



TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Digital and Computer Systems

TRANSPORT TRIGGERED ARCHITECTURES ON FPGA

Technical Report

Miika Niiranen

Version 1.1

Modified 2004-11-17

REVISION HISTORY

Version	Date	Author	Notes
1.0	2004-10-07	Miika Niiranen	Original M.Sc. Thesis
1.1	2004-11-17	Miika Niiranen	Revised Fig. 4.13, changed document type from Thesis to Technical Report

PREFACE

The work for this thesis was carried out at the Institute of Digital and Computer Systems in Tampere University of Technology during 2004 as a part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency.

I am sincerely grateful to my thesis supervisor, Professor Jarmo Takala, for his valuable guidance, confidence, and patience during the past year I have been working at the Institute of Digital and Computer Systems. I would also like to thank my colleagues, especially M.Sc. Riku Uusikartano, for their support and contribution to this thesis.

Tampere, October 5, 2004

Miika Niiranen

Insinöörinkatu 59 A 27

33720 TAMPERE

044-3490799

miika.niiranen@tut.fi

TABLE OF CONTENTS

REVISION HISTORY	II
PREFACE.....	III
TABLE OF CONTENTS	IV
ABSTRACT	VI
TIIVISTELMÄ.....	VIII
ABBREVIATIONS	X
1. INTRODUCTION	1
2. ALTERA EXCALIBUR DEVELOPMENT BOARD	3
2.1 ARM922T Processor.....	4
2.2 Advanced Microcontroller Bus Architecture	4
2.3 Embedded Stripe	8
2.4 Programmable Logic Devices	10
2.5 Development Board Memory Regions.....	14
3. SLAVE PROCESSORS ON FPGA	16
3.1 FPGA Design Recommendations and Notes	16
3.2 Communication between a Master and a TTA Slave Processor	19
4. TTA DCT32 PROCESSOR.....	21
4.1 Processor Structure.....	21
4.2 Global Control Unit.....	22
4.3 Interconnection.....	29
4.4 Function Units and Register Files	31
5. FPGA IMPLEMENTATIONS	33
5.1 Principles	34
5.2 Hardware Implementation Details	36

5.3 Software Details	40
5.4 Co-verification	42
5.5 Results	44
5.6 Different TTA Processors	50
6. CONCLUSIONS	54
REFERENCES	56
APPENDIX A.....	58
APPENDIX B.....	59
APPENDIX C.....	63

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Degree Program in Information Technology

Digital and Computer Systems

Niiranen, Miika Johannes: Transport Triggered Architectures on FPGA

Master of Science Thesis: 67 pages, 8 appendix pages

Examiners: Prof. Jarmo Takala, M.Sc. Riku Uusikartano

Funding: National Technology Agency, Nokia, Texas Instruments (France), Vaisala, Patria New Technologies, Instrumentointi, Elektrobit, Finnish Naval Research Institute

Department of Information Technology

October 2004

Keywords: Transport triggered architecture, field-programmable gate array

Programmable logic devices (PLDs) are, along with field-programmable gate array (FPGA) technology, expanding their popularity among hardware designers. These devices are a good alternative to be used in prototyping and as a manufacturing technology for relatively simple processors. Transport triggered architecture (TTA) processors, on the other hand, are an excellent option to be used as coprocessors or hardware accelerators. This property is due to hardware simplicity involved in the processor architecture. Thus, PLDs and TTA processors seem an extremely useful combination.

As there are no reported TTA processor FPGA implementations so far, and as FPGA seems a potential implementation technology for TTA processors, research on such implementations was necessary. This thesis presents an FPGA implementation of a TTA processor operating as a slave to a general-purpose master processor. Additionally, the

timing properties of five different TTA processors were analyzed in order to discover whether the processor architecture requires different optimizations for FPGA compared to ASIC technology.

An Altera EPXA10 development board is utilized as an example environment in the project, because the board provides a general-purpose ARM processor and an FPGA-technology programmable logic device for the TTA processor. TTA and ARM processors were enabled to communicate via a specific flag register, in which both processors have access. In addition, it was noticed that the global control unit of TTA processors requires optimizations, as the control unit contains a structural problem that extends the critical timing path. However, optimizing the architecture separately for FPGA technology is not necessary, as the critical path and structure of the FPGA implementation are identical to those obtained from ASIC implementations.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

Digitaali- ja tietokonetekniikan laitos

Niiranen, Miika Johannes: Transport Triggered Architectures on FPGA

Diplomityö: 67 sivua, 8 liitesivua

Tarkastajat: Prof. Jarmo Takala, DI Riku Uusikartano

Rahoitus: Tekniikan kehittämiskeskus TEKES, Nokia, Patria New Technologies, Vaisala, Texas Instruments (France), Instrumentointi, Elektrobit, Merivoimien tutkimuslaitos

Tietotekniikan osasto

Lokakuu 2004

Avainsanat: transport triggered -prosessoriarkkitehtuuri (TTA), FPGA

Ohjelmoitavat logiikkapiirit ovat FPGA-tekniikan ansiosta valtaamassa jalansijaa laitteistosuunnittelussa. Ne ovat hyvä vaihtoehto käytettäväksi piirien prototyypin sekä suhteellisen yksinkertaisten prosessorien valmistusteknologiana. Toisaalta, transport triggered -prosessoriarkkitehtuuri (TTA) soveltuu hyvin rinnakkaisprosessoreiden tai laitteistopohjaisten kiihdyttimien arkkitehtuuriksi, johtuen lähinnä arkkitehtuurin yksinkertaisuudesta. Näistä syistä TTA-prosessorit ja ohjelmoitavat logiikkapiirit vaikuttavat erittäin hyödylliseltä yhdistelmältä.

Koska FPGA:ta ei ilmeisestä tarkoitukseen soveltuvuudesta huolimatta ole aikaisemmin käytetty TTA-prosessoreiden toteutustekniikkana, nähtiin raportoitu kokeilu TTA-prosessorista toteutettuna FPGA-tekniikalla tarpeelliseksi. Tässä diplomityössä esitellään FPGA-toteutus eräästä TTA-prosessorista, jota käytetään yleiskäyttöisen isäntäprosessorin

rinnakkaisprosessorina. Lisäksi tutkittiin, kannattaako TTA-prosessoria optimoida FPGA-toteutustekniikalle eri tavalla kuin ASIC-tekniikalle. Tämä saatiin selville suorittamalla ajoitusanalyysit viidelle erilaiselle TTA-prosessorille.

Alteran EPXA10-kehitysalustaa käytettiin työn esimerkkiympäristönä, sillä se sisältää yleiskäyttöisen ARM-prosessorin sekä FPGA-tekniikalla toteutetun ohjelmoitavan logiikkapiirin TTA-prosessoria varten. TTA- ja ARM-prosessorit saatiin kommunikoidaan keskenään toteuttamalla erillinen lippurekisteri, johon molemmilla prosessoreilla on pääsy. Lisäksi kokeilussa havaittiin, että TTA-prosessoreiden kontrolliyksikkö vaatii optimointia, sillä kontrolliyksiköstä löytyi rakenteellinen ongelma, joka pidentää kriittistä ajoituspolkua. Arkkitehtuurin erillistä optimointia FPGA-tekniikalle ei sen sijaan tarvita, sillä ASIC-toteutusten kriittinen polku ja rakenne ovat täysin identtiset FPGA-toteutuksen kanssa.

ABBREVIATIONS

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
APEX	Advanced Programmable Embedded Matrix
ASB	Advanced System Bus
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DCT32	32-point Discrete Cosine Transform
EEPROM	Electrically Erasable Programmable Read-Only Memory
EPROM	Erasable Programmable Read-Only Memory
FPGA	Field-Programmable Gate Array
FU	Function Unit
GCU	Global Control Unit
GU	Guard Unit
HDL	Hardware Description Language
I/O	Input / Output
IDCS	Institute of Digital and Computer Systems

IDU	Instruction Decode Unit
IFU	Instruction Fetch Unit
IRQ	Interrupt Request
JTAG	Joint Test Action Group; IEEE standard 1149.1
LE	Logic Element
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
PC	Program Counter
PLD	Programmable Logic Device
RA	Return Address
RAM	Random Access Memory
RF	Register File
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Level
SRAM	Static Random Access Memory
SVTL	Semi Virtual-Time Latching
TTA	Transport Triggered Architecture
UART	Universal Asynchronous Receiver / Transmitter
VHDL	Very high speed integrated circuit Hardware Description Language

1. INTRODUCTION

Transport triggered architectures (TTAs) are a good option for traditional very long instruction word (VLIW) processor architectures, especially in custom hardware accelerator design. The data path in VLIW architectures can become very complex, in particular when scaled to high performance. Fortunately, with TTAs, it is possible to replace hardware complexity with compiler effort [2, p. 15]. TTAs reverse the idea behind the VLIW architecture. In TTA, the instruction word expresses only the data transfers performed. Operations occur as a side effect of transfers, which are sometimes referred to as “moves”. In contrast, in VLIW architectures, the instruction word gives the operations to be performed. In addition, TTAs enable the use of a simple register bypass network, in contrast to VLIW architectures. Register bypassing denotes using an operation result directly in the next operation. [1, pp. 101]

Recently, transport triggered processor architectures and automatic processor generation tools have been among the most current research subjects at the Institute of Digital and Computer Systems (IDCS) at Tampere University of Technology. The processor generator developed at IDCS produces a TTA processor optimized for a specific application [2]. All TTA processor realizations reported so far are based either on full-custom or application specific integrated circuit (ASIC) technology. However, as the programmable logic devices (PLDs) are improving, mainly due to field-programmable gate array (FPGA) technology, an increasing share of the design projects started nowadays are targeted to FPGA. Therefore, it becomes topical to evaluate how a TTA processor can be incorporated into FPGA technology.

TTA processors resemble rather hardware accelerators than general-purpose processors. Therefore, a case in which a TTA processor is controlled by a master processor also needs to be implemented and evaluated. ARM922T is a popular general-purpose reduced instruction set computer (RISC) processor, which is incorporated in an Altera EPXA10 development board, along with an APEX 20KE-compliant FPGA programmable logic device [7, pp. 11-12]. As the IDCS had an EPXA10 development board available, it was selected to be used as an environment for the FPGA-based TTA hardware accelerator design.

The basic case, in which a RISC processor provides the input data to the TTA processor and initiates the calculation, has been preliminarily demonstrated. More complex cases, such as slave processor instruction memory reprogramming and interrupt generation, were not evaluated. These cases, along with FPGA timing analysis and project documenting, are the main topics of this thesis work. In addition, different TTA processors are analyzed in order to discover possible problems related to FPGA-based TTA implementations.

This thesis describes the basic concepts that enable a TTA processor to be run on a PLD as a slave processor to a universal master processor. First, the EPXA10 development board is introduced in Chapter 2. FPGA design recommendations and the communication between a master and a slave processor are discussed in Chapter 3. Chapter 4 presents the TTA processor employed in the design. An FPGA-based implementation of a TTA processor as a discrete cosine transform (DCT) hardware accelerator is introduced in the first sections of Chapter 5. The results obtained from the general TTA timing analysis are presented in the last section of Chapter 5. Finally, Chapter 6 briefly concludes the most important observations and results of the study.

2. ALTERA EXCALIBUR DEVELOPMENT BOARD

Altera Excalibur EPXA10 development board was selected as a platform for the FPGA-based TTA design due to the fact that it contains an ARM922T processor and a PLD with enough logic elements for a design of this magnitude. The IDCS also has different versions of the Excalibur development boards. The other options were EPXA1 and EPXA4, although their memory capacities might not have been sufficient for the FPGA-based TTA processor design. Figure 2.1 shows the contents of and differences between those devices.

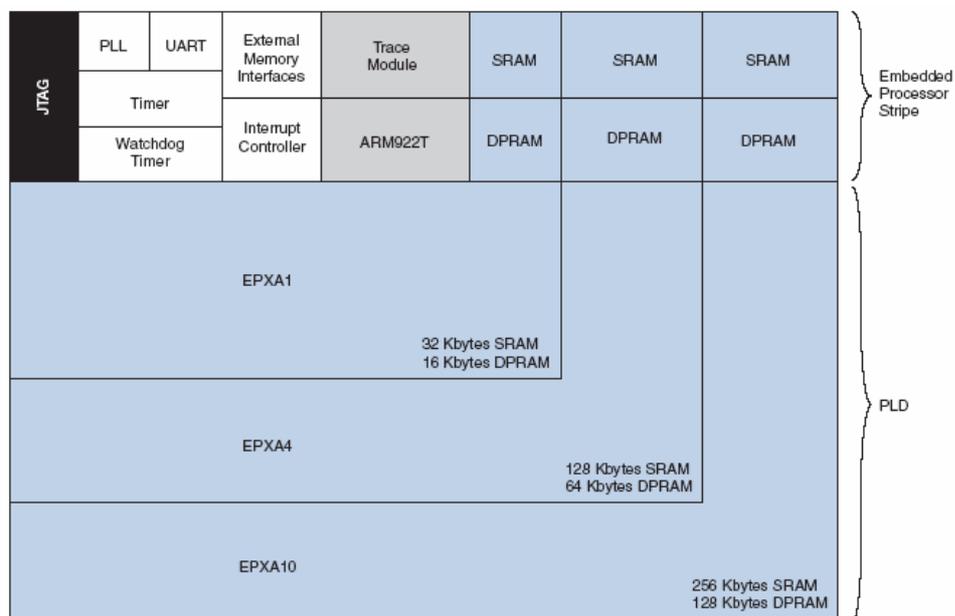


Figure 2.1: EPXA1, EPXA4, and EPXA10 contents [7, p. 13].

