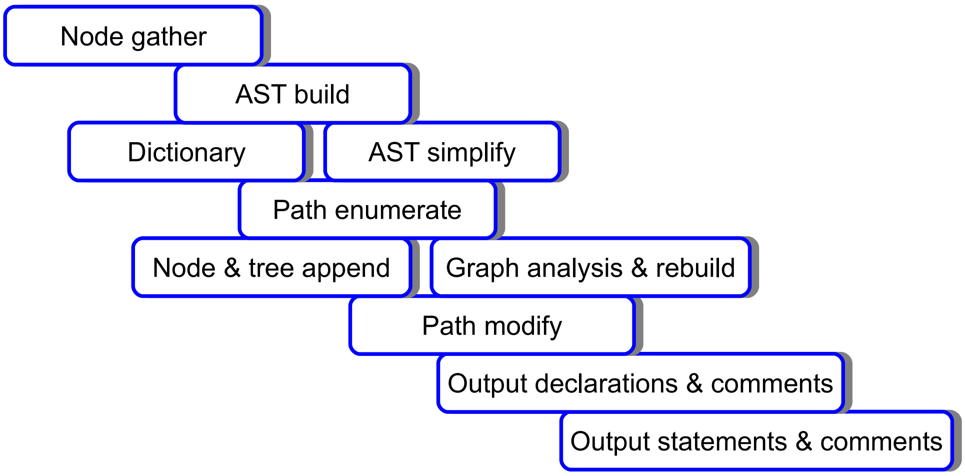


Figure 6-10 Logro composition

6.3.1 Data structure

Arithmetic and logical expressions use the same tree class, and this unifies the tree tracing operations. These two kinds of structures are different in terms of position in HDL code, but they share the common properties of abstract syntax.

The output shifting unifies the problem arisen from DPR boundary processing. The input shifting is a mirrored operation of the output shifting and there is no substantial difference between them.

The tree class is unified to both arithmetic and logic trees because they share most common features, such as root, branches, slices, etc. The tree structure must fit versatile features of HDL. Every assignment to a signal/variable is corresponding to a tree. So the tree should be capable of accommodating simple arithmetic assignments such as

```
usTlp_Req  <= usTlp_Req_i and not FIFO_Reading;
```

as well the logical conditions such as

```
dlm_rec_type <= dlm_rec0  when dlm_rec_valid0='1'
                else dlm_rec1 when dlm_rec_valid1='1'
                else (OTHERS=>'0');
```

or

```
process ( dma_clk, dma_reset) begin
  if dma_reset = '1' then
   DMA_TimeOut_i     <= '0';
 elsif dma_clk'event and dma_clk = '1' then
   if cnt_DMA_TO(C_TOUT_WIDTH-1 downto CBIT_TOUT) = C_TIME_OUT_VALUE then
    DMA_TimeOut_i  <= '1';
   else
    DMA_TimeOut_i  <= DMA_TimeOut_i;
   end if;
 end if;
end process;
```

Out of this consideration, an assignment to a signal may be mapped to multiple trees chained up. It should be able to be traced, forwards and backwards. A class function is made to trace all the loads and all the drives of one signal. A generalized tree structure is illustrated in figure 6-11.

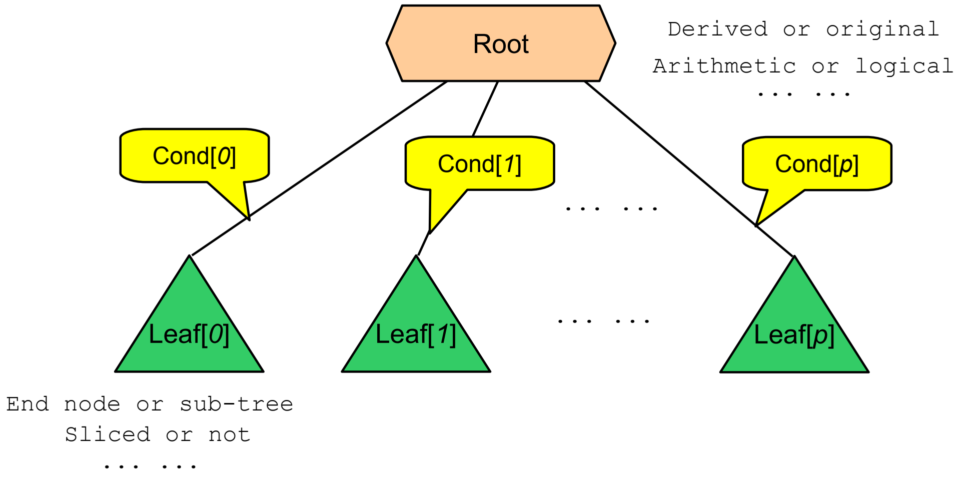


Figure 6-11 Generalized tree structure in Logro

6.3.2 Running steps

Step 1 Tree build

In the declaration part, Logro collects the node information (signal, constant, type, …) and builds up a symbol pool. In the statement part, Logro analyses the expressions (assignment, process, aggregate, …) and builds them into trees. The end leaves are the static nodes. Because the synthesis correctness is a prerequisite for the source HDL, lexicon and syntax do not have to be checked.

`GENERATE` and `BLOCK` information is tagged to corresponding trees.

Between this first step and the next step, some preliminary simplification is executed, e.g. doubled negate will vaporize.

Some signals assignment is implicit in the code, e.g.

```
A <= (1=>B_Sig, others=>'0');
```

And such implicitness is well handled so that no divergence occurs from the source design after rewriting.

Function recognition might need searching the library package, because literally we know the content in the brackets of the right-side

```
A <= S(C_bit downto C_bit-3);
```

is a slice; and

```
B <= F(In1, Var2);
```

is a function assignment, assuming every aggregate assignment must have choices and "=>" symbols. However, we cannot make sure whether `Fr()` in

```
E <= Fr(ID);
```

is a function or a slice, if it is not defined locally. If such situation happens, the software will ask the user whether the sceptical statement is a function. This is not large work for the user because the number of user-defined functions is usually small and the user is supposed to know them well. Most general functions from IEEE libraries, such as `CONV_STD_LOGIC_VECTOR()`, are already included in the data base of Logro.

A restriction is applied to the function body, which should not contain cycle delays. In this way, every function is treated as asynchronous assignments. This restriction is plausible because most functions are not for pipelined calculations.

Loops in the causality tree should be avoided because they disable the validity of some tree processing algorithms. A causality chain (driving relationship from an input to an output), is sometimes discontinued by a black-box and the traversing of this chain by seeking the sub-expression should also go around the black-box.

Feed-back FF is a potential source of looped sections in the causality tree. Expressions such as $A = d(A)$, $A = d(A + C)$ should be marked during the logic causality scanning, so that the tree traverse in next steps does not trace back to form loops.

Black-box (sub-module instance, IP core, etc.) isolates the inputs and outputs through it. Tracing into a black-box needs hierarchical functions supported by the reuse tool. In the scope of a single HDL module, the black-box can be simply taken as a solid "box" and the logic relationship inside is supposed unknown to the tool. If any signal around the black-box needs to be analysed or rewritten, the reuse tool can be applied to that black-box sub-module.

Step 2 Path transform

Paths are enumerated from inputs to outputs. Association matrix is filled. Then the trees are rebuilt according to the defined purpose.

This part is the most complicated because it involves node appending and path duplicating whereas the causality must be preserved. Path processing and rebuilding takes most CPU time for the entire program. Also this part consumes most portion of the software development. And of course this part leaves much to optimize, for example the path enumerating.

Solution finding may probably experience equivalent transformation of paths as well as necessary node duplication. For sake of an easier solution finding, minimized node duplication and minimized tree appending is the strategy in path rebuilding, because the execution time is strongly dependent upon the tree number. As the case shown in figure 6-8, where paths shall be decoupled by tree appending, the load number and drive number are calculated and the smaller one suggests the tree appending direction.

Theoretically we can isolate all paths according to outputs so that every output is expressed as a function of inputs. Then the solution is much easier to find. However, this method takes too much extra resource. Roughly estimation can be about Q times more resource is needed, where Q stands for the output port number.

A solution is a combination of pipeline stages that meets the requirement of covering every path once and only once. A most intuitive method is to try all the combinations of pipeline stages in the path space and find out which combination(s) meet(s) the purpose. However, for a path space with N stages, the complexity level will be $O(2^N)$, unacceptably large. As we know,

$$\binom{N}{1} + \binom{N}{2} + \cdots + \binom{N}{N} = 2^N$$

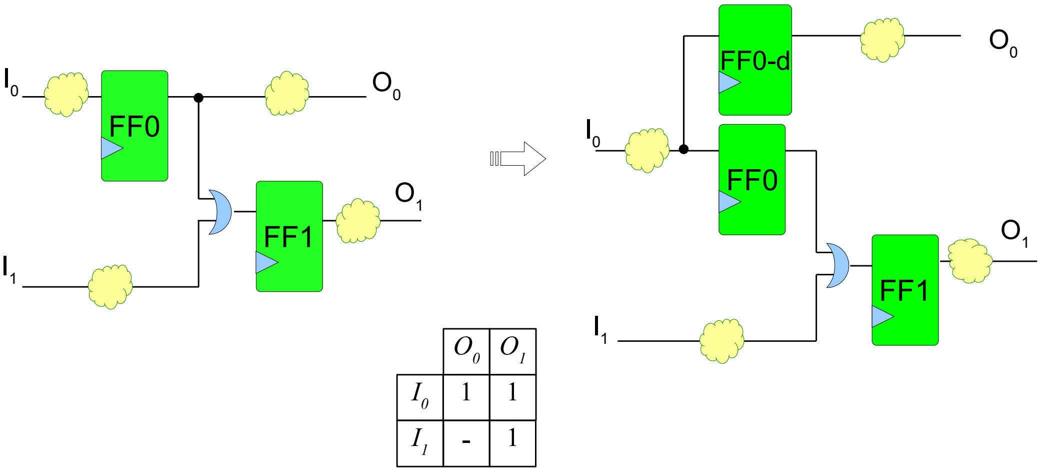
This was also testified in our first version of solution finding algorithm, which had terribly low performance for $N>30$.

For a simplified algorithm and full coverage of all combinations, we found out that in the real HDL designs there are a number of paths with only one pipeline stage. This observation leads to a work-around for the solution finding, which excludes these single-pipeline paths out of calculation and makes them as the fixed part of the solution, because the unique pipelines in them are to be covered in any case. With this work-around, the original combination enumeration approach is still used and the execution time is immensely reduced. All of our HDL designs can be processed within 10 minutes on a 2.67GHz processor, 6GB DDR3 memory PC.

For next version, heuristic searching algorithms will be tried to reduce the complexity. Another possible improvement can take advantage of path grouping, and is supposed to decrease the execution time and increase the efficiency.

If all combinations have been tested and no solution is found, path decoupling will be made to change the trees correlation whereas the input-output relationship is preserved. The rebuild of trees will increase the number of total trees. Then a new iteration is done to the newly built trees. This procedure is executed until a solution is found. The equivalent transform of trees is the most important task for the tool, because it enables the original design to be processed in a new structure, especially when none of the pipeline combination is a solution. This transform is the central idea of a dynamic reuse of HDL code.

An example in solution finding is shown in figure 6-12, where path coupling happens and decoupling is performed from the left figure to the right one. Important is that, the input-output logic behaviour must be the same before and after the transform.