In this chapter, the most essential areas of the development board, i.e., the parts that are necessary for this thesis, are discussed in detail. Section 2.1 presents the ARM922T

processor. Section 2.2 introduces the advanced microcontroller bus architecture (AMBA) compliant advanced high-performance bus (AHB), which is used to connect the PLD side devices to the embedded ARM processor stripe. The embedded stripe, some of the devices it contains, and its PLD interfaces are detailed in Section 2.3. Section 2.4 first discusses programmable logic devices in general, and finally focuses on the PLD of the EPXA10 development board. The memory regions of the EPXA10 development board are introduced in Section 2.5.

2.1 ARM922T Processor

ARM922T is a member of the ARM9TDMI family of microprocessors. It contains the processor core, separate 8K data and instruction caches, and a memory management unit (MMU). The ARM922T processor is a powerful 32-bit general-purpose RISC processor operating at a maximum core frequency of 200 MHz [7, p. 11].

The ARM922T processor supports the ARM debug architecture and includes logic to assist in both hardware and software debugging. The processor also includes support for coprocessors, exporting the instruction and data buses along with simple handshaking signals. The processor interfaces with the rest of the system over unified data and address buses. This interface enables an implementation of either AMBA AHB or AMBA advanced system bus (ASB) scheme as a fully compliant AMBA bus master, or as a slave for production test. [8, p. 24]

2.2 Advanced Microcontroller Bus Architecture

The Advanced Microcontroller Bus Architecture specification, developed by ARM, Ltd., defines an on-chip communication standard for designing high-performance embedded microcontrollers. The AMBA specification defines three different bus types: advanced high-performance bus, advanced system bus, and advanced peripheral bus (APB). The AMBA AHB is designed for high clock frequency system modules and it supports efficient connection of processors, on-chip memories, and off-chip external memory interfaces. The AMBA ASB is an alternative system bus for applications in which the

high-performance properties of the AHB are not required. The AMBA APB is intended for low-power peripherals. The AMBA AHB was selected to be applied to the FPGA-based TTA hardware accelerator design and, due to its significance to the design, is discussed more in the following subsections.

2.2.1. AMBA Advanced High-Performance Bus

A typical AMBA AHB system design contains the following components: AHB masters, AHB slaves, an AHB arbiter, and an AHB decoder. AHB masters are able to initiate transfers. AHB slaves only respond to read or write operations within a given address-space range, i.e., they do not initiate any transfers. An AHB arbiter ensures that only one bus master at a time is allowed to initiate data transfers. An AHB decoder decodes the address of each transfer and provides a select signal for the slave involved in the transfer [8, p. 1-7]. Figure 2.2 shows an example of a multi-master AHB design. In our implementation, however, neither an arbiter nor a decoder is required because the system contains only a single master and a single slave, i.e., the ARM processor stripe and the memory controller, respectively.

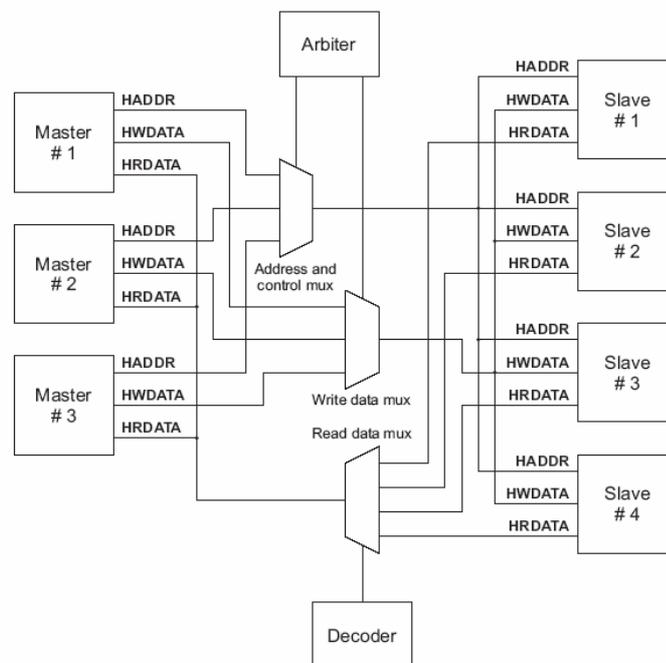


Figure 2.2: Example of a multi-master, multi-slave AHB design [8, p. 3-4].

All AHB-related signals in Figure 2.2 are denoted with an ‘H’ in the beginning of a signal name. The three most important AHB signals, HADDR, HWDATA, and HRDATA, are displayed in Figure 2.2. These signals carry the address, write data, and read data, respectively. The rest of the AHB signals are described in the following subsections.

2.2.2 AMBA AHB Master Interface

The AHB master interface is presented in Figure 2.3. AHB masters initiate transfers by issuing control signals and an address to the bus. An arbiter, where available, decides whether the master is allowed to access the bus. The AHB master interface is the most complex of all AHB interfaces. Fortunately, designers typically use pre-designed master blocks, and therefore avoid managing the master interface manually. [8, p. 3-49]

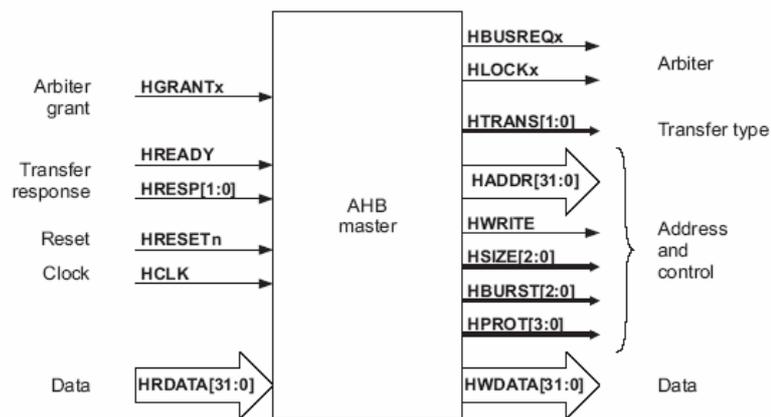


Figure 2.3: AHB master interface [8, p. 3-49].

The master is given a permission to access a bus by HGRANTx signal received from an arbiter. Slaves respond to transfers using HREADY and HRESP signals. These signals are explained in Section 2.2.3. Signal HRDATA carries the data read from a slave.

An AHB master can request an access to the bus by outputting the signal HBUSREQx to a bus arbiter. The signal HLOCKx enables a master to request a locked transfer. The HADDR signal contains the read / write address. HWRITE indicates whether a transfer will be a read or a write operation, i.e., selects the transfer direction. HSIZE indicates the size of a transfer. The size can vary from 8 to 1024 bits in powers of two. HBURST indicates whether a transfer is a part of a burst. HPROT is a protection control signal which is

intended for use by any module that wishes to implement some level of protection. The signal HTRANS indicates the current state of the transfer. It has four different values, which are IDLE, SEQUENTIAL, NONSEQUENTIAL, and BUSY. Standard transfers are of the NONSEQUENTIAL type, in which the address of a transfer is independent of previous addresses. [4, p. 2-3]

2.2.3 AMBA AHB Slave Interface

Figure 2.4 introduces the AHB slave interface. A slave responds to transfers initiated by masters in the system. Slaves use the HSELx signal received from an arbiter to determine whether they should respond to a transfer.

A slave receives the address and control signals issued by a bus master. These signals were already explained in Subsection 2.2.2. The slave responds to transfers by issuing HREADY and HRESP signals. The former indicates the master that a transfer has been completed successfully. Alternatively, the HREADY signal can be used to extend transfers to last for multiple clock cycles. Note that, contrary to as shown in Figure 2.4, slaves require HREADY both as an input and as an output signal. The HRESP signal provides additional information on the status of a transfer. Four different responses are provided: OKAY, ERROR, RETRY, and SPLIT. [8, p. 2-3]

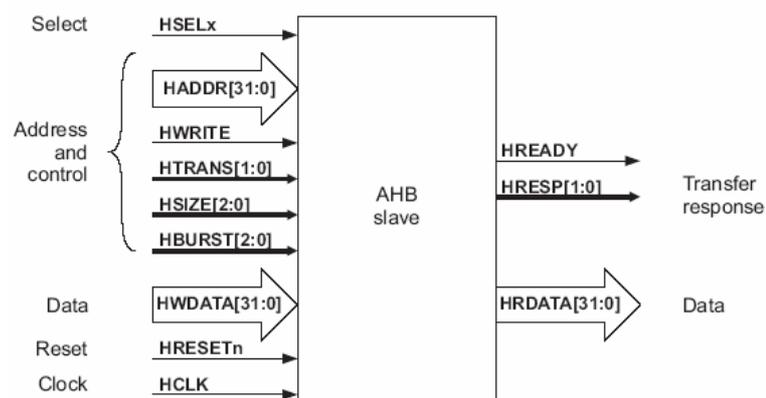


Figure 2.4: AHB slave interface [8, p. 3-45].

2.2.5 AMBA AHB Basic Transfer

Each AHB transfer consists of two phases: an address phase and a data phase. The address phase is reserved for transferring the address and control information, i.e., signals HADDR, HSEL, HWRITE, HRESP, HTRANS, and HSIZE, to the bus. The data phase is for transferring the actual data. The data phase can be, when necessary, extended to more than one clock cycle by keeping the HREADY signal low. It is important to distinguish between these two phases when designing an AHB-compliant slave module. Figure 2.5 presents the behaviour of AHB signals during the two transfer phases.

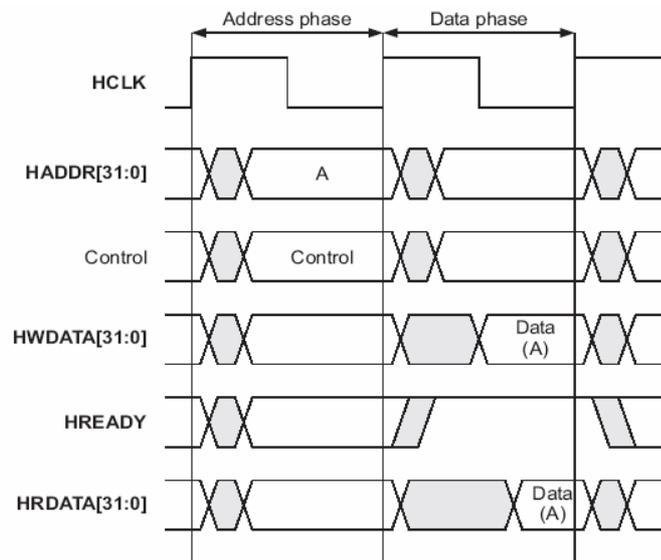


Figure 2.5: Basic AHB transfer [8, p. 3-6].

During the address phase, an arbiter enables a bus slave to latch the address and control information. During the data phase, the slave, for which the transfer is intended to, responds to the initiating master accordingly.

2.3 Embedded Stripe

As shown in Figure 2.1, the EPXA10 device is composed of two parts: an embedded processor stripe and a PLD. The term “stripe” refers to the region of the die containing the ARM processor and hard peripherals, such as timers and phase-locked loops (PLL). The

stripe is interfaced and configured to suit different applications by instantiating the Excalibur megafunction using the Megawizard Plug-In Manager in the Quartus II design software. The Quartus II software is provided with the development board kit.

The embedded processor stripe uses two AHB buses, AHB1 and AHB2, as its communication medium. This hierarchy is presented in Figure 2.6. Figure 2.1 shows the embedded stripe contents. Both buses use 32-bit address and data signals. AHB1 has only one bus master, the ARM processor. For clarity, bus masters are presented in black and bus slaves in grey in Figure 2.6. Processor-specific slaves, such as the interrupt controller, are local to AHB1. Embedded stripe memories are also local to AHB1, in order to allow the main processor fast access to the memories. Apart from the dual-port SRAM interface, which is a standard interface, the embedded stripe and the PLD bridge interfaces follow the AHB bus standard.

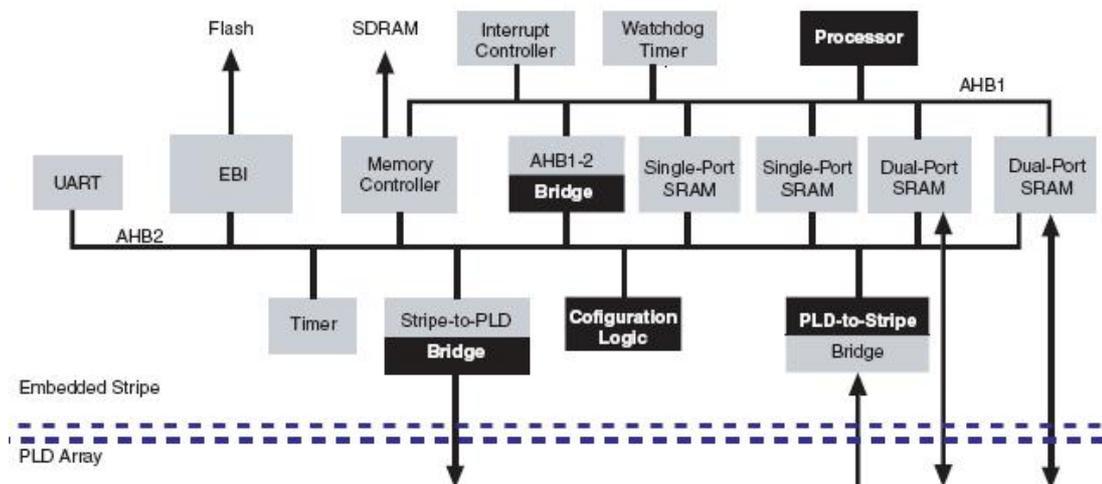


Figure 2.6: Embedded Stripe Bus Architecture [7, p.27].

Any transaction which, after decoding, is not intended for any slave in AHB1 is routed to the AHB1-2 bridge. The bridge enables the processor to access AHB2 slaves, including the stripe-to-PLD bridge. AHB2 has three bus masters, which are the AHB1-2 bridge, PLD-to-stripe bridge and configuration logic. The PLD-to-stripe bridge allows masters in the PLD to access resources in the embedded stripe. The bridge is accessed via the slave port on the embedded stripe and appears to the user as a conventional AMBA AHB slave. The configuration logic is responsible for transferring configuration data to the PLD array and setting up the system so that the embedded processor can boot. [7, pp. 28, 149]

When communication with PLD-side slaves is required, transfers are directed through the stripe-to-PLD interface to the appropriate slaves. The user must enable the stripe-to-PLD bridge and set the PLD base address so that the software that is being run on the ARM processor is able to access the PLD at the specified address range.

Both AHBs and the PLD side have their own clock domains, i.e., they use separate clocks. However, because AHB1 clock frequency is exactly twice the AHB2 frequency, the buses are synchronized. [7, p. 33]

Besides the bridges that are extremely important in this design, the embedded stripe contains several other useful devices, but they are not relevant to this thesis and thus are not described in detail. However, the interrupt controller provided by the embedded stripe needs to be introduced due to the fact that interrupts aid in implementing efficient software.

The interrupt controller on the embedded stripe provides a simple but flexible software interface to the interrupt system. There are several interrupt sources, such as an external interrupt pin and ten interrupts from the devices within the embedded stripe, but the most important interrupt source for the current application is the 6-bit interrupt bus `INT_PLD[5..0]` from the PLD-to-stripe interface. This bus enables the PLD devices, i.e., the blocks incorporated into the PLD, to send interrupt requests (IRQ) to the master processor. The interrupt controller provides three operating modes to control how the six `INT_PLD`-signals from the PLD are treated. By default, the signals are treated as six individual IRQs. Other options are a single IRQ together with a 5-bit priority value, and a single request using a 6-bit priority value. [7, pp. 124-128]

2.4 Programmable Logic Devices

A programmable logic device is a digital, user-configurable integrated circuit used to implement custom logic functions. PLDs are sometimes referred to as field-programmable devices (FPDs) for historical reasons, but this thesis applies the term “programmable logic device” instead, thereby avoiding inconsistency with literature provided by Altera Corporation [6], [7], [11]. PLDs can implement any Boolean expression or registered

function with built-in logic structures. PLDs are often utilized in prototyping and verification. In addition, PLDs are common platforms for hardware accelerators used in embedded system design. There are several different PLD technologies available; field-programmable gate array and complex programmable logic device (CPLD) are the two most common examples. [10, p. 2]

CPLDs have been used in older Altera development boards, such as in MAX7000 and MAX9000 platforms. However, as expanding the logic capacity of a CPLD to high densities is somewhat difficult, a different approach is required. As the only type of PLDs that supports very high logic capacity, FPGAs have been responsible for a major shift in the way digital circuits are implemented. [10, p. 7]

FPGAs comprise an array of uncommitted circuit elements, called logic blocks or logic elements (LEs), and interconnect resources, i.e., wire segments and programmable switches between the logic blocks. The blocks perform custom combinational or synchronous logic functions, and the interconnection wires connect these blocks appropriately. In addition, an FPGA contains I/O blocks that enable the FPGA to interface with other devices. FPGAs may also contain additional blocks, such as multipliers and embedded memory. An illustration of a typical FPGA architecture appears in Figure 2.7.

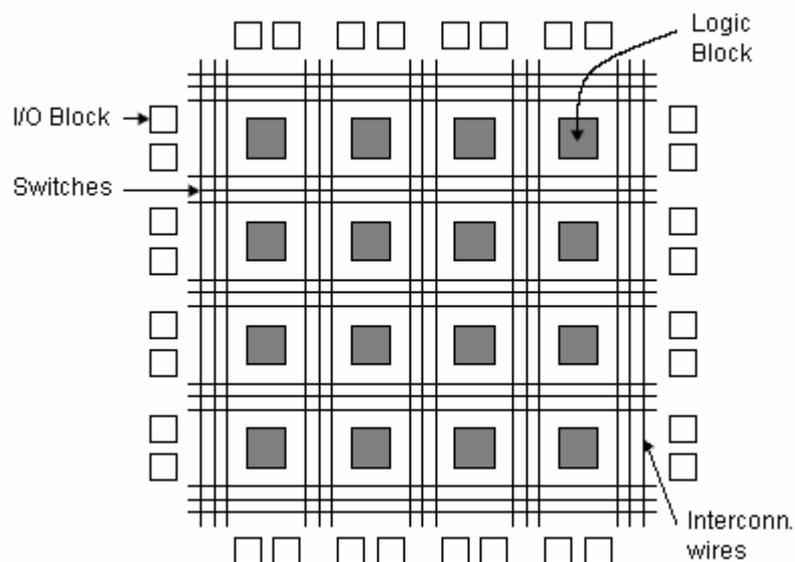


Figure 2.7: Structure of a field-programmable gate array [10, p. 7].

The key to user-customization of FPGAs are, along with logic blocks, user-configurable switches that interconnect the wires connecting the logic blocks. There are several types of these switches; erasable programmable read-only memory (EPROM) and electronically erasable programmable ROM (EEPROM) transistors, fuses, static RAM (SRAM) cells, and antifuses, for instance. Although there is no technical reason why EPROM or EEPROM transistors could not be used in FPGAs, current commercial FPGA products are based on either SRAM or antifuse technologies. [10, p. 9]

An example of SRAM-controlled switch usage is illustrated in Figure 2.8, showing two applications of SRAM cells: controlling the gate nodes of pass-transistor switches, and driving the select inputs of multiplexers connected to logic block inputs. SRAM-controlled switches operate similarly to one-bit memories, as they hold the programmed connections until the chip is reprogrammed or as long as the power stays on, i.e., the connections are volatile. Figure 2.8 gives an example of the connection of one logic block, represented by the AND-gate in the upper left corner, to another block through two pass-transistor switches, and a multiplexer, all of which are controlled by SRAM cells. Whether an FPGA uses pass-transistors, multiplexers, or both, depends on the particular product. [10, p. 9]

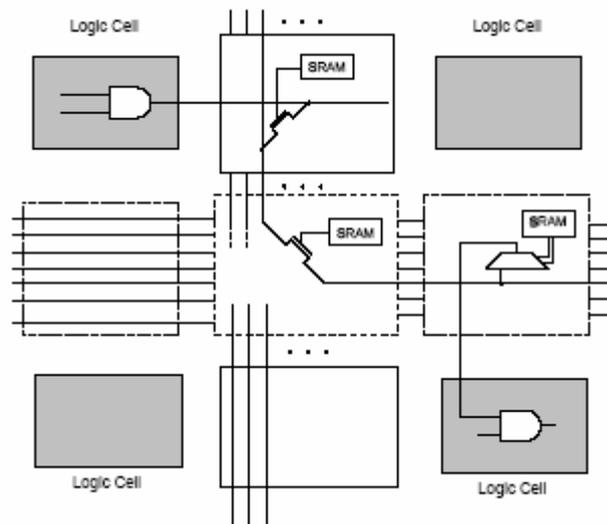


Figure 2.8: Programmable SRAM switches [10, p. 10].

In addition to SRAM, the other type of programmable switch used in FPGAs is the antifuse. Antifuses are originally open circuits which become low-resistance connections

when programmed. Antifuses are suitable for FPGAs because they can be built using modified CMOS technology. The drawback in using antifuses is that the switches can be programmed only once, i.e., the FPGA is practically not reprogrammable. [10, p. 11]

FPGAs are generally slower and draw more power than their application-specific integrated circuit (ASIC) counterparts. The relatively low performance of FPGAs is partly due to the delays caused by complex multi-level routing. However, FPGAs have several advantages such as shorter time-to-market and lower development costs when quantities are relatively small.

The EPXA10 board contains an EPXA10F2010C1 programmable logic device utilizing APEX 20KE architecture and SRAM-based FPGA technology. APEX 20KE is an Altera embedded programmable logic device family based on the advanced programmable embedded matrix (APEX) multicore architecture, which integrates look-up table (LUT) logic, product-term logic, and memory in a single device. In APEX 20K devices, signal interconnections to and from device pins are provided by a series of fast, 4-layered row and column channels that run the entire length and width of the device. APEX 20KE devices are a superset of APEX 20K devices, and include additional features such as advanced I/O standard support, content-addressable memory (CAM), additional global clocks, and enhanced clock circuitry. In addition, APEX 20KE devices extend the logic capacity to 1.5 million gates. [11, p. 6]

Designs can be programmed into the PLD via the Joint Test Action Group (JTAG) IEEE 1149.1 interface provided by the development board. The JTAG interface can also be used for debugging. The PLD can be configured to contain additional soft-core extensions, such as Ethernet media access control, universal asynchronous receiver/transmitter (UART), peripheral component interconnect or any other intellectual property core [7, p. 16]. In addition, Altera provides several user-customizable IP Megafunctions to the designer.

2.5 Development Board Memory Regions

The EPXA10 development board provides a 32-bit memory address space. Memory addresses must be allocated without overlap in any resource, otherwise undefined results will occur. The default settings and the available options are displayed in Table 2.1.

The memory map control registers are by default located at the address shown in Table 2.1. These registers define the memory ranges and base addresses of the peripherals. [7, p. 100]

resource	default address	default space	max space
registers	0x7FFFC000	16K	16K
SRAM0	0x00000000	128K	128K
SRAM1	0x00020000	128K	128K
DPRAM0	-	-	64K
DPRAM1	-	-	64K
EBI0 (FLASH)	0x40000000	4M	32M
EBI1	-	-	32M
EBI2	-	-	32M
EBI3	-	-	32M
PLD0	0x80000000	64K	2G
PLD1	-	-	2G
PLD2	-	-	2G
PLD3	-	-	2G

Table 2.1: Development board memory regions.

There are two blocks of single-port SRAM on the board. Both are accessible to the AHB masters (AHB1 and AHB2) via an arbitrated interface within the memory. Both blocks are independently arbitrated, which allows one block to be accessed by one bus master while the other block is accessed by the other bus master. [7, p. 47]

The two 64K dual-port SRAM (DPRAM0 and DPRAM1) blocks are accessible to the AHB1 and AHB2 masters, and they both have optional PLD interfaces through which they can be configured for access by AHB masters. It is also possible to build deeper and wider memories by using both I/O ports of the dual-port memories and multiplexing the data outputs within the embedded stripe. [7, p. 47]

The expansion bus interface (EBI) must be enabled when the processor is configured to boot from flash memory, because the memory is connected to the processor via EBI0. The user must set the EBI0 address space large enough to store the program codes, otherwise an error results when generating the flash programming file.

The PLD regions select the address space of the stripe-to-PLD bridge. When an AHB transfer address belongs to the PLD address space, the transfer is directed to the stripe-to-PLD bridge. The transfer is then forwarded from the bridge master interface on the PLD to the PLD side slaves.

3. SLAVE PROCESSORS ON FPGA

This chapter introduces the principles that form the basis for the TTA hardware accelerator FPGA implementation which is presented in Chapter 5. First, we discuss some general design recommendations for FPGA devices in Section 3.1. Communication between a general-purpose master and a TTA slave processor is introduced in Section 3.2.

3.1 FPGA Design Recommendations and Notes

Today's FPGA applications have almost reached the complexity and performance requirements of ASICs. In the development of such complex designs, good design practices have an enormous impact on device verification, re-use, performance, logic utilization, and reliability. Well-coded designs behave in a predictable and reliable manner even when re-targeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and ASIC implementations, for both prototyping and production. For optimal performance and better time-to-market, the designer should understand the impact of synchronous design practices, follow the generally recommended design techniques including hierarchical design partitioning, and take advantage of the architectural features in the targeted device. [16, p. 6-1]

In the past, designers often used asynchronous techniques, such as ripple counters or pulse generators in PLD designs, in order to save device resources. Asynchronous design techniques have inherent problems such as the dependence on propagation delays in a device, which can result in incomplete timing constraints and possible glitches. Because today's FPGAs provide large quantities of high-performance logic resources, registers, and

memory, the resource and performance trade-offs have changed and the old problematic asynchronous design techniques should be discarded.

Several designs use delay chains to generate pulses. These techniques are asynchronous and may seriously compromise the stability of the design. When a pulse generator is required, e.g., for sending an interrupt request, it should be implemented as in Figure 3.1, using a series of two d-flip-flops. This method is fully synchronous and predictable, always producing a pulse the width of which is equal to one clock period.

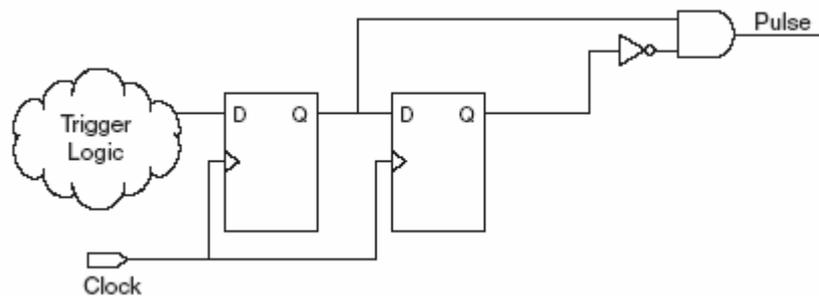


Figure 3.1: Recommended pulse-generation technique [16, p. 6-6].