	O_0	O_1
I_0	1	1
I_1	-	1

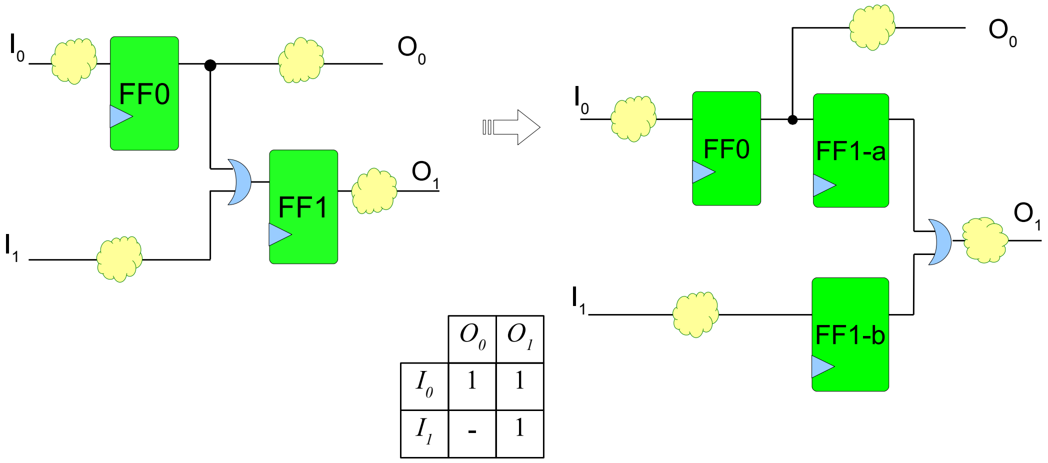
(a)

(b)

Figure 6-12 Path coupling and decoupling (load-bifurcation)

In figure 6-12 (a), neither FF0 nor FF1 can fulfil the request alone, and therefore, the paths have to be regulated as in figure 6-12(b), where the original FF0 is duplicated. In figure 6-12(b), {FF1, FF0-d} can be a solution.

Similarly we can also transform the example in figure 6-12 in another way, as figure 6-13 shows. Here we duplicate the FF1 instead of FF0 and therefore, {FF0, FF1-b} can be a solution. The transform in figure 6-12 is called load-bifurcation, which duplicates the load branch. Accordingly, figure 6-13 is called drive-bifurcation, which duplicates the drive.



	O_0	O_1
I_0	1	1
I_1	-	1

(a) (b)

Figure 6-13 Path coupling and drive-bifurcation

If the coupled pipeline has only one load and only one drive, the bifurcation cannot be directly applied, as the shared unit FF3 in figure 6-14. Then the program will shift that stage of pipeline to the input or output, until multiple loads or multiple drives are available. Always single load and single drive should not be possible, otherwise that path would not have been coupled with other path(s). Pipeline shifting does not change the total number of cycle delays along a given path.

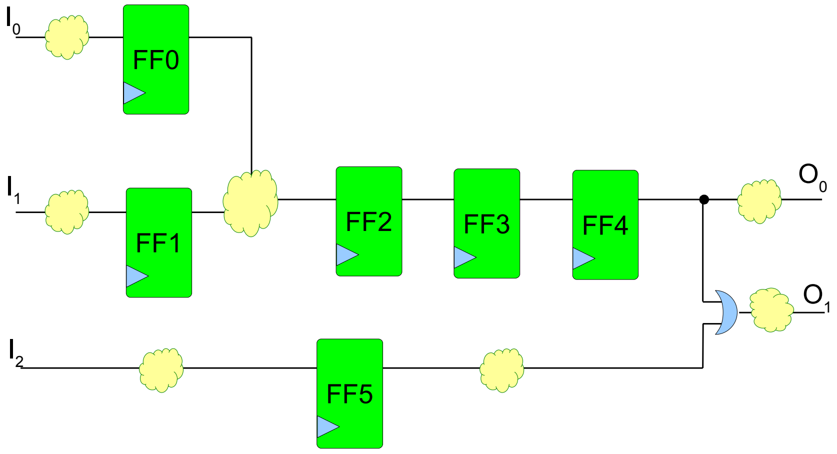


Figure 6-14 Shared unit with single ends

A simplified flow diagram for the path decoupling is summarized in figure 6-15.

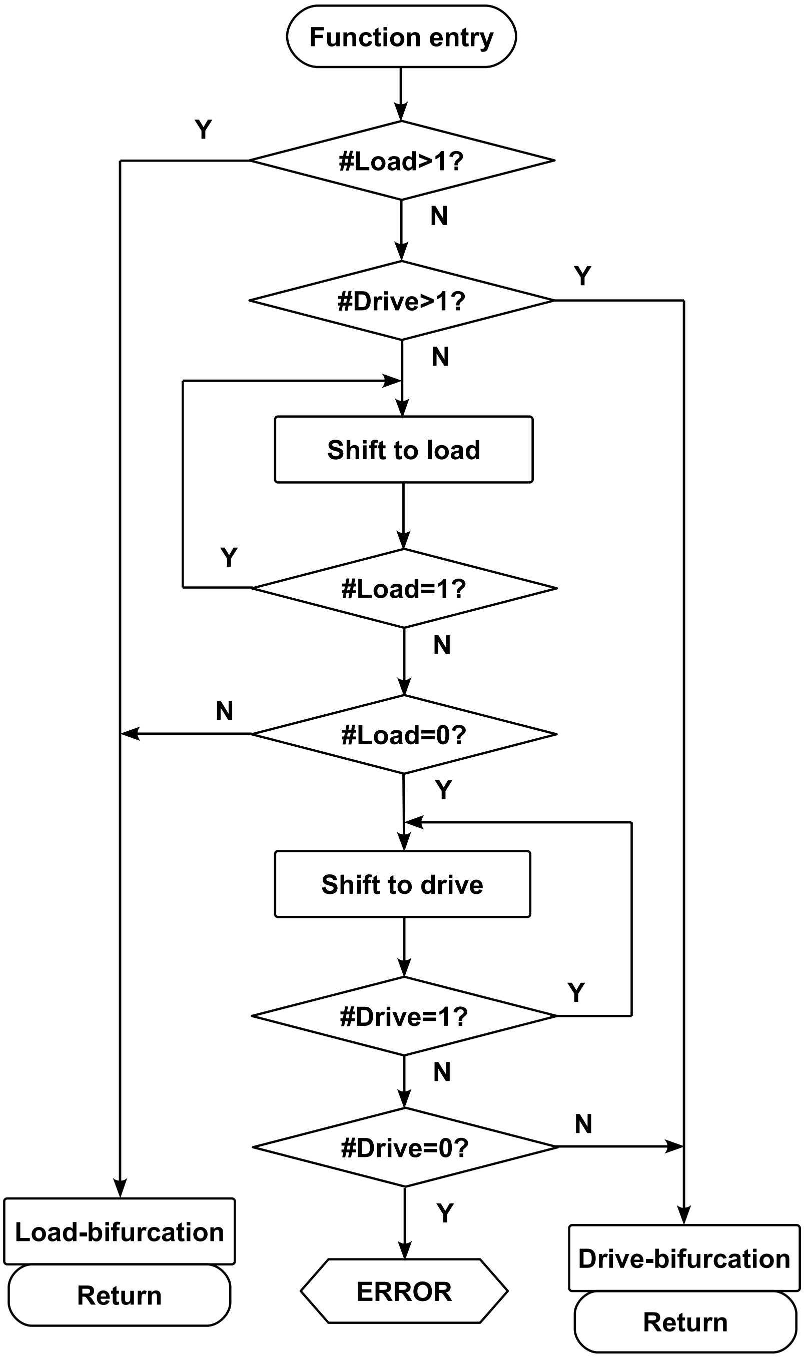


Figure 6-15 Path decoupling flow diagram

This part has also switch for the user to select whether zero-rewrite is made, which bypass the path rebuild step and provides method to debug the step 1 and step 3.

Step 3 Backward-write

Finally, the modified trees are translated back to HDL, which is eligible to the user. The entity is almost the same as the original one. The declaration part may contain more signals that are appended. And the statement part will see much difference than the original one. In the output process, comments are added to those appended signals and duplicated processes.

User comment is seen as an indispensable part of design. The comments in the original text will be tried to preserve. If preserve is impossible, the reason is printed in an area of the generated HDL code. During rewrite, the comment should not be simply deserted because it contains reuse information and development chronicle and ideas. Original user comments are tried to stay where they were. If the original comment cannot find a suitable place in the new file to anchor, it has to be removed. This might happen when a comment appears inside a process with multiple signals. Since the signals within one process are decoupled for the current version, the comment inside process is hard to decide, to which signal it belongs.

New signals in declaration parts are extra commented how they come into being and the rewritten places are additionally notified with comments. Therefore the rewritten HDL is generally larger than the original one in terms of file size.

6.4 Test and verification

Since the second step takes much programming effort, our development sequence was step 1 → step 3 → step 2. The reason lies in the dependency of step 2 upon steps 1 and 3. Step 2 cannot be separately verified. The input of step 1 is the HDL file and the output of step 3 is the generated HDL file. Without the modification on the trees, the output of step 1 can directly feed step 3. Thus, we anticipate two synthesize-equivalent HDL files, as explained in figure 6-16.

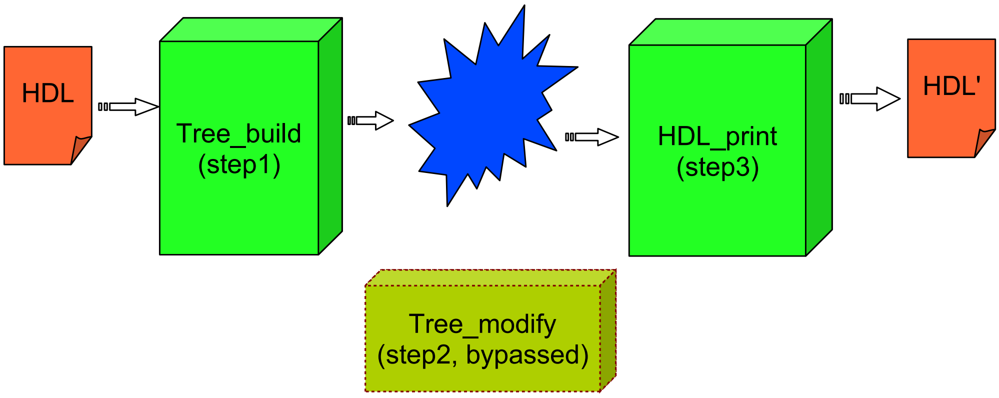


Figure 6-16 Initial phase of the tool development without step 2

At the most beginning the Logro development, a benchmark VHDL file was written. We have tried to integrate as many elements of the VHDL as possible in the demo. And this synthesizable demo VHDL design is going to grow up as a test bench for such reuse tools.

We have also tried most VHDL design files in ABB project and MPRACE2 project, as well as some Xilinx EDK core design.

For instance, the iterative rewrite is used to test the convergence. After N times iterative rewrite, the generated files should have similar synthesis report. Here N is the iteration times.

The basic idea to verify the rewritten code is to replace the original one in the simulation environment. If the outputs (o^0 and o^*) are the same, the rewritten logic can be accepted, as figure 6-17 shows.