Latches can be problematic in designs, as they do not belong to the synchronous design scheme. A latch holds the value of a signal until a new value is assigned. Although latches are memory elements, they are fundamentally different from registers. When a latch is in feed-through or transparent mode, there is a direct path between the data input and the output. Glitches on the data input can pass through to the output. In some ASIC technologies, latches add less delay and can be implemented using less silicon area than registers. However, FPGA architectures are based on registers. Therefore, in FPGA devices, latches actually use more logic resources and lead to lower performance than registers. [16, p. 6-7]

Latches can be inferred from HDL code even when the designer does not intend to use a latch. For example, when a CASE or IF statement does not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. In other words, omitting the final ELSE or WHEN OTHERS clause in an IF or CASE statement can generate a latch. “Don’t care” assignments on the default conditions, however, tend to prevent latch generation. Synthesis software generally treats unknowns as “don’t care” conditions to optimize logic. For the best logic optimization, the default

CASE or final ELSE value should be assigned to “don’t care” instead of a logic value. [16, p. 7-31]

Gated clocks can be a powerful technique to reduce power consumption. When a clock is gated, both the clock network and the registers driven by the clock stop toggling when not needed, thereby eliminating unnecessary power consumption. However, gated clocks are not a part of the synchronous design scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks also contribute to clock skew and make device migration difficult. In addition, these clocks are sensitive to glitches, which can cause a design failure. In the technique shown in Figure 3.2, a register generates the clock enable signal to ensure that it is free of glitches. The register that generates the enable signal is triggered on the falling edge of the clock. This technique assures that only one input of the AND-gate changes at a time, which avoids glitches in the output.

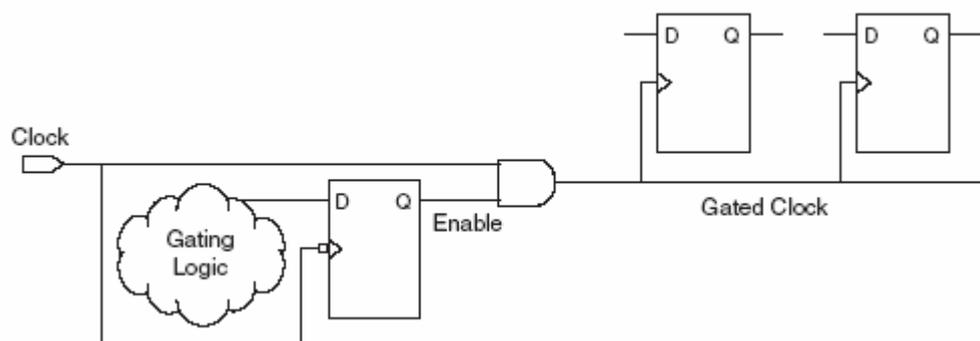


Figure 3.2: Recommended clock-gating technique [16, p. 6-12].

In ASIC designs, balancing the clock delay as it is distributed across the device is often very important. Modern FPGAs, on the other hand, provide device-wide global clock routing resources and dedicated inputs, so there is no need to manually balance delays on the clock network. The designer should always use the low-skew, high-fan-out, dedicated routing provided by FPGA, whenever it is available. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are typically organized into a hierarchical clock structure that allows many clocks with low skew and delay in each device region. To take full advantage of these special routing resources, the sources of clock signals in a design should drive only the clock input ports of registers. In certain devices, if a clock signal

feeds the data ports of a register, the signal may not be able to use the dedicated routing, which can lead to decreased performance. [16, p. 6-14]

In FPGA technology, routing delays dominate logic propagation delays. This is in contrast to ASIC technology and might cause problems with asynchronous designs. In addition, when more resources are mapped to an FPGA, the routing becomes more complex and the distance between interconnected logic elements can increase, which results as longer critical paths and lower attainable clock frequency.

3.2 Communication between a Master and a TTA Slave Processor

Co-operation of two (or more) processors is based on successful messaging between the processors. A master processor usually initiates transfers, while slave processors respond to these transfers. Some designs may require synchronous communication, whereas asynchronous communication may suffice for other designs.

Using flags is a simple, yet fairly efficient, way of communication. Flags are bits indicating the status of a certain resource or process. When flags are used, a flag register is required to store the values of the flag bits. The flag register must obviously be accessible to both (all) processors in the system. However, in order to efficiently use the register to inform the master processor of an event, the use of interrupt requests is required to notify the processor of a change in the register contents. Otherwise the master processor would have to continuously poll the register, thus wasting processor time.

The memory structure of a system should be carefully planned, as every superfluous data transfer degrades overall system performance. An optimal solution is to store data, which is common to all processors, in a location in which both the master and the slaves have direct access. This implies that the processors should have a common memory space, in which all the data required by all processors is stored.

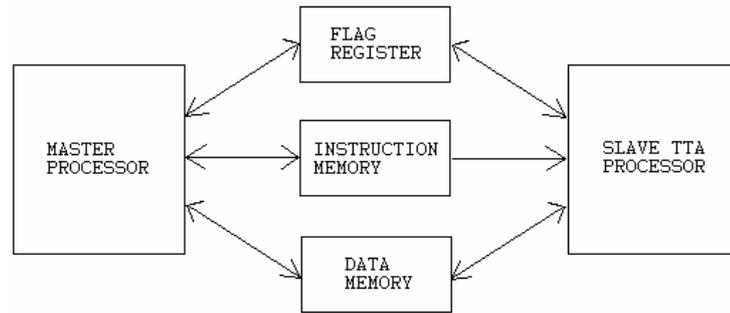


Figure 3.3: Design elements and communication between the processors.

Figure 3.3 presents a conceptual view of a design, in which a TTA processor is used as a hardware accelerator. The design elements, i.e., the master processor, flag register, instruction memory, data memory, and the slave TTA processor, are connected according to possible data transfers. These connections are shown with arrows in Figure 3.3. All transfer paths are bidirectional, except the one between the instruction memory and the slave TTA processor. This path does not enable the slave processor to modify its own instruction memory. Note that both memories are common to both processors, as recommended earlier in this section.

As all TTA processors can be equipped with an option to use a global lock signal for halting the pipeline of the processor, as well as a reset signal for initializing the processor, controlling a TTA processor is relatively simple. In addition, TTA processors can be configured to write certain values to the flag register after specified events.

4. TTA DCT32 PROCESSOR

A TTA processor customized for 32-point discrete cosine transform (DCT32) was selected as an example processor for this project, as one was already available and documented in [2]. This DCT32 processor was generated using a tool known as MOVE Processor Generator. The processor consists of four function units (FUs), eight register files (RFs), an interconnection network, and a global control unit (GCU). The original version of the processor had two data buses, whereas the amount of buses in the newest version has been increased to five. The examples and figures are based on the two-bus version of the processor in order to keep the figures simple and illustrative. The same information, however, applies to the five-bus version of the processor.

The general structure of the TTA DCT32 processor is discussed in Section 4.1. All the processor components are individually introduced and explained in Sections 4.2 - 4.4.

4.1 Processor Structure

The DCT32 processor has been designed using the standard transport triggered architecture. The processor consists of function units and register files connected by five 32-bit data buses. Figure 4.1 shows an older two-bus version of the processor. The processor is fully connected, i.e., all the FUs and RFs are connected to all the data buses. Besides the FUs and RFs, the processor needs a global control unit to perform necessary control tasks, such as instruction fetching, instruction decoding, program counter updating, and transfer guarding.

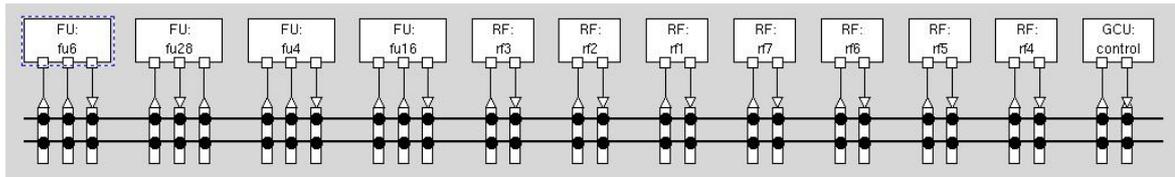


Figure 4.1: TTA Processor Designer view of a two-bus TTA DCT32 processor.

A logical view of the common structure of a TTA processor is presented in Figure 4.2. The processor is composed of different building blocks connected to each other by an interconnection (IC) network. The network also connects the control signals, i.e., the load signals, operation codes, lock signals, and so forth. The DCT32 processor contains all the block types displayed in Figure 4.2. Each block is briefly discussed later in this chapter.

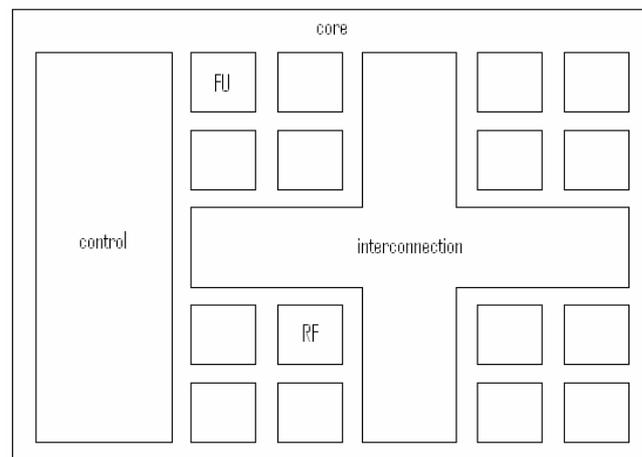


Figure 4.2: Logical structure of a TTA processor core.

4.2 Global Control Unit

A global control unit of a two-bus TTA processor is depicted in Figure 4.3. The GCU is composed of four functional units: an instruction fetch unit (IFU), an instruction decode unit (IDU), an immediate unit, and a guard unit.

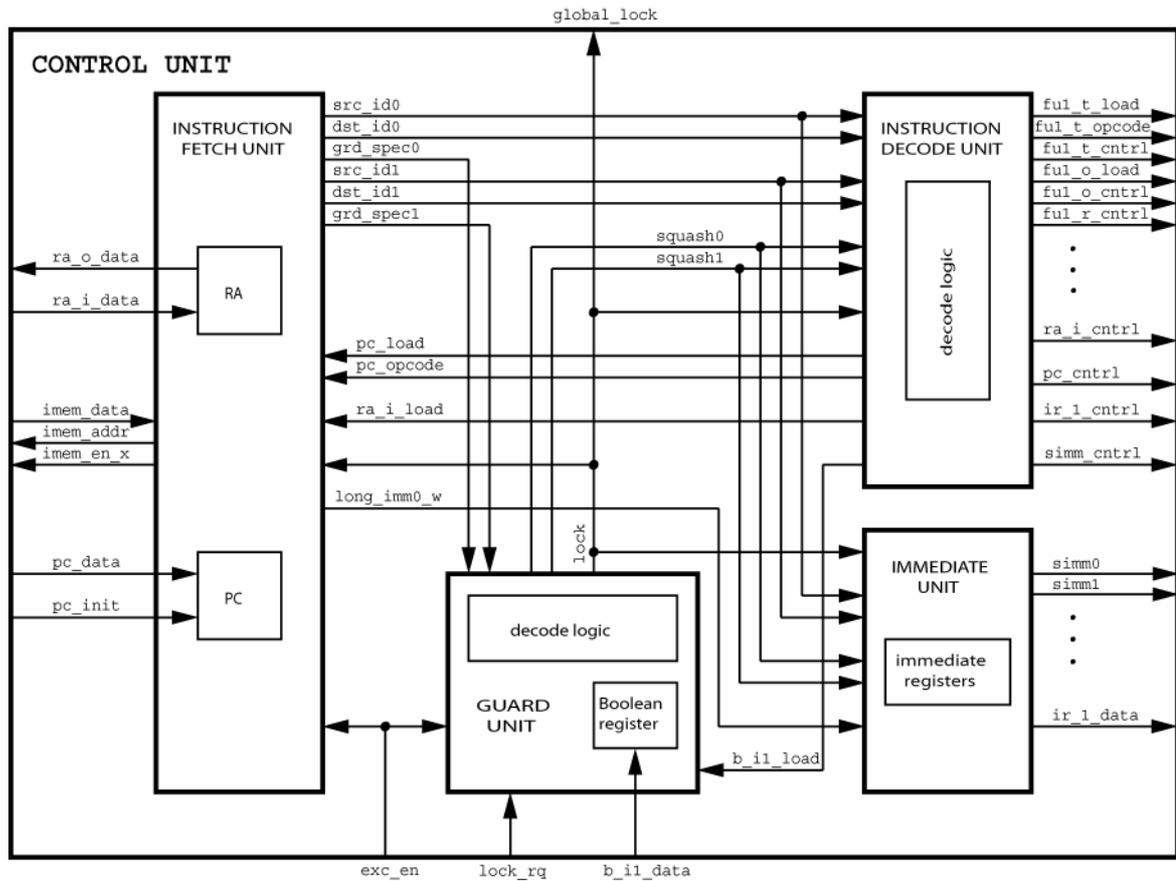


Figure 4.3: Global control unit of the TTA DCT32 processor.

Each unit will be separately introduced in Subsections 4.2.1 - 4.2.4. Each subsection also contains at least one schematic of the unit under discussion. The schematics illustrate the units of a two-bus TTA processor, although the TTA DCT32 processor contains five data buses. In addition, these schematics typically show only a limited amount of resources, e.g. control signals or sockets for only one function unit instead of every control signal or socket in the processor. The clock and reset signals are also omitted from the figures.

4.2.1 Instruction Fetch Unit

The IFU fetches instructions from the instruction memory. The IFU contains a program counter (PC) register which is incremented every clock cycle, unless overridden by a direct assignment from parallel software code (`pc_data`) when a function call or jump situation occurs. In addition, the PC register can be loaded an initial value (`pc_init`). The current PC value is used as an address when accessing the instruction memory. When a jump occurs, only the PC register contents are changed, whereas a call requires the

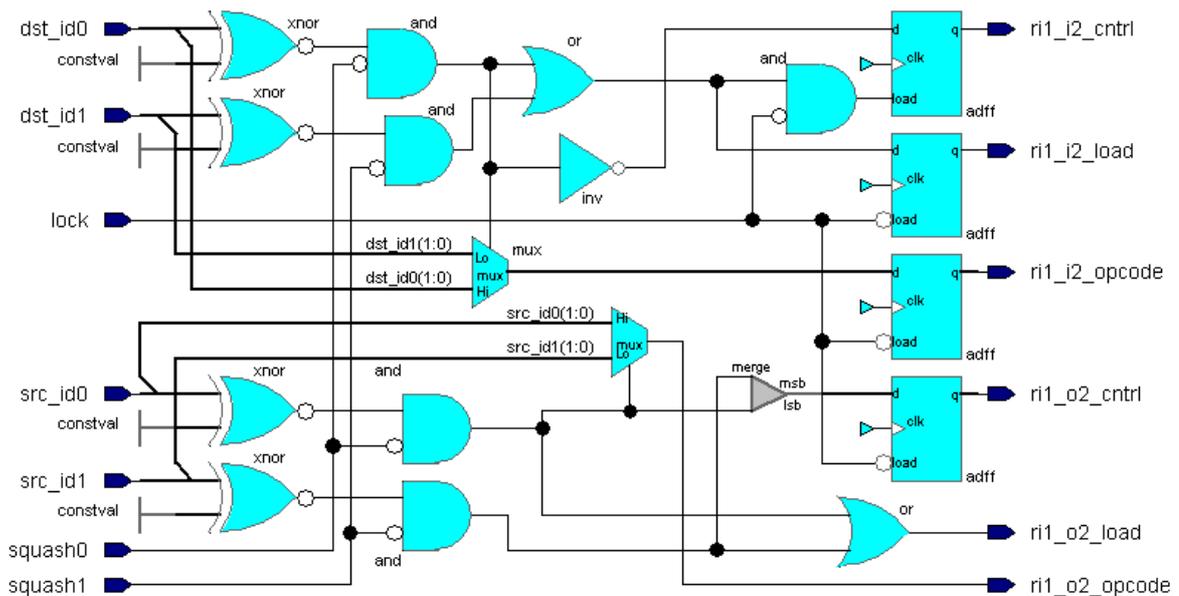


Figure 4.7: Instruction decode logic for a RF.

The instruction decode logic in Figure 4.6 has control signals for only one standard-interfaced function unit, FU1. Similarly, only one register file, RF1, is controlled by the decode logic depicted in Figure 4.7. In addition to those signals shown in Figures 4.6 and 4.7, a valid IDU requires at least control signals for the instruction fetch unit (program counter control signals `pc_load` and `pc_opcode`, return address register control signals `ra_i_load` and `ra_i_cntrl`). These signals are generated similarly to the FU control signals in Figure 4.6.

The destination and source IDs are compared against hardwired constant values assigned to the FU sockets. These constant values are shown as `constval` in Figures 4.6 and 4.7. Each I/O socket has been assigned a unique value, a socket ID. If a destination ID is equal to a hardwired socket ID, such as `fu1_t` socket ID, the transfer is intended for the corresponding FU input, and the logic enables the appropriate control signals so that the FU will successfully receive the transfer.

4.2.3 Immediate Unit

Parts of the source IDs can also be used to carry short immediate values. The remaining instruction bits are reserved for a long immediate value. The immediate unit handles and stores these immediate values. The immediate unit is shown in Figure 4.8.

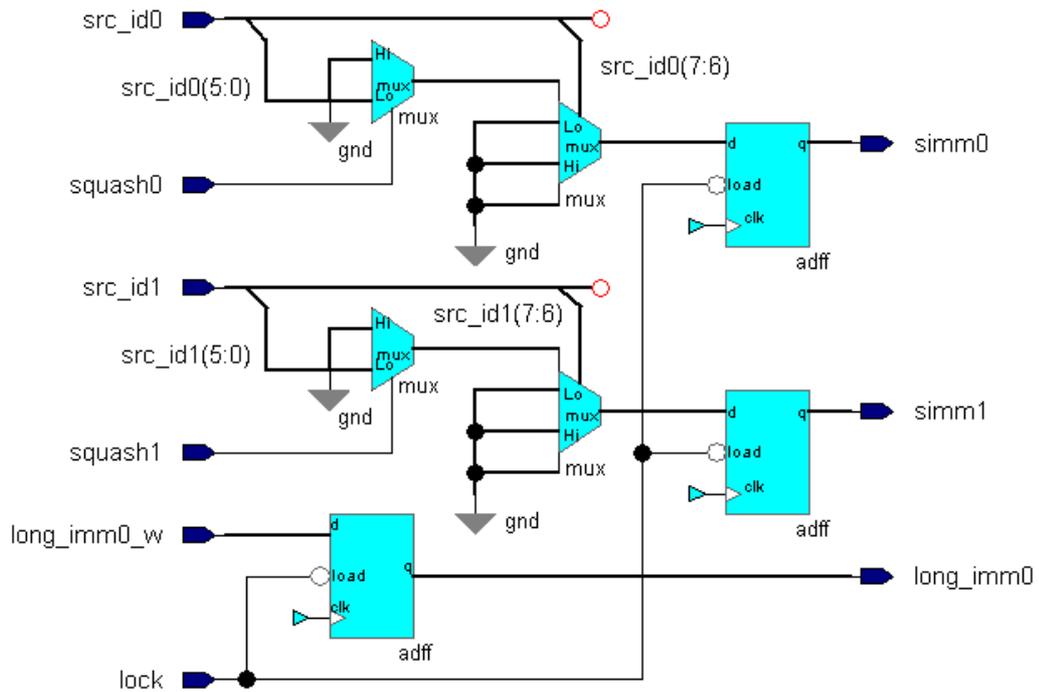


Figure 4.8: Immediate unit of a two-bus TTA processor.

In Figure 4.8, the source IDs `src_id0` and `src_id1` are 8-bit values. The length of source IDs, however, varies in different TTA processors. In this example, the two most significant bits of the source IDs are reserved for selecting the source of the short immediate (`simmm`), resulting in a 6-bit short immediate value. If a short immediate value is present, and the transfer to the current bus is not cancelled with the squash signal, the value is registered and transferred to the output (`simmmX`, where `X` denotes a natural number) of the appropriate data bus. As there is a fixed long immediate field in the instruction word, the long immediate value only needs to be registered. The signal `long_imm0_w` is merely a bus connected to the long immediate field of the instruction word.

4.2.4 Guard Unit

The guard unit evaluates guard expressions from the guard specifier fields (`grd_specX`) in the instruction. The unit contains hardware that supports conditional execution. When a guard expression is evaluated as false, the corresponding transfer on a bus is cancelled. Blocked transfers are indicated to the IDU by squash signals. In addition, the guard unit contains Boolean registers for storing comparison results obtained from the comparator FU of the processor. The guard unit can also issue a lock, which is propagated as a global lock outside the control unit, in order to stall the entire pipeline for a certain amount of clock cycles. Figure 4.9 shows the guard unit structure.

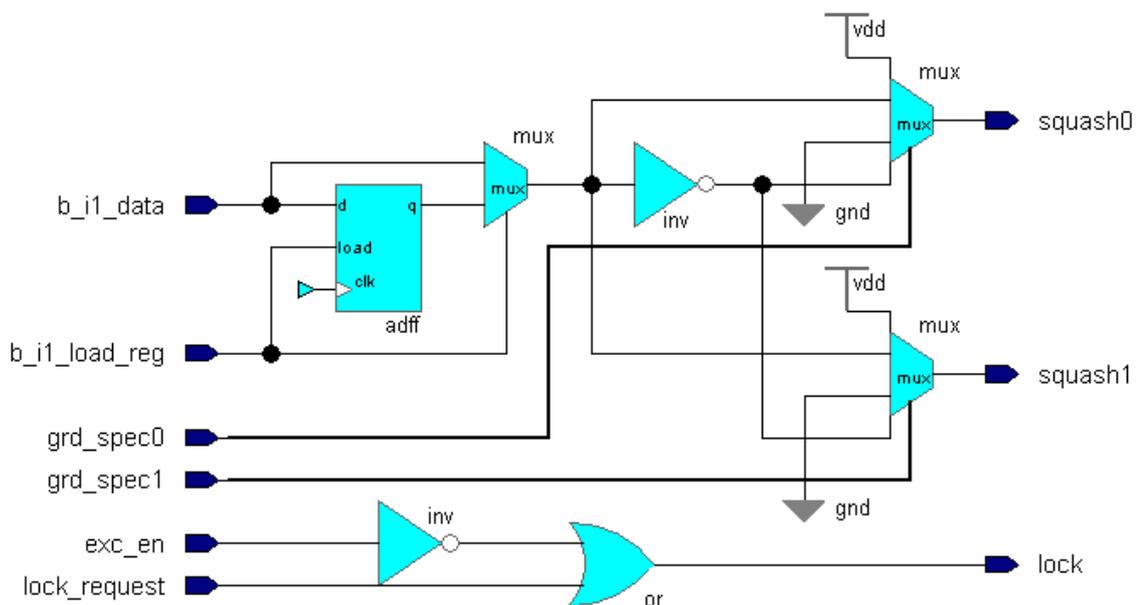


Figure 4.9: Guard unit of a two-bus TTA processor.

In Figure 4.9, the `b_i1_data` input signal is the Boolean output value of the comparator FU. The register, referred to as Boolean register, holds the comparison result for the next clock cycle. Signal `b_i1_load_reg` functions as a load enable signal for the Boolean register. The signal also controls the multiplexer, which determines whether a registered Boolean value is used (when `b_i1_load_reg = 1`), or the Boolean register is bypassed and the current comparator FU result is used directly (`b_i1_load_reg = 0`).

Signal `grd_spec0` is the guard specifier field for processor data bus 0. The multiplexer on the upper right corner of Figure 4.9 selects the source for the `squash0` signal, the

selection being based on the `grd_spec0` value. One such multiplexer is required for each bus in the processor, as the buses require dedicated squash signals. Independent of the other logic in the guard unit, a lock signal is generated from `exc_en` and `lock_request` signals received from the control FU.

4.3 Interconnection

The interconnection network consists of data buses and sockets, as already shown in Figure 4.1. The sockets provide programmable connections to the buses. Input sockets are basically input multiplexers that select a value from one of the buses and write it to an input register of an FU. Output sockets are output demultiplexers that transfer the contents of an FU result register to one of the buses. There are three different options for socket structure: AND-OR, tri-state and multiplexed. The AND-OR structure has been found to be the most efficient solution and is thus selected to be used in the DCT32 processor [2, pp. 64-65].

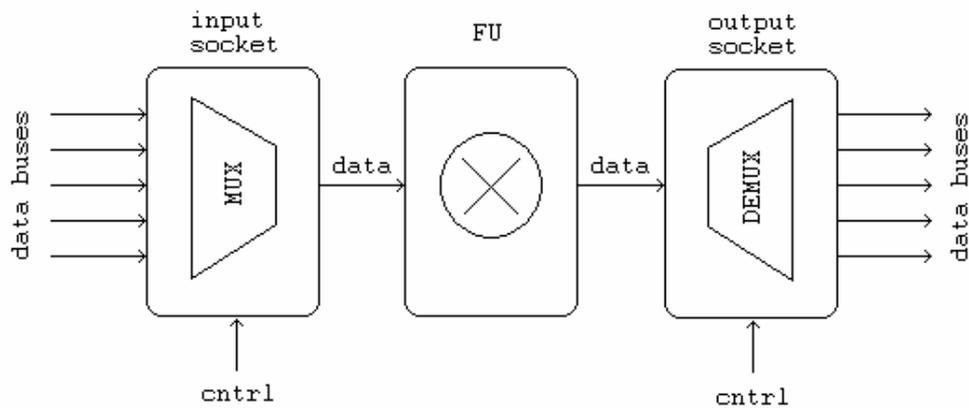


Figure 4.10: Operation of input and output sockets.

Figure 4.10 shows a conceptual view of the socket operation in the interconnection of a five-bus TTA processor. The input socket selects an appropriate data line to be connected to the FU input register according to a control signal originating from the global control unit. Similarly, the output socket selects a data bus, which is connected to the data output of the FU.

Figure 4.11 shows a part of the interconnection network structure of a TTA processor with two data buses. In Figure 4.11, there are two FUs, FU1 and FU2, both having the standard interface, i.e., a trigger input (fuX_t_data), an operand input (fuX_o_data), and a result output (fuX_r_data). The output sockets of both FUs are visible, but only FU1 trigger and operand input sockets are shown in order to keep the important signal names readable. FU2 trigger and operand input sockets and related signals are connected similarly to those of FU1.