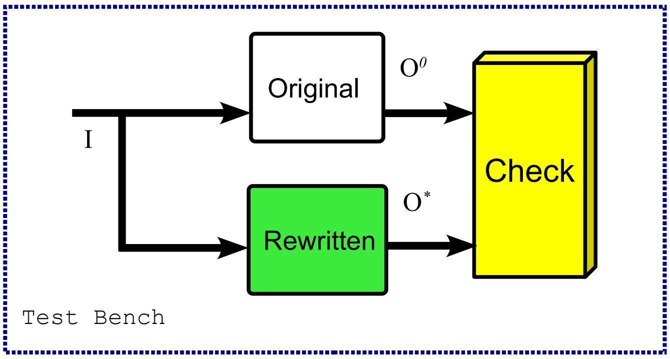


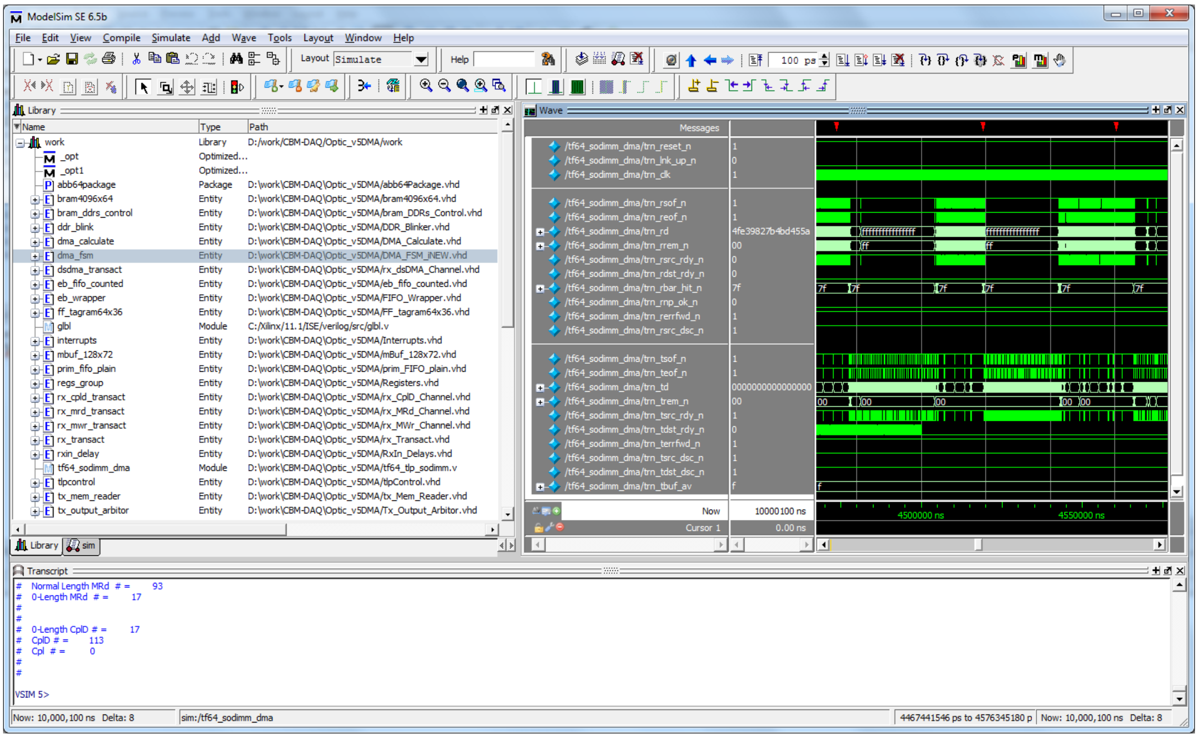
Figure 6-17 Verification scheme

In the DMA kernel module DMA_FSM, there are enough coupled paths and several inputs with implicit pipelines. We replace the original file (`DMA_FSM.vhd`, 979 lines) with the rewritten version (`DMA_FSM_iNew.vhd`, 1211 lines) in the simulation discussed in chapter 3. And after 30 ms of simulation time, the simulation does not stop, as figure 6-18. This can be a proof, that the rewrite is successful in simulation.

This rewritten design file, `DMA_FSM_iNew.vhd`, is also used in the ML605 DPR experiment, decoupling the boundary for the original file `DMA_FSM.vhd`. The test result shows the rewrite is successful, all tests being correctly repeated. The performance tests have the same results because the rewrite does not change the pipeline relationship inside the logic, as figure 4-10.

6.5 Suggestion to DPR vendors

Logro is proven concise and useful. However, its DPR module functions, e.g. tree building, write-back, etc., are commercially mature in EDA giants, such as Mentor, Synopsys, Xilinx, Altera, 0-In, and so on. To integrate the DPR function into their software suites should not be difficult for them, especially for Xilinx. In PlanAhead, such decoupling can be realized even without returning back to HDL. Basic retiming operation will help a lot in a DPR project and the user does not have to rearrange the registers or to repeat the error-prone work themselves.



Name	Type	Path
work	Library	D:/work/CBM-DAQ/Optic_v5DMA/work
_opt	Optimized...	
_opt1	Optimized...	
abb64package	Package	D:\work\CBM-DAQ\Optic_v5DMA/abb64Package.vhd
bram4096x64	Entity	D:\work\CBM-DAQ\Optic_v5DMA/bram4096x64.vhd
bram_ddrs_control	Entity	D:\work\CBM-DAQ\Optic_v5DMA/bram_DDRs_Control.vhd
ddr_blink	Entity	D:\work\CBM-DAQ\Optic_v5DMA/DDR_Blinker.vhd
dma_calculate	Entity	D:\work\CBM-DAQ\Optic_v5DMA/DMA_Calculate.vhd
dma_fsm	Entity	D:\work\CBM-DAQ\Optic_v5DMA/DMA_FSM_iNEW.vhd
dsdma_transact	Entity	D:\work\CBM-DAQ\Optic_v5DMA/rx_dsDMA_Channel.vhd
eb_fifo_counted	Entity	D:\work\CBM-DAQ\Optic_v5DMA/eb_fifo_counted.vhd
eb_wrapper	Entity	D:\work\CBM-DAQ\Optic_v5DMA/FIFO_Wrapper.vhd
ff_tagram64x36	Entity	D:\work\CBM-DAQ\Optic_v5DMA/FF_tagram64x36.vhd
glbl	Module	C:/Xilinx/11.1/ISE/verilog/src/glbl.v
interrupts	Entity	D:\work\CBM-DAQ\Optic_v5DMA/Interrupts.vhd
mbuf_128x72	Entity	D:\work\CBM-DAQ\Optic_v5DMA/mBuf_128x72.vhd
prim_fifo_plain	Entity	D:\work\CBM-DAQ\Optic_v5DMA/prim_FIFO_plain.vhd
regs_group	Entity	D:\work\CBM-DAQ\Optic_v5DMA/Registers.vhd
rx_cpld_transact	Entity	D:\work\CBM-DAQ\Optic_v5DMA/rx_CplD_Channel.vhd
rx_mrd_transact	Entity	D:\work\CBM-DAQ\Optic_v5DMA/rx_MRd_Channel.vhd
rx_mwr_transact	Entity	D:\work\CBM-DAQ\Optic_v5DMA/rx_MWr_Channel.vhd
rx_transact	Entity	D:\work\CBM-DAQ\Optic_v5DMA/rx_Transact.vhd
rxin_delay	Entity	D:\work\CBM-DAQ\Optic_v5DMA/RxIn_Delays.vhd
tf64_sodimm_dma	Module	D:\work\CBM-DAQ\Optic_v5DMA/tf64_tlp_sodimm.v
tlpcontrol	Entity	D:\work\CBM-DAQ\Optic_v5DMA/tlpControl.vhd
tx_mem_reader	Entity	D:\work\CBM-DAQ\Optic_v5DMA/tx_Mem_Reader.vhd
tx_output_arbitor	Entity	D:\work\CBM-DAQ\Optic_v5DMA/Tx_Output_Arbitor.vhd

Messages	
/tf64_sodimm_dma/trn_reset_n	1
/tf64_sodimm_dma/trn_lnk_up_n	0
/tf64_sodimm_dma/trn_clk	1
/tf64_sodimm_dma/trn_rsof_n	1
/tf64_sodimm_dma/trn_reof_n	1
/tf64_sodimm_dma/trn_rd	4fe39827b4bd455a
/tf64_sodimm_dma/trn_rrem_n	00
/tf64_sodimm_dma/trn_rsrc_rdy_n	0
/tf64_sodimm_dma/trn_rdst_rdy_n	0
/tf64_sodimm_dma/trn_rbar_hit_n	7f
/tf64_sodimm_dma/trn_rnp_ok_n	0
/tf64_sodimm_dma/trn_rerrfwd_n	1
/tf64_sodimm_dma/trn_rsrc_dsc_n	1
/tf64_sodimm_dma/trn_tsof_n	1
/tf64_sodimm_dma/trn_teof_n	1
/tf64_sodimm_dma/trn_td	0000000000000000
/tf64_sodimm_dma/trn_trem_n	00
/tf64_sodimm_dma/trn_tsrc_rdy_n	1
/tf64_sodimm_dma/trn_tdst_rdy_n	0
/tf64_sodimm_dma/trn_terrfwd_n	1
/tf64_sodimm_dma/trn_tsrc_dsc_n	1
/tf64_sodimm_dma/trn_tdst_dsc_n	1
/tf64_sodimm_dma/trn_tbuf_av	f
Now	10000100 ns
Cursor 1	0.00 ns

```
#   Normal Length MRd  # =       93
#   0-Length MRd  # =       17
#
#
#   0-Length CplD # =       17
#   CplD # =      113
#   Cpl  # =       0
#
#

VSIM 5>
```

Now: 10,000,100 ns Delta: 8 | sim:/tf64_sodimm_dma | 4467441546 ps to 4576345180 p | Now: 10,000,100 ns Delta: 8

Figure 6-18 A DPR-replacement simulation in the PCI Express DMA project

If the FPGA vendors integrate the boundary decoupling into their tools, the logic reuse will not lose its post to stand. It sees miscellaneous applications in the EDA world. For example, HDL code optimization or normalization is purely HDL-level operations and they are sure to benefit from the dynamic HDL reuse concept.

6.6 Application perspective

Besides the DPR boundary decoupling example, Logro is promising in following application cases,

– HDL optimization/simplification: redundancy in the design can be removed if a component drives no logic;

– FSM rebuild: a minor timing change in the sensitivity list of a FSM may require a totally new design and Logro will be improved to deal with such situations;

– Paths/Process decoupling with the module: decoupling can be made to divide the original design into multiple isolated modules;

– HDL translator: as a by-product of Logro, it will be possible to rewrite a VHDL design in Verilog HDL or vice versa.

Since we have the tree level structure (directed acyclic graph) of the original design, we have the freedom to make much more modification over it and then the backward-write translate the modified graph to HDL code, as figure 6-16 implies. Such modification can be merge, divide, simplify, balance, … It might also be possible that a series of modified designs is generated for simulation or optimization.

Figure 6-19 shows the concept of design convergence outside the bench test. There are two entry points for such development iteration, either from design or from simulation. And we can see there should be no ending for such iteration, which means, the design process is in a dynamic state and can never be proved bug-free. The traditional logic project starts from the design. The input-output requirement is expressed in HDL and then, the synthesizable design is delivered to simulation. The simulation environment is built previously or parallel to the design and it is highly recommended that the design and the simulation environment are generated by two separate groups to avoid the simulation blind spots. If the simulation shows the design has the proposed behaviour, the logic can be released to the bench test; otherwise, the bug (unexpected behaviour) is traced in the design and fixes are correspondingly made to provide a new design for simulation. Same as the design, the simulation can also contain bugs, so when errors occur during the simulation, it must be sure they are not out of the simulation environment itself.

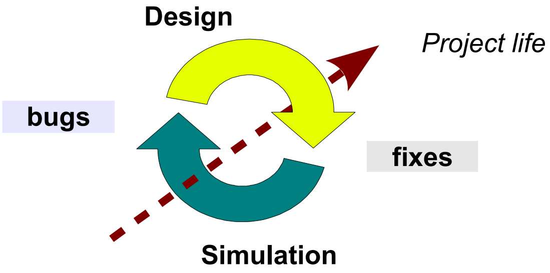


Figure 6-19 Design-simulation iteration

There are simple designs that do not need simulation and therefore, are directly released to bench tests after HDL coding. In such cases, the simulation is actually done in the designer's mind and no explicit simulating tools are needed.

Since we have the infinite verification environment presented in chapter 3, the design-simulation iteration in figure 6-19 can be improved with logic reuse tool, such as Logro. The bugs found in simulation are traced to the design HDL and Logro makes the modification as well as the new version of design. In this way, the project can be directly started from simulation instead of from design and the development procedure is half-automated. The logic developer first sets up the simulation environment, which should allow appending. The simulation defines the expected output after certain input. If the output has variance to the defined output pattern, the simulation should be able to indicate quantitatively the fault module or the fault pipeline, which the bug tracer can direct the logic rewrite tool to modify.

The bug tracer is not yet finished. It has to correlate the output errors to the design positions.

6.7 Conclusion

Although there is much to improve in Logro development, such as better efficiency than O(2^N) complexity, hierarchical design support, and so on, the automatic reuse of HDL design has been proved feasible and plausible by Logro. The boundary rewriting for the DPR projects is successfully done by Logro v1.0. The infrastructure of the tool allows further improvement on more versatile functions.

Conventional IP core is difficult to change an internal part if there is no corresponding parameters/generics preserved for the user. Logro's concept is to go deep into the original HDL code and to make arbitrary modification to any part of it.

If arbitrary modification to the original HDL design can be realized, a new hardware development pattern is then possible, which alleviates the engineer's effort and promotes the reliability of the generated design.