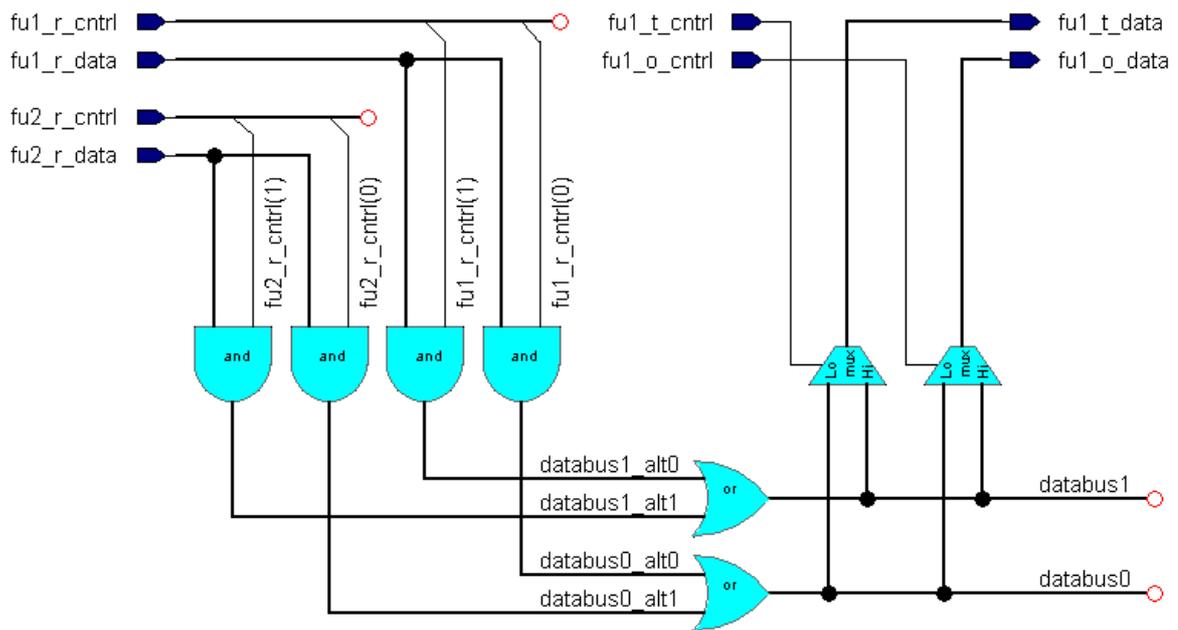


Figure 4.11: Interconnection network realized using AND-OR structure.

The AND-OR-structure in Figure 4.11 can be considered as the output sockets of the two FUs. More specifically, the OR-gates are actually bus write units which write the data received from the output sockets to the buses. The multiplexers are the input sockets of FU1. The control signals and buses, i.e., buses $fu1_r_cntrl$ and $fu2_r_cntrl$, and signals $fu1_t_cntrl$ and $fu1_o_cntrl$, are obtained from the instruction decode logic located in the global control unit. The output socket control bus width is equal to the amount of data buses, while the input socket control bus width is determined so that the control bus can uniquely index each data bus of the processor. For example, when there are five data buses in the processor the output socket control bus width is five, i.e., there is one signal for each data bus. The input socket control bus width, on the other hand, is three, as three binary signals can index a maximum of $2^3 = 8$ data buses.

The GCU sends each output socket a control signal which selects the buses to be connected to the FU output. A connection to a bus is enabled when a corresponding bus signal is brought high. Each input socket also receives a control signal from the GCU, the control signal value indexing a bus that is connected to the FU input. In Figure 4.10, for example, when the input socket `fu1_t` (the leftmost mux) receives a logical '0' as its control signal, it connects the `fu1_t_data` input to `databus0`.

4.4 Function Units and Register Files

Function units are custom blocks with a standard interface. The standard FU interface is shown in Figure 4.12. FUs perform different operations and communicate with external devices, such as RAMs. An FU can, for example, multiply or shift values, or load data from memory. A TTA processor typically has several different FUs to be able to perform the required operations.

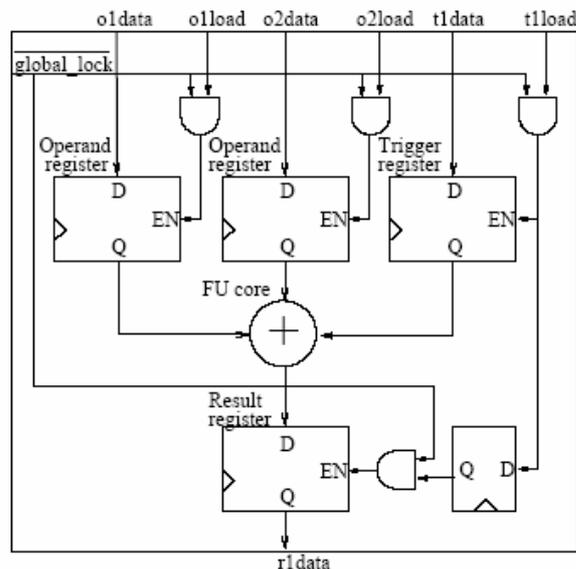


Figure 4.12: TTA function unit interface and structure [2, p. 21].

Function units may use different pipeline latching disciplines which determine when pipeline registers are allowed to accept new data [1, p. 209]. The function unit in Figure 4.6 follows the semi virtual-time latching (SVTL) discipline, i.e., only a move to the trigger input (`t1data`) of an FU initiates a new operation [1, p. 222]. This latching discipline is used throughout the FUs in the DCT32 processor.

Register files are similar to FUs, except they perform the identity operation, i.e., they only store data for future use. Register files can be assigned to specific resources or they can be general-purpose registers. Figure 4.13 shows a view of a register file structure.

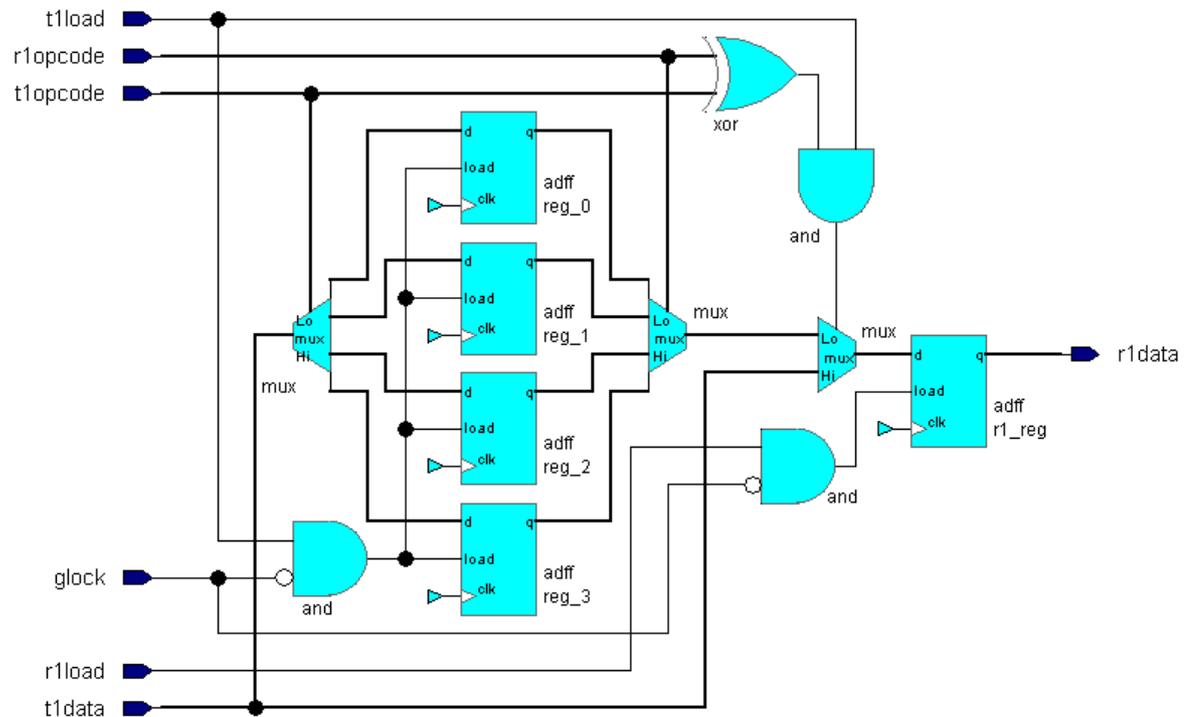


Figure 4.13: Register file structure.

Figure 4.13 shows the structure of a register file with one read and one write port. Register file size is 4 by 32 bits, as there are four 32-bit registers that store data. The registers are controlled by load and operation code (`opcode`) signals. The output (result) register placement is similar to the standard FUs.

5. FPGA IMPLEMENTATIONS

This chapter presents the FPGA implementation results of the TTA hardware accelerator, as well as the results of various experiments with the DCT32 processor. In addition, four different TTA processors were synthesized and mapped to FPGAs, and their timing results were analyzed in order to discover the FPGA timing properties of TTA processors in general.

The TTA DCT32 implementation is discussed in Sections 5.1-5.5. The implementation basics are introduced in Section 5.1. For clarification purposes, hardware and software aspects of the implementation are individually discussed in Sections 5.2 and 5.3, respectively. Section 5.4 introduces the co-verification tools and techniques. Section 5.5 presents and discusses the results of the implementation, as well as the results of other FPGA experiments. Timing results of different TTA processors are separately analyzed in Section 5.6.

The design was synthesized and fitted using version 4.1 of Altera Quartus II design software. The Quartus II software provides several useful tools for design analysis and optimization. The hardware design was verified using Mentor Graphics ModelSim version 5.7e. AXD Debugger from ARM Limited was used along with Altera SignalTap II Logic Analyzer for co-verification and debugging. Xilinx results presented in Section 5.5 were obtained using version 6.2 of Xilinx ISE design software.

5.1 Principles

As discussed in Chapter 3, the co-operation of two processors requires messaging in the form of transfers and acknowledgements. The implementation presented in this chapter is based on the simple transfer model introduced in Section 3.2, in which the master processor (ARM922T) initiates transfers to the slave (TTA DCT32) processor. After receiving a transfer in which the processing is initiated by a write to the flag register, the slave processor calculates the 32-point DCT and notifies the master processor when the transformed data is ready to be read. The notification can be considered an acknowledgement to the initiating transfer.

A special function unit was created to enable external control over the TTA processor. The function unit, serving as a flag register (a flag register was discussed in Section 4.2), is a control FU which enables the DCT32 processor pipeline to be locked and execution to be initiated. The control FU contains some additional control signals so that the flag register can also be written and read by the master processor.

Figure 5.1 shows the principal structure of the interface between the TTA and ARM processors. The ARM processor is located in the top edge, and the relevant parts of the TTA processor in the bottom edge of Figure 5.1. The memories and the control FU containing the control register are located in the middle, between the two processors. Memory access control logic, which is discussed later in this chapter, is also visible in Figure 5.1.

5.2 Hardware Implementation Details

Appendix A presents the top-level block diagram view of the design. The leftmost block, named `arm_stripe`, represents the embedded processor stripe with UART, EBI and stripe-to-PLD bridge enabled. The block below the stripe, `movecore`, is the core of the TTA processor. The upper block in the middle, `memory_controller`, represents the memory controller module that interfaces with the stripe-to-PLD bridge, the memories, and the control register. The remaining blocks are synchronous RAM instances, dedicated to either data memory (the block below `memory_controller`) or instruction memory (the four blocks on the right). An interesting detail in the block diagram is the signal from `memory_controller` output pin `IRQ_out` to `arm_stripe` input pin `intpld[0]`. This line carries an interrupt request signal from the memory controller to the ARM processor.

In this section, each non-trivial resource of the system is discussed. First, the control register is examined in Subsection 5.2.1. Second, the memory controller is introduced in Subsection 5.2.2. Last, the implementation of the 128-bit instruction memory is discussed in Subsection 5.2.3.

5.2.1 Control FU and Control Register

The control FU is a control register wrapped in a standard FU interface. The interface, however, contains a few additional signals to enable processor control. Figure 5.2 shows the control FU structure.

The control FU resembles a standard FU, as it contains a trigger input (`tldata`) as well as a trigger load enable (`tlload`) input. In addition, `rldata` output is also a part of the standard FU interface. Besides the standard interface, there are additional inputs: `data_in` and `wr`. The `data_in` signal delivers master processor write data to the control register when the `wr` signal is active.

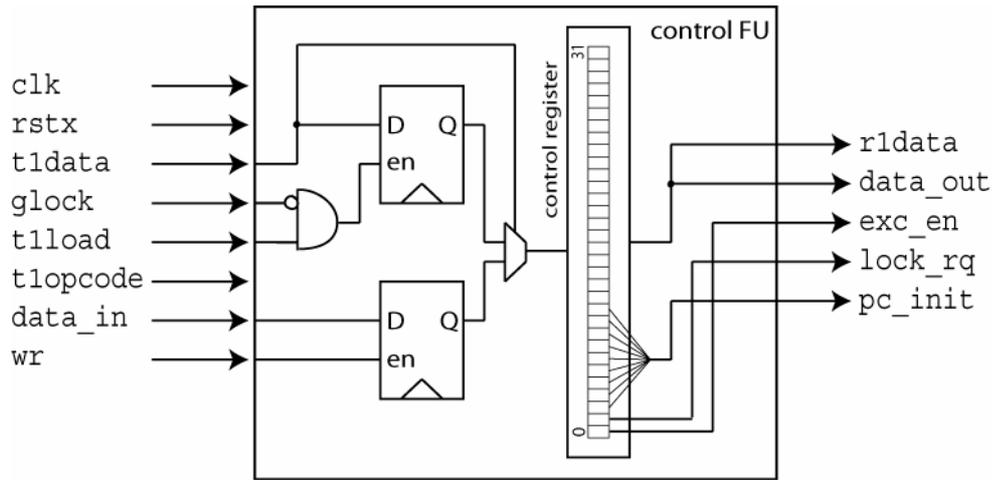


Figure 5.2: Control FU structure.

The control register is 32 bits wide. Register bits have been assigned different purposes. The assignment is shown in Figure 5.3. The bits emphasized with grey colour in the control register are reserved for program counter initialization. The amount of reserved bits depends on the width of the instruction memory. The PC initialization enables the ARM to load custom values to the PC register.

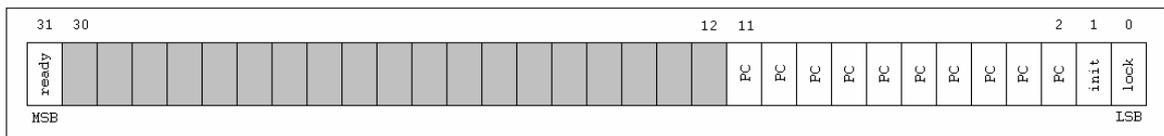


Figure 5.3: Control register bit assignment.