Chapter 7 Summary and outlook

7.1 The Active Buffer

The detector system in high-energy physics is growing with bigger DAQ portion. DAQ is expected to be more powerful and more intelligent to alleviate much work of the physicists.

The DAQ of CBM experiment is featured with large data density and flexible triggering strategy.

Weak triggering DAQ is a trend to produce opportunities both for FEE and for physicists. Larger amount of data thereafter demands larger buffer. The Active Buffer is promising to upgrade performance both in size and processing capability. It is built upon commercial SDRAM modules and therefore has great potential to be enlarged according to the Moore's Law. The building of the Active Buffer has also general meaning for similar kinds of large-size buffer design. Stand-alone ASIC wrapper chips to transform the SDRAM module into FIFO are technically feasible.

The self-test, mixing pseudo-random inputs and manual interference, is designed and provides deeper verification to the large-size buffer. The mixed test has a better coverage to the buffer design and the behavioural correctness is proved in CBM DAQ beam tests.

Provided that the bus width of the buffer kernel is 64-bit, without additional bits for framing or marking, the event building approach takes advantage of pattern insertion workaround. It is worth for the next upgrading of this buffer to have extra bits parallel to the payload data bus width, which will greatly simplify the logic as well as increase the performance in event building assistance work, such as epoch marker indexing and SYNC marker identifying.

The kernel for the Active Buffer is the large-size buffer made of a 512MB DDR2 SDRAM module. Scaling it up to the size of contemporary SDRAM modules, such as 1 GB, 2 GB or 4 GB, is fairly easy. This buffer complies very well with the standard FIFO port protocol, supporting concurrent write and read and providing 8 Gbps bandwidth per port. It is ideal for large and irregular data rate and proves to be efficient in the CBM DAQ system.

7.2 DMA over PCI Express

DMA engine is developed over PCI Express bus to provide a fast and reliable data path from the data source to the host nodes. A DMA transaction is usually chained up with descriptors, which provides flexibility in USER memory mode. DMA is categorized into passive (slave) and active (master) modes. The thesis work is focused on the passive (scatter-gather) DMA development and the master DMA is not yet completed.

A practical DMA design should support scatter-gather mode for the USER memory applications. And due to the properties of the CBM DAQ system, our DMA design is adapted to its high data flow density and to the large-size DDR-FIFO structure.

The verification approach is important in the logic development of a growing and varying design. It helps to avoid new bugs in the appending or optimization upon the original HDL code. The DMA engine is running on the transaction layer of PCI Express, so the verification tries to emulate all related behaviours of the PCI Express core on the transaction layer. The verification uses infinite loop mode with randomized input stimulus so that the behavioural coverage can be maximized. It is convinced that this verification environment has good performance in finding hidden logic bugs. To determine the ending of the infinite loop, TLV approach is initiated and needs to be completed.

The DMA design over PCI Express has been expanded to several similar projects. Furthermore, the DMA engine is towards to have more functions inside the data stream, for example, Epoch Marker finding and SYNC Marker finding. Such upgrading leads to master-mode DMA scenario. Master-mode DMA can further relieve the host CPU load.

With PCI Express Gen1 ×4 design, we have achieved over 500 MB/s for DMA read and over 700 MB/s for DMA write on the AVNET Virtex5 board (ABB2). Based on the simulation environment in chapter 3, the performance is supposed to be accordingly improved by squeezing pipeline stages of the DMA read channel. The same DMA engine is also implemented in the Virtex6 FPGA on ML605 board and the performance values are comparable to the Virtex5 tests.

7.3 DPR

DPR is not only popular but also useful in the FPGA research. It benefits from booming FPGA technologies and software support. The barrier between hardware and software implementation is going to be removed in this way and hardware-software co-design is becoming a reality.

We apply DPR technology to the DMA engine overlapping and to avoid PCI Express system reboot after reconfiguration in the first DPR experiment on Virtex4 FPGA. Two dynamic modules with six versions per module are implemented. The boundary between static and dynamic is isolated by synchronous bus-macros under PlanAhead 9.2. It presents good perspective of DPR application in HEP, especially for the future resource-critical FPGA development in the CBM experiment.

Under updated version of PlanAhead (v12.4), we have tried similar DPR projects with the Virtex6 FPGA on ML605 board and with the Virtex5 FPGA on the AVNET board. Bus-macros are deprecated and boundary decoupling is highly recommended between the dynamic and the static modules by Xilinx. Logic rewrite is still needed. ICAP is used as the reconfiguration port and the dynamic reconfiguration is realized via the PCI Express bus. An ICAP operation package with PIO accessing mode is added to the MPRACE library. The result is encouraging with flexibility and simplicity as well as data integrity and system stability. A preliminary test shows over 6 MB/s reconfiguration speed and it is possible to be further accelerated.

Task alternating in FPGA just like software module loading and unloading is therefore possible with the ICAP-PCIe solution and this can be generalized to a universal computing acceleration instrument.

7.4 HDL Reuse

In the beginning of the DPR project, HDL code rewrite took unexpectedly long time. Concerning similar request from the hardware design, the HDL reuse tool, *Logro*, is developed.

Traditional HDL reuse concentrates on the IP core verification and has produced plenty of successful macros and blocks. However, it demands developer's extra effort and usually it is impossible for a reuser to go into the core design and make customized modifications. Compared with the conventional “static” reuse, the “dynamic” reuse concept inside *Logro* keeps on the HDL layer and promises vast varieties of applications, such as design code optimization or functional decoupling inside a module.

Logro is developed to provide the reuser with more confidence in dealing with the existed HDL codes. It is clearly divided into three steps of processing the original codes, HDL → AST, AST → AST' and AST' → HDL. The middle step is much more flexible to integrate more versatile functions.

Logro can even affect the traditional design flow of HDL projects. If the reuse practice is transparent to the designer, the development of logic module can be simplified, because the design iteration can be done in an automatic way.

Logro is verified in the ML605 DPR project, where HDL codes of the dynamic modules are generated by *Logro* and the DMA correctness test and performance test are successfully completed.

Appendix A.1

ABB1 FPGA implementation report

```
Release 11.4 Map L.68 (nt64)

Xilinx Mapping Report File for Design 'pcieDMA'

Design Information
------------------
Command Line   : map -ise PCIeDMA.ise -intstyle ise -p xc4vfx60-ff672-11 -timing
-logic_opt on -ol high -xe n -t 1 -register_duplication off -global_opt speed
-retiming on -equivalent_register_removal on -cm balanced -ir off -pr b -power
off -o pcieDMA_map.ncd pcieDMA.ngd pcieDMA.pcf
Target Device  : xc4vfx60
Target Package : ff672
Target Speed   : -11
Stepping Level : 0
Mapper Version : virtex4 -- $Revision: 1.51.18.1 $
Mapped Date    : Tue Mar 23 11:48:11 2010

Design Summary
--------------
Number of errors:      0
Number of warnings:   33
Logic Utilization:
  Total Number Slice Registers:      12,404 out of  50,560   24%
    Number used as Flip Flops:       12,403
    Number used as Latches:               1
  Number of 4 input LUTs:            15,718 out of  50,560   31%
Logic Distribution:
  Number of occupied Slices:         13,198 out of  25,280   52%
    Number of Slices containing only related logic:  13,198 out of  13,198 100%
    Number of Slices containing unrelated logic:          0 out of  13,198   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      16,643 out of  50,560   32%
    Number used as logic:            13,971
    Number used as a route-thru:        925
    Number used for Dual Port RAMs:     200
      (Two LUTs used per Dual Port RAM)
    Number used as Shift registers:   1,547

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IPADs:                26 out of      80   32%
  Number of bonded OPADs:                24 out of      32   75%
  Number of bonded IOBs:                  2 out of     352    1%
  Number of BUFG/BUFGCTRLs:               6 out of      32   18%
    Number used as BUFGs:                 5
    Number used as BUFGCTRLs:             1
  Number of FIFO16/RAMB16s:              66 out of     232   28%
    Number used as RAMB16s:              66
  Number of DSP48s:                       2 out of     128    1%
  Number of DCM_ADVs:                       1 out of      12    8%
  Number of BSCAN_VIRTEX4s:                 1 out of       4   25%
  Number of ICAP_VIRTEX4s:                  1 out of       2   50%
  Number of GT11s:                         12 out of      12  100%
  Number of GT11CLKs:                       1 out of       8   12%

  Number of RPM macros:           12
Average Fanout of Non-Clock Nets:               3.37

Peak Memory Usage:  1089 MB
Total REAL time to MAP completion:  10 mins 56 secs
Total CPU time to MAP completion:   10 mins 52 secs
```

Appendix A.2

MPRACE2 Bridge FPGA implementation report

```
Release 11.5 Map L.70 (nt64)

Xilinx Mapping Report File for Design 'pcieDMA'

Design Information
------------------
Command Line   : map -ise Bridge.ise -intstyle ise -p xc4vfx20-ff672-11
-timing -logic_opt on -ol high -xe n -t 1 -register_duplication off -global_opt
speed -retiming on -equivalent_register_removal off -cm balanced -ir off
-ignore_keep_hierarchy -pr b -power off -o pcieDMA_map.ncd pcieDMA.ngd
pcieDMA.pcf
Target Device  : xc4vfx20
Target Package : ff672
Target Speed   : -11
Stepping Level : 0
Mapper Version : virtex4 -- $Revision: 1.51.18.1 $
Mapped Date    : Tue Oct 05 14:49:17 2010

Design Summary
--------------
Number of errors:      0
Number of warnings:   23
Logic Utilization:
  Number of Slice Flip Flops:        12,322 out of  17,088   72%
  Number of 4 input LUTs:            15,244 out of  17,088   89%
Logic Distribution:
  Number of occupied Slices:          8,534 out of   8,544   99%
    Number of Slices containing only related logic:   8,534 out of   8,534 100%
    Number of Slices containing unrelated logic:          0 out of   8,534   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:      15,942 out of  17,088   93%
    Number used as logic:            13,810
    Number used as a route-thru:        698
    Number used for Dual Port RAMs:     336
      (Two LUTs used per Dual Port RAM)
    Number used as Shift registers:   1,098

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IPADs:                20 out of      24   83%
  Number of bonded OPADs:                16 out of      16  100%
  Number of bonded IOBs:                  4 out of     320    1%
    IOB Flip Flops:                       2
  Number of BUFG/BUFGCTRLs:               7 out of      32   21%
    Number used as BUFGs:                 6
    Number used as BUFGCTRLs:             1
  Number of FIFO16/RAMB16s:              19 out of      68   27%
    Number used as FIFO16s:               2
    Number used as RAMB16s:              17
  Number of DSP48s:                       2 out of      32    6%
  Number of DCM_ADVs:                     1 out of       4   25%
  Number of GT11s:                        8 out of       8  100%
  Number of GT11CLKs:                     2 out of       4   50%

Average Fanout of Non-Clock Nets:                3.33

Peak Memory Usage:  1226 MB
Total REAL time to MAP completion:  16 mins 28 secs
Total CPU time to MAP completion:   16 mins 25 secs
```

Appendix A.3

DDR FIFO self-test (AVNET board) FPGA implementation report

```
Release 11.5 Map L.70 (nt64)

Xilinx Mapping Report File for Design 'SAT_DDR2FIFO'

Design Information
------------------
Command Line   : map -ise SAT.ise -intstyle ise -p xc5vlx110t-ff1136-1 -w
-logic_opt on -ol high -xe n -t 1 -register_duplication off -global_opt off -mt
off -cm balanced -ir off -pr off -ignore_keep_hierarchy -lc off -power off -o
SAT_DDR2FIFO_map.ncd SAT_DDR2FIFO.ngd SAT_DDR2FIFO.pcf
Target Device  : xc5vlx110t
Target Package : ff1136
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.51.18.1 $
Mapped Date    : Tue May 11 12:44:56 2010

Design Summary
--------------
Number of errors:      0
Number of warnings:   15
Slice Logic Utilization:
  Number of Slice Registers:                 1,126 out of  69,120    1%
    Number used as Flip Flops:               1,126
  Number of Slice LUTs:                      1,150 out of  69,120    1%
    Number used as logic:                    1,123 out of  69,120    1%
      Number using O6 output only:             853
      Number using O5 output only:             243
      Number using O5 and O6:                   27
    Number used as Memory:                       9 out of  17,920    1%
      Number used as Shift Register:             9
        Number using O6 output only:             9
    Number used as exclusive route-thru:        18
  Number of route-thrus:                       266
    Number using O6 output only:               261
    Number using O5 output only:                 5

Slice Logic Distribution:
  Number of occupied Slices:                   491 out of  17,280    2%
  Number of LUT Flip Flop pairs used:        1,405
    Number with an unused Flip Flop:           279 out of   1,405   19%
    Number with an unused LUT:                 255 out of   1,405   18%
    Number of fully used LUT-FF pairs:         871 out of   1,405   61%
    Number of unique control sets:              94
    Number of slice register sites lost
      to control set restrictions:             221 out of  69,120    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                       120 out of     640   18%
    Number of LOCed IOBs:                      120 out of     120  100%
    IOB Flip Flops:                            178

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                       4 out of     148    2%
    Number using BlockRAM only:                  4
    Total primitives used:
      Number of 18k BlockRAM used:               8
    Total Memory used (KB):                    144 out of   5,328    2%
  Number of BUFG/BUFGCTRLs:                      9 out of      32   28%
    Number used as BUFGs:                        9
  Number of IDELAYCTRLs:                         4 out of      22   18%
  Number of DCM_ADVs:                            4 out of      12   33%

Average Fanout of Non-Clock Nets:                2.91

Peak Memory Usage:  668 MB
Total REAL time to MAP completion:  58 secs
Total CPU time to MAP completion:   54 secs
```

Appendix A.4

ABB2 (AVNET board) FPGA implementation report

```
Release 11.5 Map L.70 (nt64)

Xilinx Mapping Report File for Design 'v5pcieDMA'

Design Information
------------------
Command Line   : map -ise v5PCIeDMA.ise -intstyle ise -p xc5vlx110t-ff1136-1 -w
-logic_opt on -ol high -xe c -t 1
-register_duplication on -global_opt speed -retiming on
-equivalent_register_removal off -mt off -cm speed -ir off -pr
off -ignore_keep_hierarchy -lc off -power off -o v5pcieDMA_map.ncd v5pcieDMA.ngd
v5pcieDMA.pcf
Target Device  : xc5vlx110t
Target Package : ff1136
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.51.18.1 $
Mapped Date    : Fri Nov 19 11:38:49 2010

Design Summary
--------------
Number of errors:      0
Number of warnings:   99
Slice Logic Utilization:
  Number of Slice Registers:              16,968 out of  69,120   24%
    Number used as Flip Flops:            16,964
    Number used as Latches:                    4
  Number of Slice LUTs:                   18,659 out of  69,120   26%
    Number used as logic:                 16,886 out of  69,120   24%
      Number using O6 output only:        15,519
      Number using O5 output only:           895
      Number using O5 and O6:                472
    Number used as Memory:                 1,685 out of  17,920    9%
      Number used as Dual Port RAM:          502
        Number using O6 output only:         336
        Number using O5 output only:          48
        Number using O5 and O6:              118
      Number used as Shift Register:       1,183
        Number using O6 output only:       1,183
    Number used as exclusive route-thru:      88
  Number of route-thrus:                   1,014
    Number using O6 output only:             977
    Number using O5 output only:              33
    Number using O5 and O6:                    4

Slice Logic Distribution:
  Number of occupied Slices:               7,735 out of  17,280   44%
  Number of LUT Flip Flop pairs used:     23,286
    Number with an unused Flip Flop:       6,318 out of  23,286   27%
    Number with an unused LUT:               4,627 out of  23,286   19%
    Number of fully used LUT-FF pairs:      12,341 out of  23,286   52%
    Number of unique control sets:           1,130
    Number of slice register sites lost
      to control set restrictions:           2,633 out of  69,120    3%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                       121 out of     640   18%
    Number of LOCed IOBs:                      121 out of     121  100%
    IOB Flip Flops:                            178
    Number of bonded IPADs:                     16 out of      50   32%
    Number of bonded OPADs:                     12 out of      32   37%

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                      41 out of     148   27%
    Number using BlockRAM only:                 37
    Number using FIFO only:                      4
    Total primitives used:
      Number of 36k BlockRAM used:              27
      Number of 18k BlockRAM used:              15
      Number of 36k FIFO used:                   4
    Total Memory used (KB):                  1,386 out of   5,328   26%
  Number of BUFG/BUFGCTRLs:                     19 out of      32   59%
    Number used as BUFGs:                       19
  Number of IDELAYCTRLs:                         4 out of      22   18%
  Number of BUFDSs:                              2 out of       8   25%
  Number of DCM_ADVs:                            3 out of      12   25%
  Number of GTP_DUALs:                           3 out of       8   37%
    Number of LOCed GTP_DUALs:                   3 out of       3  100%
  Number of PCIEs:                               1 out of       1  100%
  Number of PLL_ADVs:                            4 out of       6   66%

Average Fanout of Non-Clock Nets:                3.80

Peak Memory Usage:  1336 MB
Total REAL time to MAP completion:  24 mins 32 secs
Total CPU time to MAP completion:   24 mins 21 secs
```

Appendix A.5

ML605 FPGA implementation report

```
Release 12.4 Map M.81d (nt64)

Xilinx Mapping Report File for Design 'v6pcieDMA'

Design Information
------------------
Command Line   : map -intstyle ise -p xc6vlx240t-ff1156-1 -w -logic_opt off -ol
high -xe n -t 1 -xt 0
-register_duplication off -r 4 -global_opt speed -retiming on
-equivalent_register_removal on -mt 2 -ir off
-ignore_keep_hierarchy -pr off -lc off -power off -o v6pcieDMA_map.ncd
v6pcieDMA.ngd v6pcieDMA.pcf
Target Device  : xc6vlx240t
Target Package : ff1156
Target Speed   : -1
Mapper Version : virtex6 -- $Revision: 1.52.76.2 $
Mapped Date    : Tue Apr 12 14:40:46 2011

Design Summary
--------------
Number of errors:      0
Number of warnings:    1
Slice Logic Utilization:
  Number of Slice Registers:                 8,721 out of 301,440    2%
    Number used as Flip Flops:               8,719
    Number used as Latches:                      2
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                     10,476 out of 150,720    6%
    Number used as logic:                    9,800 out of 150,720    6%
      Number using O6 output only:           8,882
      Number using O5 output only:             593
      Number using O5 and O6:                  325
      Number used as ROM:                        0
    Number used as Memory:                     545 out of  58,400    1%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            0
      Number used as Shift Register:           545
        Number using O6 output only:           524
        Number using O5 output only:             1
        Number using O5 and O6:                 20
    Number used exclusively as route-thrus:    131
      Number with same-slice register load:     93
      Number with same-slice carry load:        38
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                 3,334 out of  37,680    8%
  Number of LUT Flip Flop pairs used:       11,495
```

```
    Number with an unused Flip Flop:         2,990 out of  11,495   26%
    Number with an unused LUT:               1,019 out of  11,495    8%
    Number of fully used LUT-FF pairs:       7,486 out of  11,495   65%
    Number of unique control sets:             243
    Number of slice register sites lost
      to control set restrictions:             730 out of 301,440    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                         9 out of     600    1%
    Number of LOCed IOBs:                        9 out of       9  100%
    Number of bonded IPADs:                     10
    Number of bonded OPADs:                      8

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                 21 out of     416    5%
    Number using RAMB36E1 only:                 17
    Number using FIFO36E1 only:                  4
  Number of RAMB18E1/FIFO18E1s:                  1 out of     832    1%
    Number using RAMB18E1 only:                  1
    Number using FIFO18E1 only:                  0
  Number of BUFG/BUFGCTRLs:                      5 out of      32   15%
    Number used as BUFGs:                        5
    Number used as BUFGCTRLs:                    0
  Number of ILOGICE1/ISERDESE1s:                 0 out of     720    0%
  Number of OLOGICE1/OSERDESE1s:                 0 out of     720    0%
  Number of BSCANs:                              1 out of       4   25%
  Number of BUFHCEs:                             0 out of     144    0%
  Number of BUFOs:                               0 out of      36    0%
  Number of BUFIODQSs:                           0 out of      72    0%
  Number of BUFRs:                               0 out of      36    0%
  Number of CAPTUREs:                            0 out of       1    0%
  Number of DSP48E1s:                            0 out of     768    0%
  Number of EFUSE_USRs:                          0 out of       1    0%
  Number of FRAME_ECCs:                          0 out of       1    0%
  Number of GTXE1s:                              4 out of      20   20%
    Number of LOCed GTXE1s:                      4 out of       4  100%
  Number of IBUFDS_GTXE1s:                       1 out of      12    8%
    Number of LOCed IBUFDS_GTXE1s:               1 out of       1  100%
  Number of ICAPs:                               1 out of       2   50%
  Number of IDELAYCTRLs:                         0 out of      18    0%
  Number of IODELAYE1s:                          0 out of     720    0%
  Number of MMCM_ADVs:                           1 out of      12    8%
  Number of PCIE_2_0s:                           1 out of       2   50%
    Number of LOCed PCIE_2_0s:                   1 out of       1  100%
  Number of STARTUPs:                            1 out of       1  100%
  Number of SYSMONs:                             0 out of       1    0%
  Number of TEMAC_SINGLEs:                       0 out of       4    0%

  Number of RPM macros:            9
Average Fanout of Non-Clock Nets:                4.10

Peak Memory Usage:  1003 MB
Total REAL time to MAP completion:  9 mins 17 secs
Total CPU time to MAP completion (all processors):   9 mins 24 secs
```

Appendix A.6

Epoch marker (EM) indexing

An epoch is about 16 µs, which corresponds to about 160 events. To every epoch, the synchronization system attaches an epoch marker (EM). Epoch markers are important in the event building. They are searched by the software but it takes CPU time. The epoch markers are going to be picked up in ABB. The link module (contributed by **CAG**) interface will provide a signal telling that the current message packet is an epoch marker or not. Identifying this signal, the ABB fabric logic is able to process the epoch markers in several ways on behalf of the software convenience. The original copy of EMs stay in the data block and their address information or tags will be picked out and stored in the EM index collector (a separate set of memory space), so that it is easy for the applications to trace the EM entries in the data space.

We prefer the option of a ring-buffer in the host memory space and acknowledgement via interrupts. The proposed mechanism is to help the software locate the EMs in memory region if there are any. ABB is responsible to provide methods to control and monitor the EM collector, answering "how many" and "where" questions to the host. ABB also provides the EM statistics register in the FPGA, namely the producer counter, which is read-only to the host but can be reset by the host. The host can have its own consumer counter, which is invisible to the ABB. In this way, the producer counter is updated by the ABB and the consumer counter is updated by the host. Otherwise, the host can throw the counter management work to the ABB, when every time it reads EM tags away, the corresponding amount of consumption is sent to a reduction register in ABB, where the subtraction will be done.

A.6.1 EM preparation

A.6.1.1 EM format

The 32-bit logical addresses of the EM entry in the hit data memory space are the content of the EM collector, so a more precise name for it should be EM tag collector. Every time an EM is transferred to the host memory space via DMA (read), ABB writes a copy of its destination address (pointer) into the EM collector, at the same time updating the producer counter and increasing the next pointer. Relative addresses are suitable for software.

A.6.1.2 Storage destination/EM collector

The software provides one page or several pages of memory to ABB as the EM storage destination, whose starting address is written into a defined register in ABB BAR[0]. The ABB then, after enabled, writes EMs sequentially to that area, meanwhile increasing the EM counter in another register in ABB. After the software reads an EM, the corresponding entry is supposed to be available to ABB and the EM counter is decreased. The software is required to guarantee enough pages for EMs. This can be checked by polling the EM counter register.

Actually this is a software-implemented FIFO, whose ownership belongs to the software.

An EM tag is 32-bit in size. One 4KB page can therefore hold `1024` indices.

A.6.1.3 Buffer status acknowledge to the host

The ring-buffer itself should be supervised for a safe application. The threshold is configurable via software to define when the application is expected to be acknowledged in case of buffer overflow. If overflow is going to happen, ABB sends INT to the host and the software can choose to disable the EM storage to prevent the current EM tags from lost, or to allow the FPGA overwriting the existing EM tags. Polling approach is also supported in the development phase.