Bit 31, the MSB, informs the master central processing unit (CPU) that the DCT calculation is ready. The master CPU continuously polls the MSB of the register. Bits 30 to 12 are, as mentioned, reserved for PC initialization in case of a wider instruction memory (resulting in a wider PC register). In the DCT32 processor, control register bits from 11 to 2 are used for program counter initialization, as the instruction memory address width is 10 bits. Bit 1 is assigned for initiating the DCT32 execution, i.e., enabling the program counter of the instruction fetch unit. Bit 0 is for sending a pipeline lock request for the global control unit of the processor. When the GCU receives a lock request, it issues a global lock which halts the TTA processor. A global lock is also issued when the init-bit is set to zero.

5.2.2 Memory Controller

As already discussed in Chapter 1, Introduction, the basic idea of the TTA hardware accelerator case is to connect the ARM922T processor stripe to the PLD side devices, i.e., the DCT32 processor control register and the memories, via the advanced high-performance bus provided by the EPXA10 board. The ARM processor stripe offers an AHB-based stripe-to-PLD interface and a dual-port RAM interface with PLD access, both of which enable the user to transfer data to and from the PLD. The former was selected to be used in the design, as it was considered a more flexible solution. There were two options for the AHB connection method: either to use a single wrapper for the DCT32 core, the instruction memory, and the data memory, or to wrap each component individually. The former seemed to be a better method, mainly because using a dedicated wrapper for each component would require an external arbiter to control the access to the AHB in order to avoid multiple bus drivers. In addition, the DCT32 core and the memories compose a processor which can be considered as a single AHB slave.

The block diagram displayed in Appendix A shows the top-level structure of the design. Theoretically, both the data and the instruction memory could be accessed simultaneously by the DCT32 and ARM922T processors, but this should never occur in practice. When the DCT32 accesses the data memory, it is performing the discrete cosine transform (DCT) calculation and its memory access should not be blocked. The ARM expects to receive the final results when accessing the data memory, therefore memory access is not allowed until the DCT calculation is complete. Instruction memory write, on the other hand, is an exceptional procedure which should only be performed when the DCT32 processor is in its initial state. In general, the state of the DCT32 processor should be checked from the control register prior to accessing either the instruction or the data memory.

Appendix B shows the VHDL source code of the memory controller module. The controller consists of an AHB interface, which receives and transmits AHB compliant signals to the bus, and separate interfaces for the memories and the DCT32 control register. `HWDATA`, `HWRITE`, and `HADDR` are functionally the most critical signals that the memory controller receives from ARM via AHB. These signals select the data to be

written and the write destination (when HWRITE is high), or the read source (when HWRITE is low). Signal HTRANS helps to distinguish idle states from active transfers.

Component	Base address
data memory	0x80000000
instruction memory	0x80010000
control register	0x80020000

Table 5.1: Component base addresses.

The components, i.e., the data memory, the instruction memory, and the control register, are separated using unique address offsets. The offsets, i.e., the base addresses, are shown in Table 5.1. In addition, the memory controller selects the read source by monitoring the latched address received from the AHB and by connecting the output of the control register, instruction memory array, or data memory to the ARM data input signal, HRDATA.

5.2.3 Instruction Memory

A special memory cell arrangement was required to enable the master to easily read and write the instruction memory contents. A simplified view of the arrangement is shown in Figure 5.4.

The instruction memory is divided into four 32-bit memory blocks for easier access from the 32-bit AHB bus. The `data_in[31..0]` input pin is directly connected to the AHB HWDATA signal. Read or write address, `address[9..0]`, is selected between HADDR and the address received from DCT32. Each memory array has two enable inputs, write enable and read enable. They are individually controlled by the memory controller, according to the enable signal received from the TTA DCT32 and the AHB signals HADDR and HWRITE.

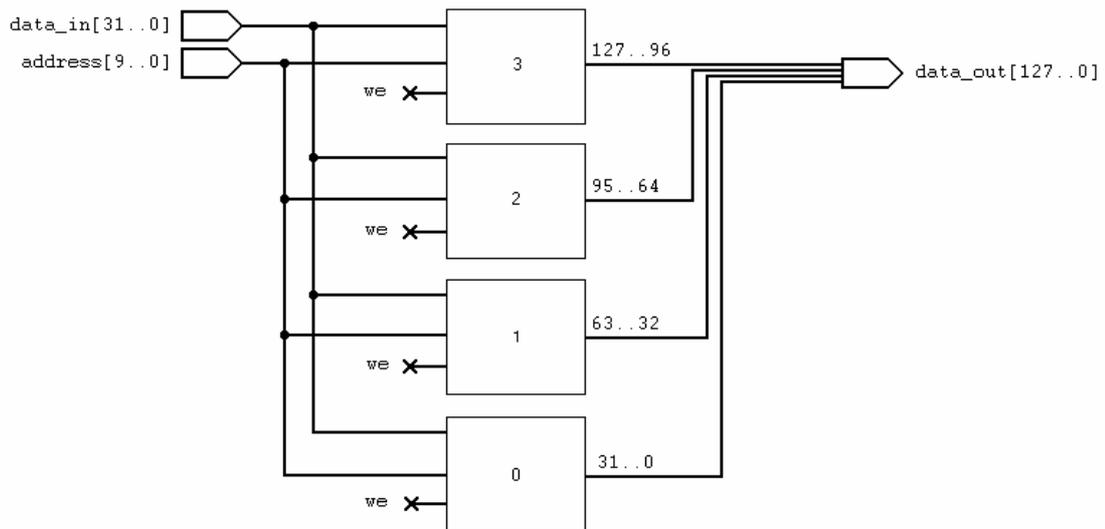


Figure 5.4: Physical structure of the instruction memory.

When the AHB reads from or writes to the instruction memory, the correct memory array is enabled by the address bits $HADDR[3..2]$. As mentioned earlier, the ARM processor indexes memory in 8-bit words, therefore the two AHB address bits select a 32-bit memory array. On the other hand, when the TTA processor accesses the instruction memory, its only purpose is reading, and thus all write enables are brought to an inactive state and all read enables are brought to an active state. As a result, all four memory regions output a 32-bit word from an address location specified by the address received from the DCT32. Concatenating these four 32-bit words according to Figure 5.4 results in a 128-bit instruction read operation.

5.3 Software Details

The instruction memory has so far been implemented as a pre-programmed read-only memory (ROM) in the TTA hardware accelerator design, but reprogrammability requires the memory to be implemented as random access memory (RAM). Instructions must therefore be loaded into the instruction RAM prior to initiating the first DCT calculation. The instructions are loaded to the instruction memory by writing four 32-bit words to consecutive memory locations, starting from the word containing the MSB of the instruction to be written. The hardware stores the 32-bit instruction segments into the

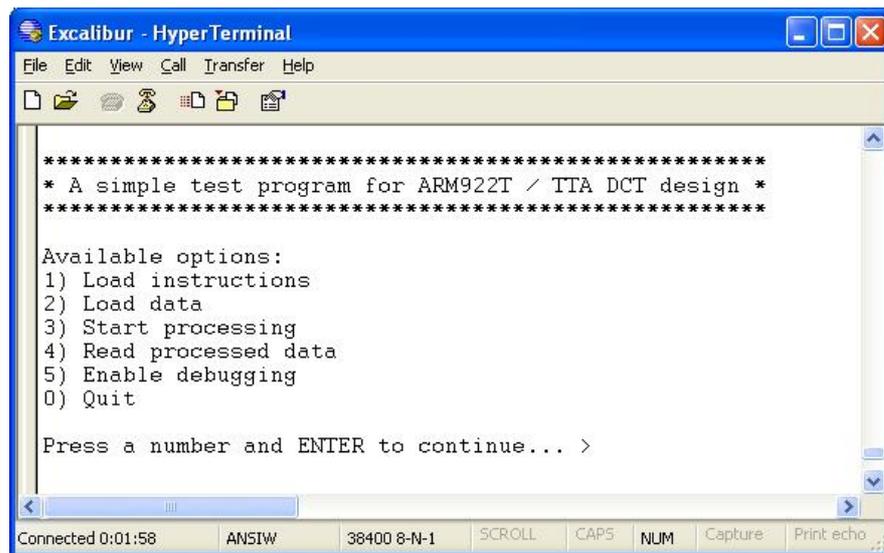
memory arrays and, when the instruction is being read, concatenates the array outputs properly to form a 128-bit instruction for the TTA DCT32 processor.

The functionality of the implementation was tested using a C-language program that enables the user to read and write data and instruction memory contents, modify control register contents, and to initiate coprocessor calculation. Appendix C presents the source code of the test program.

The test program utilizes the interrupt controller provided by the embedded stripe. The program contains an interrupt service routine, mainly to clear the interrupts set by the memory controller and to notify the user that the DCT data is available. As mentioned in Chapter 3, the operating mode in which six individual interrupts are enabled is the default mode after reset. The mode is also used in the current application, as one interrupt signal, which indicates the master processor that the DCT32 processor has finished processing, is sufficient. Using an interrupt instead of continuous control register polling in software results as more efficient code and enables the processor to efficiently run other tasks while the DCT data is being processed. This enables the use of pipelining in time-critical applications. The test program implemented for verifying the hardware operation, however, does not support pipelining.

Altera provides the basic initialization files for all embedded stripe devices, such as the interrupt controller, UART, and timer. The initialization files must be considered as templates that may require modifications to suit the current application.

The test program and the interrupt handler function are both located in file `dct32.c`. A great benefit offered by the test program is that users no longer have to enter the data and instructions manually, but the instructions can be read from files provided in the simulation folder of the TTA processor source code package, and uploaded to memories through the UART connection between a PC running a terminal software and the development board. The test program is controlled through a terminal, as a menu-based application. Figure 5.5 shows a view of the initial screen of the test program, being run on Windows HyperTerminal terminal software.

The image shows a screenshot of a HyperTerminal window titled "Excalibur - HyperTerminal". The window has a menu bar with "File", "Edit", "View", "Call", "Transfer", and "Help". Below the menu bar is a toolbar with icons for file operations. The main text area contains the following text:

```
*****  
* A simple test program for ARM922T / TTA DCT design *  
*****  
  
Available options:  
1) Load instructions  
2) Load data  
3) Start processing  
4) Read processed data  
5) Enable debugging  
0) Quit  
  
Press a number and ENTER to continue... >
```

At the bottom of the window, there is a status bar with the following information: "Connected 0:01:58", "ANSIW", "38400 8-N-1", "SCROLL", "CAPS", "NUM", "Capture", and "Print echo".

Figure 5.5: Running the test program through an UART connection.

The menu option “1” loads the instruction memory contents to the TTA DCT32 instruction memory. The data, which can either be copy-pasted from a file or transferred directly using the send file-option of the terminal software, is parsed prior to forwarding it to a correct address space on the PLD. The same applies to menu option “2”; the user may send the data from a file provided along with the TTA processor source package. Processing starts when the control register init-bit is set to ‘1’ and the lock-bit is set to ‘0’. The memory controller sends an interrupt request when it notices a transfer from ‘0’ to ‘1’ in the ready bit of the control register. The interrupt handler then resets the control register to the lock-state, and notifies the user that the DCT data is ready. The user may read the processed data to a file using the menu option “4”. The user can enable debugging, which brings extra information to the screen, by using the menu option “5”. Debugging also enables the user to read the instruction memory contents, write values to the control register and read the control register contents. These options are hidden behind the debug menu, because they are not supposed to be used during normal operation.

5.4 Co-verification

There were several alternatives for the co-verification of the design. Altera provides an Excalibur stripe simulator (ESS) model to be used in a software debugger, such as the AXD Debugger. This model connects to ModelSim HDL simulator through a foreign-

language interface (FLI). The ESS simulation model, however, is not fully cycle-accurate [12, p.13]. As one EPXA10 development board was available for testing, another option was to use Altera-RDI through JTAG interface in AXD Debugger. The RDI is the standard application interface between ARM processors and ARM-supported debuggers [13, p. 2]. Other options for verification were Mentor Graphics Seamless Co-Verification Environment (CVE), and Excalibur bus-functional model. The Seamless CVE does approximately the same as the AXD debugger combined with ModelSim and the ESS model. The Excalibur bus-functional model enables the user to quickly test AMBA compliant slaves by emulating the bus traffic [14, p. 9]. The UART connection can also be used to transfer useful debug information to the terminal window.

The AXD debugger is extremely useful, especially in viewing the memory and register contents, and stepping through the executable code, as the debugger takes full control over the ARM processor through the processor debug interface. The debugger displays a disassembly view of the machine-language program that is being run, unless the user loads in the debug symbols generated by the Quartus II software, in which case the debugger is able to display the full C-language source codes.

Altera's SignalTap II embedded logic analyzer, provided with Quartus II design software, was used to display the signals during software execution with the actual hardware, i.e., the EPXA10 development board. Figure 5.6 displays a logic analyzer view of selected tapped signal waveforms during the DCT calculation.