A.6.2 Implementation

The EM handling module is illustrated in figure A.6-1. An additional TLP channel is added to hold the epoch marker indices and takes part in the PCI Express Tx arbitration.

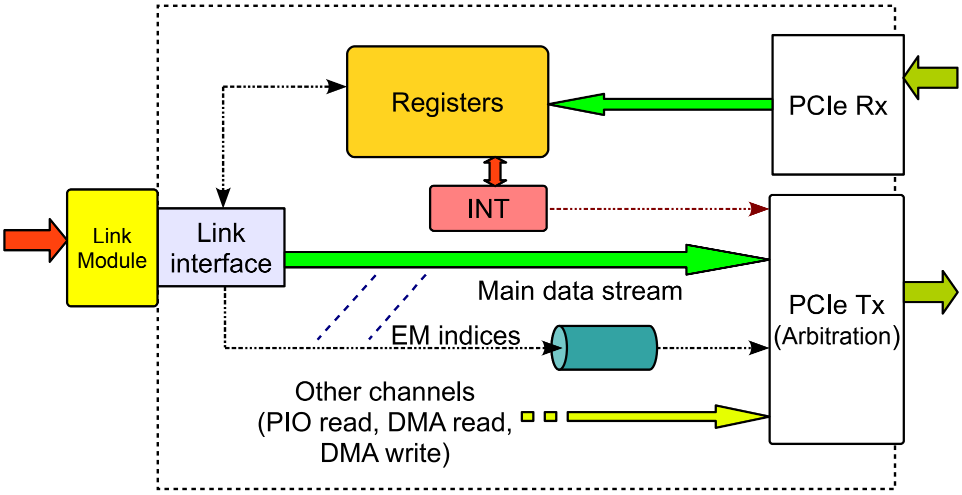


Figure A.6-1 Epoch marker handling module block diagram

A.6.2.1 Registers

In the ABB FPGA, six extra registers are set and assigned to four addresses, as listed in table A.6-1. They are all 32-bit in width and allocated in BAR[0]. RING_START is the start (physical) address of the ring-buffer and RING_SIZE is the ring-buffer size measured in bytes. CNT_PRODUCER and CNT_REDUCE are assigned to the same address `+0x00B8`, related to the producer counter. RING_STATUS and RING_CTRL are assigned to the same address `+0x00BC`. The application can ignore the CNT_REDUCE register, if the host calculates the ring-buffer occupancy by itself. If the host application wants the ABB to manage the ring-buffer occupancy, the CNT_REDUCE register provides the possibility to synchronize the producer and the consumer. Note that, if the application chooses not to use the CNT_REDUCE register, the INT_Enable bit in the RING_CTRL register must be '0', otherwise, the ABB will issue unexpected interrupts whenever the CNT_PRODUCER value exceeds a defined threshold.

Table A.6-1 EM-related registers

Specifier	**Description**	**offset**	**R/W**
RING_START	Start (physical) address of the ring buffer	`+0x00B0`	R+W
RING_SIZE	Size of the ring buffer in bytes	`+0x00B4`	R+W
CNT_PRODUCER	EM tags count in the producer side	`+0x00B8`	R
CNT_REDUCE	Number of EM tags taken by the consumer	`+0x00B8`	W
RING_STATUS	Ring buffer status	`+0x00BC`	R
RING_CTRL	Ring buffer control	`+0x00BC`	W

Ring buffer status and control register definitions are listed in tables A.6-2 and A.6-3.

In table A.6-2, Reset is defined as `0x0A`. A reset command is combined with reset type bits. For example, `0x010A` resets the EM ring-buffer (puts the producer pointer to the start), `0x020A` resets the CNT_PRODUCER to zero. INT_Enable enables the interrupt acknowledge. For this version, the interrupt is issued when the threshold of EM occupancy is reached. Bits[29:16] define the interrupt threshold divided by `4`, which means, the threshold granularity is `4` EM tags and the maximum threshold is `0xFFFC = 65532` tags. This granularity is a fixed parameter and can be changed in the future version according to the real operation state. If the number of maximum EM entries is less than 16K (14-bit range), we can set this granularity to unit `1`.

In table A.6-3, RING_STATUS register are defined, where reserved bits read always 0. INT_Enable, EM_Enable and the interrupt threshold can be reviewed by reading this register. Almost_full bit indicates the ring-buffer is almost full and it is asserted according to the Threshold/4 value.

In both tables A.6-2 and A.6-3, the 32 bits are partitioned into higher and lower 16 bits. Higher and lower partitions are relatively independent upon each other.

Table A.6-2 RING_CTRL definition

31	30	29 ~ 16	15	14 ~ 10	9	8	7 ~ 0
INT_Enable	reserved	Threshold/4	EM_Enable	reserved	RST_CNT	RST_REW	**Reset**

Table A.6-3 RING_STATUS definition

31	30	29 ~ 16	15	14 ~ 0
INT_Enabled	Almost_full	Threshold/4	EM_Enabled	reserved

A.6.2.2 Software side

Before starting DAQ with EM indexing assistance, the host should

(a) set up the ring-buffer,

(b) write the RING_START and RING_SIZE registers in the ABB firmware,

(c) write the EM_ENABLE bit in the RING_CTRL register to start the EM indexing; if the INT is used, INT_ENABLE and the threshold should also be provided.

Then the host can start the DMA. During the operation, the host may check the status by polling the RING_STATUS register and calculate the EM collector occupancy with CNT_PRODUCER. It can also orientate itself to respond the interrupts. If necessary, the software application maintains itself a counter for the EM tag consumption, e.g. CNT_CONSUMER.

If the ring-buffer is almost full and the INT is enabled, the host will receive an INT correspondingly. Then the host can choose to

(1) clear/reset the current ring-buffer, and some EMs may get lost,

(2) pause the EM module for a while and resume again after a new ring-buffer is allocated and updated to the FPGA, and no EMs get lost.

Even for the best choice (2), the software should be fast enough in processing (fetching) the EM tags, to make places for subsequent EM tags.

What should be done when the ring-buffer gets full and EM tags get lost, is to be defined in the future.

The EM range parameters, RING_START and RING_SIZE, can be modified at any instance. However, to make them effective in the FPGA, a write to the RING_CTRL register with EM_ENABLE bit asserted must be issued. This provides a way to update those range parameters while the EM module is running.

A.6.2.3 Hardware side

On the transaction layer of the PCI Express, the MWr TLP is the best choice for the EM tag transfer. Therefore, another TLP channel will be built for sending the EM tags to the host space (the ring-buffer). If the EM tag collector is not empty, the logic reads out one entry and builds up an MWr TLP. After the arbitration of the Tx module, this MWr TLP will go to the host space, somewhere in the range defined by RING_START and RING_SIZE registers. The FPGA then increases the current address for the next write. If the ring-buffer end is reached, the address pointer will rewind to RING_START.

INT_Enable bit in RING_CTRL is to enable the interrupt. EM_Enable enables the EM indexing function. Switching EM_Enable on or off will start or stop the EM indexing, as well as enable new range parameters or disable old parameters. The producer counter and the EM collector buffer are not affected through individual EM_Enable write. The EM_Reset must be combined with specific bit(s) to take effect. EM_Reset(RST_REW) rewinds the producer pointer to the start, some similar action to empty the EM collector buffer; EM_Reset(RST_CNT) clears the the CNT_PRODUCER counter to zero. In table A.6-4, some typical commands for the lower 16 bits of RING_CTRL are listed. The higher 16 bits are for interrupts and overwrite should be avoided in lower commands if INT is used.

Table A.6-4 Typical EM commands: lower 16 bits (write to RING_CTRL[15:0])

Command [15:0]	Break-down	Effects

X"830A"	EM_Enable = '1'
	Reset = X"A"
	RST_CNT = '1'
	RST_REW = '1'
		Clear EM collector buffer, producer pointer rewound to RING_START.
		Clear CNT_PRODUCER register.
		Take the range parameters.
		Start/Continue EM indexing.

X"030A"	EM_Enable = '0'
	EM_Rst = X"A"
	RST_CNT = '1'
	RST_REW = '1'
		EM indexing stops.
		Clear EM collector buffer, producer pointer rewound to RING_START.
		Clear CNT_PRODUCER register.
		Return to initial state without EM indexing.

X"8000"	EM_Enable = '1'
	EM_Rst = X"0"
		Start/Resume EM indexing.
		Take the range parameters.
		(No effect when Enabled='1' and range parameters not changed)

X"0000"	EM_Enable = '0'
	EM_Rst = X"0"
		Halt EM indexing.
		(No effect when Enabled='0')

X"810A"	EM_Enable = '1'
	EM_Rst = X"A"
	RST_REW = '1'
		Start/Resume EM indexing.
		Clear EM collector buffer, producer pointer rewound to RING_START.
		(CNT_PRODUCER register not changed.)

X"820A"	EM_Enable = '1'
	EM_Rst = X"A"
	RST_CNT = '1'
		Start/Resume EM indexing.
		Clear CNT_PRODUCER register.
		(producer pointer not moved.)

A.6.3 Sequential coherence

Ideally, the EM indices go together with the data stream and at the read-out port, they have perfect timing correlation. However, for the AVNET board, the DDR-II SDRAM is in 64-bit bus. This makes it difficult to combine the EM indices along with the data.

Against this shortage, we have two options, branched index storage and token insertion.

(1) Branched index storage

The indices are in a branch of the main data stream and hold the information of the special message positions. If the timing relationship is not well controlled, it is probable that the index comes too much earlier in the ring-buffer than the corresponding EM message does in the data memory space. If the host application happens to get this fake index and fetch the indexed EM in the main data space, there is error.

To avoid such situation, a correlation shall be maintained between the index and the EM entry so that the index goes into the host memory always later than the EM entry and the sequential relationship is correctly sustained. Of course, being too much later is also not acceptable because the latency can worsen the performance.

Another difficulty to keep this coherence comes from the internal DAQ hit packet buffer which is 64 bits in width. This buffer is built upon the DDR2 SDRAM SODIMM module and no additional bits are available for index marking or framing. Due to its large size, 64-bit, 256 MByte, additional index marking bit is not practical, since we would need at least 256/8 = 32 Mb memory inside the FPGA. This exceeds the maximum supported BRAM volume in Virtex5 LX110T, i.e. 6.448 Mbits. Some of them must be used as DMA buffers in the logic.

And with current arbitration policy which is mainly packet-based, the data packets can be delayed with very high possibility, because the quantitative ratio between data and EM indices is quite high.

A straightforward solution comes out of weighted priority arbitration, which gives the data packets higher priority than the indices. However, this weight is possibly varying during the DAQ process. For example, an EM may arrive after 8KB hit data, and may also arrive after 16KB. If this interval can be fixed, we would not have to use EM indexing, whereas a hit counting mechanism could work. In this means, our arbitration policy is called variable weighted priority arbitration. To have such kind of arbitration, a FIFO is built up to store the interval number between consecutive EMs. This interval FIFO is 27 bits (addressing range 128 M × 8 B = 1 GB) wide and 1024 entries deep, which buffers maximum 1024 intervals. These 1024 intervals should cover all range of the 256 MB data buffer. This corresponds to a maximum EM rate of about

$$1024/\ (256 \times 10^6 / 8) = 1/32768 \approx 0.003\%.$$

If this rate is much higher in practical environment, we can consider to use 2-stage indexing buffer.

With this interval FIFO, the Tx can know exactly the next position of EM index, before which the EM index buffer will not be read. It guarantees that the index request is always issued after the correlated EM packet is read out. In this way, the index arbitration is always made after the corresponding EM entry and as well as not too late, about one or two TLPs later. This gives enough time to the host memory to settle the dependency and to avoid dummy EM indices fetching.

(2) Epoch marker index token insertion

The branched solution has limitation upon the maximum epoch marker rate due to the implementation style with the FPGA. The correlation to avoid dummy EM indices is complicated.

Therefore, we have considered the token insertion solution, which inserts a token word if an EM comes and if a data word identical to the token word, the word is duplicated. By reading the event buffer, doubled token symbols are taken as a data of token word and a separated single token symbol is an epoch marker. For an epoch marker, a write packet carrying the address offset information is sent to the current epoch marker index position in the host memory space. The offset information is acquired by a counter attached to the read port of the event buffer, which increases itself at every valid data read.

The EM indices are stored in the external SDRAM resource and have no limit on the epoch marker density. And there is no need to correlate the indices and the EM entity, because they have good timing relationship. It guarantees that the index MWr request is always issued after the correlated EM packet is read out. Same as the option (1), the index arbitration is always after the corresponding EM entry and as well as not too late. Such a token word should be chosen carefully with lower appearance probability, otherwise the overhead to frequently insert the token words will be noticeably high.

Another consideration goes to the PCI Express Tx TLP building, which starts the TLP building after the DMA read module has made sure that the required amount of data is ready in the event buffer. However, if there are ghost data (tokens) in the buffer, fake TLP request will be issued from the DMA read module due to incorrect payload count. A solution is feasible to have a small-size FIFO between the event buffer and the PCI Express Tx module, so that only the pure payload is counted by the DMA read module. The minimum accommodation of the small-size FIFO should be larger than the size of a maximum size TLP (`MAX_PAYLOAD_SIZE`), which is 128 bytes in our system. Of course, this solution introduces extra cycles of delay into the main data path and the data count difficulty for the total amount of data due to the extra small pure-payload FIFO. Another solution is preferred by means of uncounted token words, in which the write and read of inserted token words are both not counted. The data count logic bypasses such token words on both ports. In this way, the inaccuracy of the data count and hazard of fake requests are avoided.

Insertion of EM tokens is not a big penalty to the event buffer performance if the arrival rate of EM is not too high. If one EM is expected per 4KB hit data, the overhead is only

$$1/1024 \approx 0.1\ \%,$$

which can be ignored for our buffer size over 256 MB.

Therefore, we take the solution (2). Such insertion should be generalized because in the future, there might be more markers to be processed.

Appendix A.7

Interrupt Generator (IG)

The interrupt service capability is critical in the DAQ system because of irregular data flow. To test this capability, a configurable interrupt generator (IG) is developed and can be optionally implemented in the DMA logic project as figure 3-11. IG block diagram is depicted in figure A.7-1.

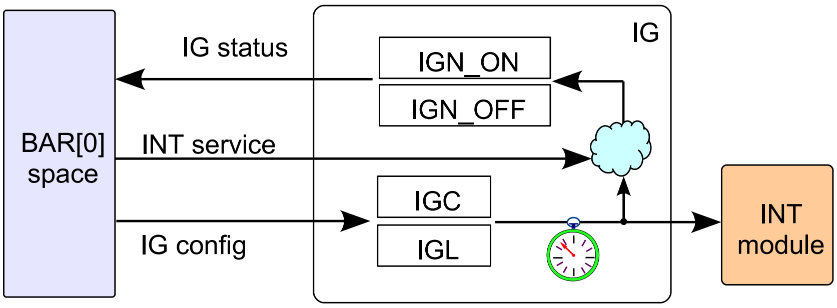


Figure A.7-1 Interrupt Generator block diagram

IGC (**I**nterrupt **G**enerator **C**ontrol) and IGL (**I**nterrupt **G**enerator **L**atency) are used as the Interrupt Generator control and parameter registers in BAR[0]. The corresponding bit in the Interrupt Status register (ISR) is bit 2.

A unit in IGL register is calculated as 8 ns (125 MHz clock rate) in ABB2 4-lane version. The value in IGL is the delay between two interrupts generated. To start the Interrupt Generator, a non-zero value is necessary to the IGL register. Thus, a zero written to the IGL register serves as a pause command, which does not reset the statistics registers but makes the Interrupt Generator stop generating interrupts. Resuming the paused interrupt issuing is done by writing a non-zero value to the IGL register. To emulate the interrupt process service, another feature word (`0x00F0`) is needed to be written into the IGC register, which clear one interrupt every time.

An example procedure is given below.

```
*(0xED000080) = 0x000A;       // Reset Interrupt Generator
*(0xED000010) = 0x0004;       // Enable interrupt generation
*(0xED000084) = 0x3000;       // Set Interrupt Generator Latency to 98304 ns,
                              //   and trigger it run

// Interrupt service program
void Int_Service ( )
{
 int Int_Index = *(0xED000008);
 if (Int_Index & 0x0004)         // Interrupt(s) from the Int Generator come
   {
    ... ...                      // Servicing
    *(0xED000080) = 0x00F0;      // Clear one interrupt
   }
}
```

Afterwards, the number of assert interrupts and that of the deassert interrupts can be obtained from the IGN_ON (+0x0088) and IGN_OFF (+0x008C) registers, respectively. Their values give information of the status of the interrupt service performance. These two statistic registers and the Interrupt Generation register are reset by a reset word (0x0A) written to the IGC register (+0x0080).

Tests show that the system we are using can stand an interrupt arrival rate of 60 kHz.

Appendix A.8

Data Generator (DG)

The data generator is built in the FPGA to enable the self-test and performance test on ABB2 in receiving the link data. The DG has a RAM table to hold its descriptors. A descriptor, made up of a 32-bit data part and a 32-bit instruction part, can be addressed by the host via BAR[1] (the same BAR with the memory BRAM). The DG RAM table is started from the offset `0x000C0000` and goes up to `0x000C7FF8`, altogether 4096 descriptors.

DG is attached to the transmit port of ABB2, as shown in figure 3-11.

The DG table structure is shown in figure A.8-1.

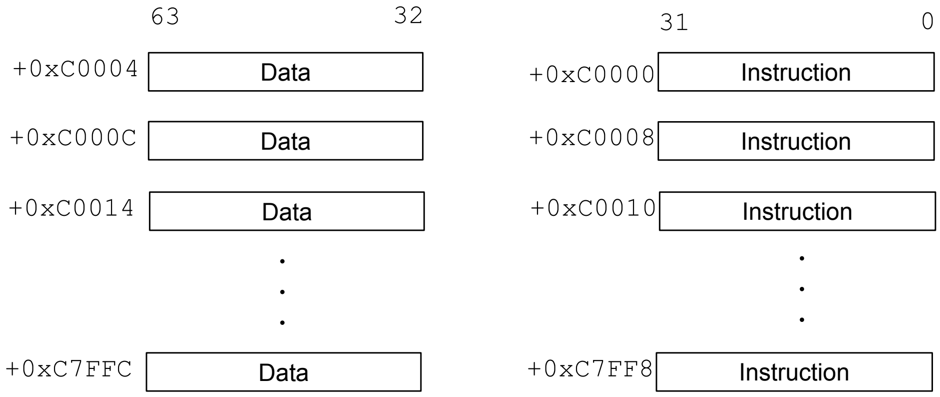


Figure A.8-1 Data Generator table-RAM

Table A.8-1 lists the bits for an instruction.

Table A.8-1 Descriptor format: lower 32 bits (instruction)

bit	31	30	29:28	27:16	15:00
Def.	Enable	Stop	Traffic class	Next address	Delay count

In table A.8-1,

Enable: starting descriptor. DG is not started if the Enable bit of the first descriptor is 0. Default 0. This bit is needed only for the first descriptor at address 0. In all other descriptors this bit is ignored.

Stop: DG terminates after this descriptor is executed. Default 0.

Traffic class: 1-DAQ; 2-CTL; 3-DLM; 0-Invalid. Default 0. Class 0 is reserved to avoid unexpected data sending out of a null descriptor.

Next address: absolute address of the next descriptor. 8-byte aligned or 3 digits right shifted.

Delay count: number in cycles between this descriptor and the next descriptor. One cycle for the card is 8 ns.

Table A.8-2 shows the data format of the DG, classified by traffic classes.

Table A.8-2 Descriptor format: higher 32 bits (data)

bit	63:51	50	49	48	47:32	
DAQ	***reserved***	**crc_err**	**sof**	**eof**	**DAQ Data**	
CTL	*reserved*	*reserved*	sof	eof	CTL Data	
DLM	*reserved*	*reserved*	DLM valid	*reserved*	[47:36] *reserved*	[35:32] DLM type

We can see that every 64-bit descriptor contains maximal 16-bit data payload for DAQ class data generation. So for non-loop mode, we can send maximal 8192 bytes data payload with the last descriptor marked with STOP.

The DG is reset and controlled via the DG_CTRL register in BAR[0], offset 0x00A8, as listed in table A.8-3. Bit[8] corresponds to DG Mask. For resets, 0x000A resets DG. Masks can of course be combined with resets.

Table A.8-3 Bit definition for DG Control register

bit	63:9	8	7:0
Def.	*reserved*	Mask	Reset command (0x0A)

The DG status can be acquired by reading the same address, as in table A.8-3.

Table A.8-4 Bit definition for DG Status register

bit	63:9	8	7:2	1	0
Def.	*reserved*	Mask	*reserved*	Busy	*reserved*

Both Mask and Busy bits are valid high.

Also, the presence of DG can be checked in the Global Status Register (GSR, offset 0x20). If the bit 5 in GCR is asserted, the firmware is equipped with a data generator, as the GSR definition 3.1.5.

The DG is reset by a write of 0x0A (C_DG_RESET) to the DG_CTRL register. After reset, DG starts to poll the address 0 for an enabled descriptor. So normally the address 0 of DG should be the last write in the program.

After started, the DG runs through according to the table content, until it gets a descriptor with STOP bit asserted; otherwise it will never stop. Therefore, the DG can be put into a loop mode, with a wind-back descriptor chain. In loop mode, the corresponding channel will be fill always almost full if the delay parameter for the descriptor is set enough small. A stopped DG must be reset to the address 0 to run again or to poll there.

The DG output is 16-bit data every descriptor. However, our buffer FIFO is 64-bit wide, so there might be some data shifting problem for DMA tests if the DG packets are not 64-bit aligned.

Traffic class 0 (invalid) can be used as idle delay if 16-bit delay (524 280 ns)is not enough.

Also a write of 0x0100 (C_DG_MASK)to the DG_CTRL register will mask the DG output, then no data come out of the DG even if its enabled. C_DG_MASK can be combined with the C_DG_RESET, i.e. writing 0x010A will reset the DG to the zero address as well as mask the DG output.

If a non-zero delay value comes together with the STOP bit, the delay will be ignored, because the DG is into halt state anyway.

The DAQ class interface definition have been upgrade, which expands the bus width from 16-bit to 64-bit. To this modification, the DG will simply duplicate the DAQ data in table A.8-2 four times to have a 64-bit data generation.

Bibliography

[1] R. K. Ellis, W. J. Stirling, B. R. Webber*; QCD and Collider Physics;* Cambridge University Press, 1996.

[2] Hubble Space Telescope; **URL** http://hubblesite.org/

[3] Giant Magellan Telescope; **URL** http://www.gmto.org/

[4] CERN; **URL** http://public.web.cern.ch/public/

[5] LHC; **URL** http://lhc.web.cern.ch/lhc/

[6] FAIR; **URL** http://www.gsi.de/portrait/fair.html

[7] B. Friman, C. Höhne, J. Knoll, S. Leupold, J. Randrup, R. Rapp, P. Senger (editors); *The CBM Physics Book*; Springer Science+Business Media, October 14, 2010.

[8] P. Senger; *The Compressed Baryonic Matter Experiment*; GSI, Darmstadt, June 2002.

[9] $\overline{\text{P}}$ANDA; **URL** http://www-panda.gsi.de/

[10] D. J. Griffiths. *Introduction to Elementary Particles (Second, Revised Edition)*. Wiley-VCH Verlag GmbH & Co. 2008.

[11] CBM collaboration; *Compressed Baryonic Matter Experiment Technical Status Report*; January 2006.

[12] GSI website; **URL** http://www.gsi.de

[13] *GSI Scientific Report 2009*; GSI Helmholtzzentrum für Schwerionenforschung GmbH; June 2010.

[14] *The ALICE experiment at the CERN LHC, Journal of Instrumentation, JINST 3 S08002*; The ALICE Collaboration; August 2008.

[15] *The ATLAS Experiment at the CERN Large Hadron Collider, Journal of Instrumentation, JINST 3 S08003*; The ATLAS Collaboration; August 2008.

[16] C. Steinle, A. Kugel, R. Männer; *Implementation of a Hough Tracker for CBM*; CBM Progress Report 2006; February 2007.

[17] Xilinx Inc.; *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide, UG076 (v4.1)*; November 2, 2008;

[18] Xilinx Inc.; *Virtex-5 FPGA RocketIO GTP Transceiver User Guide, UG196 (v2.1)*; December 3, 2009.

[19] Walter F. J. Müller; *CBM STS Readout Chain;* CBM DAQ Meeting, GSI, Darmstadt; January 14, 2010.

[20] A. Kotynia; *Status of STS Simulations*; CBM Meeting, GSI, Darmstadt; December 4, 2009.

[21] IDT Inc.; *2.5V 18M-BIT HIGH-SPEED TeraSync™ FIFO 36-BIT CONFIGURATIONS 524,288 × 36 IDT72T36135M*; February 2009.

[22] Infineon Technologies North America Corporation and Kingston Technology Company, Inc.; *Intel Dual-Channel DDR Memory Architecture White Paper, Rev. 1.0*; September 2003.

[23] InifniBand Trade Association; *InfiniBand Architecture Specification Volume 1, Release 1.1*; 2002

[24] Mark Norris; *Gigabit Ethernet Technology and Applications*; Artech House, Inc. 2003.

[25] R. Budruk, D. Anderson, T. Shanley; *PCI Express System Architecture*; MindShare, Inc., 2004.

[26] Don Anderson, Jay Trodden; *HyperTransport System Architecture*; MindShare, Inc., 2003.

[27] HyperTransport Technology Forum; *HyperTransport™ I/O Link Specification, Revision 3.00c*; 2007.

[28] Xilinx web site. **URL** http://www.xilinx.com

[29] Altera web site. **URL** http://www.altera.com

[30] Xilinx Inc.; *Virtex-4 FPGA User Guide, UG070 (v2.6)*; December 1, 2008.

[31] Xilinx Inc.; *Virtex-5 FPGA User Guide, UG190 (v5.3)*; May 17, 2010.

[32] Altera Co.; *Nios II Processor Reference Handbook*; December, 2010.

[33] Xilinx Inc.; *ChipScope Pro 12.3 Software and Cores User Guide, UG029 (v12.3)*; September 21, 2010.

[34] Altera Co.; *Quatus II Handbook Version 10.1 Volumn 3: Verification*; December, 2010.

[35] J. Strunk, A. Heinig, T. Volkmer, W. Rehm, H. Schick; *Run-time reconfiguration for HyperTransport coupled FPGAs using ACCFS*; First international workshop on HyperTransport research and applications (WHTRA 2009); Mannheim, Germany. 2009.

[36] Xilinx Inc.; *Partial Reconfiguration User Guide*, *UG702 (v 12.3)*; October 5, 2010.

[37] Altera Co.; *Introducing Innovations at 28 nm to Move Beyond Moore's Law*; Altera white paper, July 2010.

[38] Peter J. Ashenden; *The Designer's Guide to VHDL, 3rd Edition*; Morgan Kaufmann Publishers, 2008.

[39] Accellera; **URL** http://www.accellera.org/

[40] Samir Palnitkar; *Verilog HDL, A Guide to Digital Design and Synthesis, Second Edition*; Prentice Hall PTR, February 21, 2003.

[41] Aldec ActiveHDL; **URL** http://www.aldec.com/

[42] JHDL website; **URL** http://www.jhdl.org/

[43] David C. Black, Jack Donovan, Bill Bunton, Anna Keist; *SystemC: From the Ground Up, Second Edition*; Springer Science+Business Media, 2010.

[44] Stuart Sutherland, Simon Davidmann, Peter Flake; *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling, Second Edition*; Springer Science+Business Media, 2006.

[45] Cadence web site. **URL** http://www.cadence.com/

[46] Synopsys web site. **URL** http://www.synopsys.com/

[47] Mentor Graphics web site. **URL** http://www.mentor.com/

[48] Atmel web site. **URL** http://www2.atmel.com

[49] Lattice web site. **URL** http://www.latticesemi.com/

[50] M. Keating, P. Bricaud; *Reuse Methodology Manual for System-on-a-Chip Designs, Third Edtion*; Kluwer Academic Publishers, 2002.

[51] S. Manz, U. Kebschull; *Design and Implementation of the Read Out Controller for the GET4 TDC of the CBM ToF Wall Prototype*; GSI Scientific Report 2009.

[52] F. Lemke, D. Slogsnat, N. Burkhardt, U. Brüning; *A unified interconnection network with precise time synchronization for the CBM DAQ-system*; RT'09, Beijing, May 2009.

[53] W. Gao, A. Kugel, A. Wurz, G. Marcus, R. Männer; *Active buffer for DAQ in CBM experiment*; 16th IEEE-NPSS, Beijing, May 2009.

[54] J. Adamczewski-Musch, H.G. Essel, N. Kurz, S. Linev; *First release of Data Acquisition Backbone Core*; RT'09, Beijing, May 2009.

[55] Frank Lemke; *CBM protocol second generation*; CBM DAQ Meeting document; , 2010.

[56] A. Kugel; *CBM ABB Software*; CBM DAQ Meeting document; February 12, 2009.

[57] Micron ×72 SODIMM; **URL** http://micron.com/products/ProductDetails.html?product= products /dram_modules/sodimm/MT18HTF25672PKZ-667

[58] Clifford E. Cummings, Peter Alfke; *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*; SNUG-2002, San Jose, CA, 2002.

[59] Clifford E. Cummings; *Simulation and Synthesis Techniques for Asynchronous FIFO Design*; SNUG-2002, San Jose, CA, 2002.

[60] PCI-SIG; *PCI Express^{TM} Base Specification Revision 1.1*; March 28, 2005.

[61] PCI-SIG; *PCI Express^{TM} Base 2.0 Specification*; January 15, 2007.

[62] PCI-SIG; *PCI Express^{TM} Base 3.0 Specification*; November 18, 2010.

[63] Xilinx Inc.; *LogiCORE™ Endpoint v3.6 for PCI Express® User Guide, UG185*; October 10, 2007.

[64] Xilinx Inc.; *LogiCORE™ IP Endpoint Block Plus v1.9 for PCI Express® User Guide, UG341*; September 19, 2008.

[65] Xilinx Inc.; *Virtex-6 FPGA Integrated Block for PCI Express, Pre-Production User Guide, UG671 (v1.0)*; October 5, 2010.

[66] Andrew S. Tanenbaum; *Modern Operating System, 3rd Ed.*; Prentice-Hall, Inc.; 2008

[67] Lun Li, Mitchell A. Thornton; *Digital System Verification: A Combined Formal Methods and Simulation Framework*; Morgan & Claypool, 2010.

[68] Daijue Tang; *Boolean quantification techniques with applications in formal verification: algorithms and analysis*; Doctor dissertation of Princeton University, 2007.

[69] F. Lemke; *FSMDesigner4 - Development of a Tool for Interactive Design and Hardware Description Language Generation of Finite State Machines*; Diploma Thesis presented to the Computer Engineering Department, Mannheim University, 2006.

[70] W. Gao, A. Kugel, R. Männer, N. Abel, N. Meier, U. Kebschull; *DPR in CBM: an Application for High Energy Physics*; DATE09, Nice, France, 2009.

[71] G. Marcus, G. Lienhart, A. Kugel, R. Männer; *On buffer management strategies for high performance computing with reconfigurable hardware*; IEEE-FPL, 2006.

[72] A. Kugel; *The ATLAS ROBIN – A High-Performance Data Acquisition Module*; Doctor dissertation of Mannheim University; 3 September 2009.

[73] Gerhard Lienhart; *Beschleunigung Hydrodynamischer Astrophysikalischer Simulationen mit FPGA-Basierten Rekonfigurierbaren Koprozessoren*; Doctor dissertation of Heidelberg University; 23 August 2004.

[74] Avnet Electronics Marketing, *Xilinx® Virtex™-5 PCI Express Development Kit User Guide (Rev. 1.0)*; December 04, 2007.

[75] PCIe SG DMA project on OpenCores.org; **URL** http://opencores.org/project,pcie_sg_dma

[76] Xilinx Inc.; *ML605 Hardware User Guide, UG534 (v1.5)*; February 15, 2011.

[77] Guillermo Marcus Martinez; *Acceleration of Astrophysical Simulations with Special Hardware*; Doctor dissertation of Heidelberg University, 22 March 2011.

[78] B. Osterloh, H. Michalik, S.A. Habinc, B. Fiethe; *Dynamic Partial Reconfiguration in Space Applications*; NASA/ESA Conference on Adaptive Hardware and Systems, 2009. San Francisco, USA.

[79] Davin Lim, Mike Peattie; *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations, XAPP290 (v1.0)*; Xilinx Inc.; May 17, 2002.

[80] Emi Eto; *Difference-Based Partial Reconfiguration, XAPP290 (v2.0)*; Xilinx Inc.; December 3, 2007.

[81] Xilinx Inc.; *Early Access Partial Reconfiguration User Guide For ISE 8.2.01i, UG208 (v1.1)*; March 6, 2006.

[82] Xilinx Inc.; *PlanAhead User Guide* 9.2; July 27, 2007.

[83] Xilinx Inc.; *PlanAhead User Guide*, *UG632 (v 12.4)*; December 21, 2010.

[84] Scott Hauck, André DeHon (editors); *Reconfigurable computing: the theory and practice of FPGA-based computation*; Elsevier Inc., 2008.

[85] N. Abel, S. Manz, F. Grull, U. Kebschull; *Increasing design changeability using dynamical partial reconfiguration*; RT'09, Beijing, May 2009.

[86] N. Abel; *Design and Implementation of an Object-Oriented Framework for Dynamic Partial Reconfiguration*; Doctor dissertation of Heidelberg University; 2011.

[87] PRCC; **URL** http://users.isc.tuc.gr/~kpapadimitriou/prcc.html

[88] Xilinx Inc.; *Virtex-4 FPGA Configuration User Guide, UG071 (v1.11)*; June 9, 2009.

[89] Xilinx Inc.; *Virtex-5 FPGA Configuration User Guide, UG191 (v3.9.1)*; August 20, 2010.

[90] Xilinx Inc.; *Virtex-6 FPGA Configuration User Guide, UG360 (v3.2)*; November 1, 2010.

[91] Xilinx Inc.; *PlanAhead User Guide*, *UG632 (v 13.1)*; March 1, 2011.

[92] J. Bergeron; *Writing Testbenches - Functional Verification of HDL Models*; Kluwer Academic Publishers, 2002.

[93] S. Meiyappan, K. Jaramillo, P. Chambers; *10 tips for generating reusable VHDL*; **URL** www.ednmag.com; August 19, 1999.

[94] Hamburg HDL Archive; **URL** http://tams-www.informatik.uni-hamburg.de/vhdl/vhdl.html; Last modified 15 January 2008.

[95] Open FPGA; **URL** http://www.openfpga.org/

[96] ANTLR v3; **URL** http://www.antlr.org/

[97] LLVM; **URL** http://llvm.org/.

[98] BlueSpec Inc.; **URL** http://www.bluespec.com/

[99] *A Survey of FPGA Benchmarks*; **URL** http://www1.cse.wustl.edu/~jain/cse567-08/ftp/fpga/

List of errata

No.	Page	Line	Original	Corrected
1	4	13	the huge data rate about 1 TB/s out of the high reaction rate about 10 MHz	the huge data rate **of** about 1 TB/s **due to** the high reaction rate **of** about 10 MHz
2	5	10	5 GT/s	5 **Gbps**
3	7	3	and InfiniBand was used to deliver the data to the back-end because of its robust and availability of software frameworks	and InfiniBand **were** used to deliver the data to the back-end **due to their robustness** and availability of software frameworks
4	17	34	universal solutions	universal **solution**
5	51	14 19 23	TAG BRAM	TAG **RAM**
6	61	26	they changes	they **change**
7	67	12	provides coverage interface	**provide** coverage interface
8	71	6	is founded	is **found**
9	73	11	figure 3-34	figure **3-36**
10	103	21	formally in	formally **be** in
11	129	11	a comment inside	a comment **appears** inside