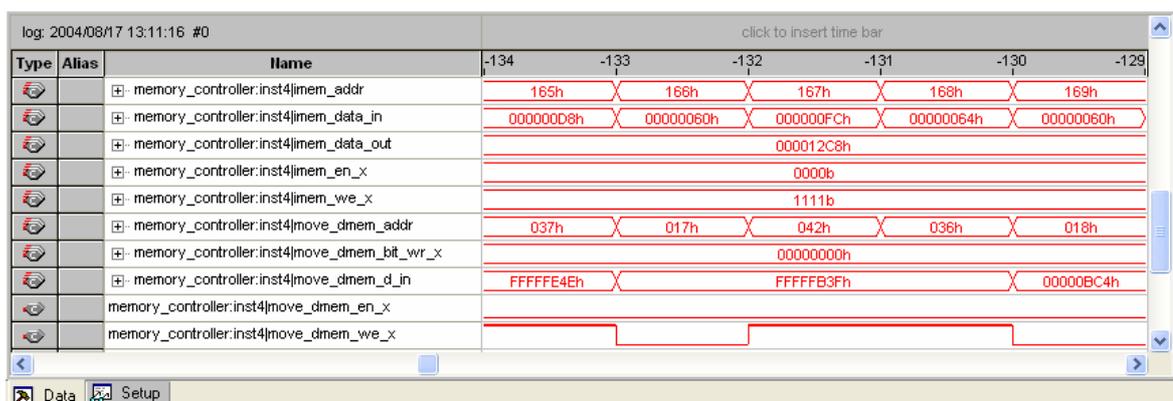


Figure 5.6: SignalTap II logic analyzer signals.

Using the SignalTap II logic analyzer consumes development board resources, mainly memory and FPGA logic elements. The resource requirements are directly proportional to the sample lengths and the amount of tapped signals. This is because the logic analyzer is implemented on the PLD, similarly to the rest of the design. This leads to one of the drawbacks concerning the SignalTap II logic analyzer: every time the hardware is modified, a full compilation process must be run in order to be able to view the results of the modification. The full compilation process, including fitting, may last hours when compiling a large design, such as the current application, on a slow computer.

5.5 Results

This section presents the results of the implementation, as well as the results obtained from various experiments with the TTA DCT32 processor. First, the timing reports of the ARM / TTA design implementation on an Altera device are examined in Subsection 5.5.1. Second, the timing results of a modified TTA DCT32 processor implementation on an Altera FPGA are analyzed in Subsection 5.5.2. Third, the amount of resources the design requires is discussed in Subsection 5.5.3.

In addition to the Altera APEX 20KE device, the design was also synthesized for a Xilinx Virtex-II Pro FPGA chip [18, pp. 2-3] for comparison purposes. The timing and resource utilization results of the Xilinx implementation are presented in Subsections 5.5.4 and 5.5.5. Results concerning the Altera device are obtained from the analysis and reporting tools of Quartus II design software. Xilinx results are provided by Xilinx ISE design software.

5.5.1 Altera Timing Results of the ARM / TTA Design

The Quartus II software performs static timing analysis which evaluates the critical paths of the design mapped to the FPGA. The static analysis needs to be manually adjusted to ignore paths that have no effect on the operation of the circuit. In literature, these paths are referred to as false paths. Ignoring the false paths is important as it enables the designer to efficiently discover the real critical paths of the design.

The design was synthesized and mapped to an Altera APEX 20KE1000E, i.e., the FPGA device in the EPXA10 development board. The design contains a single clock signal, referred to as PLD clock. The PLD clock is common to the TTA DCT32 processor core, AHB bus, memory controller, and memory modules. The maximum clock frequency of the PLD clock is 37.5 MHz. In some compilations, the multiplier FU was the bottleneck of the design. This, however, was easily solved by increasing the amount of pipeline stages in the multiplier. This issue is addressed in Subsection 5.5.2.

The ASIC implementation of the DCT32 processor yields an operating frequency of approximately 200 MHz [2, p. 60]. As expected, the maximum core clock frequency of the FPGA implementation is nearly one fifth of the original ASIC implementation. For comparison, when implementing a RISC processor core on an FPGA, the resulting clock frequency is approximately 20 % of the equivalent ASIC implementation [17, p. 4]. The timing results of the ASIC implementations can roughly be compared, because the technology used in the RISC processor was comparable to the technology used in the TTA DCT processor synthesis.

5.5.2 Altera Timing Results of the Modified TTA DCT32 Processor

In some compilations, the multiplier FU of the TTA processor was the slowest component in the processor. However, as the multiplier critical path is not the architecture-specific bottleneck of the design, the multiplier was pipelined in order to discover the real bottleneck of the processor architecture on the FPGA. The Quartus II synthesis tool automatically replaces the multiplier used in the design with a multiplier megafunction that is optimal for the FPGA realization. This megafunction offers an option to use a pipelined multiplier with two or more pipeline stages.

After pipelining the multiplier, the critical path is the path starting from the comparator FU output register, travelling through the global control unit, and ending at a register file. Figure 5.7 shows a detailed view of the critical path.

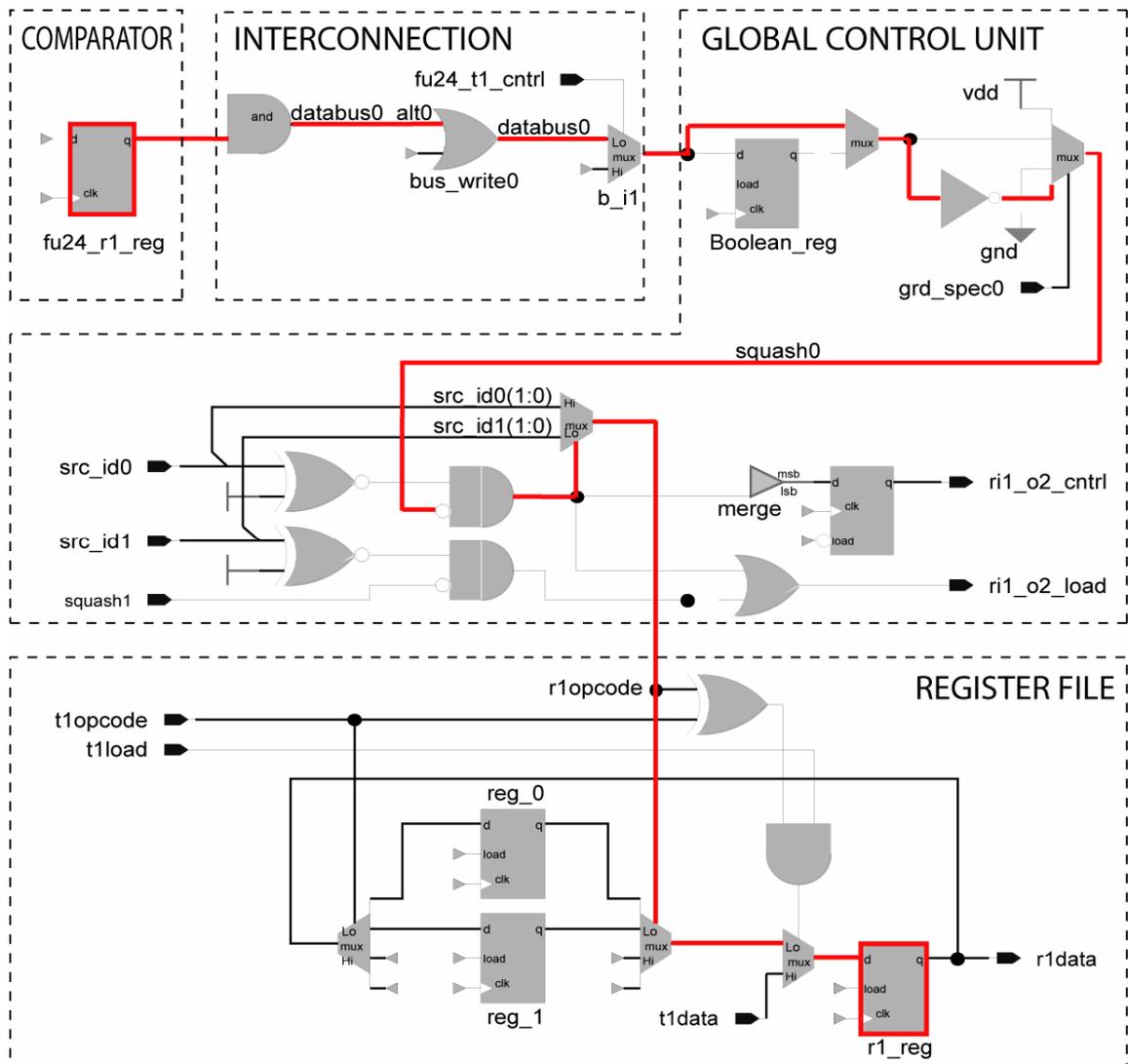


Figure 5.7: Architecture-specific critical path of the TTA DCT32 processor.

The dashed line surrounds the individual processor components in Figure 5.7. The logic that is not related to the critical path is excluded for clarity. The first component, the comparator FU, is similar to the adder FU shown in Figure 4.11, except that the FU core (adder) is replaced by a comparator. The comparison result is transferred via the interconnection network to the global control unit. The GCU can be divided into two units: the guard unit and instruction decode unit. Finally, the last component in the critical path is the register file.

The critical path is related to the cancelling of transports during conditional branches. The comparator FU evaluates a condition and transfers the result from its result register to the guard unit of the GCU. The result would normally be obtained from the Boolean register, but if the squash signal is needed on the same clock cycle, the register is bypassed. The

squash signal is transferred to the instruction decode unit, in which the signal affects the generation of the network control signals. The IDU generates a register file operation code signal, which is used for controlling the register file operation. The critical path ends at the register file result register. Table 5.3 shows the critical path composition and timing information.

node name	node details	routing delay (ns)	cell delay (ns)	total delay (ns)
fu24:r1reg(0)	comparator result register	0.000	0.142	0.142
ic:bus_write_alt_22:bus2	databus_alt	3.239	0.800	4.181
ic:bus_write_alt_22:bus2	bus write OR	0.298	0.335	4.814
ic:bus_write_alt_22:bus2	databus	0.290	0.726	5.830
ic:input_socket:b_i1	input socket for guard unit	1.670	0.726	8.226
cntrl:mux	comparator result from bus	0.290	0.335	8.851
cntrl:squash	squash signal	1.455	0.335	10.641
cntrl:and	-	0.363	0.335	11.339
cntrl:and	-	2.927	0.335	14.601
cntrl:mux	multiplexer selecting opcode	0.382	0.783	15.766
cntrl:mux	mux output	1.108	0.726	17.600
rf8:and	-	2.861	0.115	20.576
rf8:and	-	0.000	0.512	21.088
rf8:mux	mux selecting output data	2.920	0.726	24.734
rf8:r1_reg	register file result register	1.044	0.437	26.215
total routing delay (ns)		18.847	71.89 %	
total cell delay (ns)		7.368	28.11 %	

Table 5.3: Critical path composition of TTA DCT32 with pipelined multiplier FU.

The critical path structure in Table 5.3 is somewhat different from the one that is shown in Figure 5.7 because Quartus II timing analyzer and RTL viewer tools show and analyze the optimized post-synthesis structure of the circuit, whereas the structure shown in Figure 5.7 is based on the VHDL RTL description of the processor.

As can be seen in Table 5.3, the FPGA routing delays comprise approximately 70 % of the total critical path delay. The total delay is 26.22 ns, which corresponds to a maximum operating frequency of 38.15 MHz.

The critical path in an ASIC technology is similar to the path shown in Figure 5.7 and Table 5.3. However, the Boolean register bypass line is significantly faster than in FPGAs, as would be expected. This is due to the extremely fast routing on ASIC technologies. When implemented using a 0.13 μm technology, the maximum clock frequency of the two-bus version of the DCT32 processor was 268 MHz [2, p. 60]. Using a 0.11 μm technology resulted in a maximum clock frequency of 302 MHz with a five-bus DCT32.

In addition, when the Boolean register bypass line is cut in ASIC technology, the resulting critical path is located in the interconnection network, similarly to as in the FPGA.

5.5.3 Altera Device Resource Utilization

Table 5.4 displays the device resource usage and the available resource amounts of an APEX EP20K1000E device, which is included in the EPXA10 development board. The design requires a relatively large portion of the total available memory. However, we could have utilized a smaller-capacity APEX EP20K400E FPGA device as well, as its logic element and memory capacities meet the requirements of the design.

Resource	Usage (Max)	%
Logic elements	11887 (38400)	30
Pins	66 (715)	9
Memory bits	196608 (327680)	60

Table 5.4: Summary of Altera device resource utilization.

The original version of the TTA DCT32 processor, which has no extra pipelining in the multiplier FU, uses 11,129 logic elements alone. This constitutes 94 % of all the logic elements utilized, making the DCT32 processor the dominant device on the PLD. The memory controller module consumes 758 logic elements.

As the performance of the design was more important than the resource consumption, the design timing was optimized at the expense of resource usage. The Quartus II design software provides options for selecting the synthesis optimization method. When the resource optimization method was used instead of performance optimization, the logic element utilization dropped by 9 %, to 10,782 consumed logic elements. As a disadvantage, the maximum clock frequency suffered a large penalty of approximately 10 %, resulting as 33.7 MHz.

5.5.4 Xilinx Timing Results of the Original TTA DCT32 Processor

The original design of the DCT32 processor was also synthesized for a Xilinx Virtex-II Pro XC2VP20 device with speed grade 5. After standard-effort place & route, a minimum clock period of 29.2 ns was obtained. This corresponds to a maximum clock frequency of 34.3 MHz. The critical path was identical to the Altera case presented in Subsection 5.5.2. The critical path comprises of 77 % routing delay and 23 % logic delay. The path between the multiplier FU I/O registers was not among the most critical paths of the design mainly because Virtex-II Pro contains hardware multipliers, which rather than logic blocks, were utilized in implementing the multipliers.

When switched to high optimization effort, the minimum clock period was shortened to 26.5 ns, which corresponds to a clock frequency of 37.8 MHz. The critical path structure remained the same. The elapsed time for place & route increased slightly, but remained at about 10 % compared to that of the Altera software. As a result, the whole compilation took approximately 10 times longer for Quartus II than for Xilinx ISE.

5.5.5 Xilinx Device Resource Utilization

Table 5.5 shows a summary of the Xilinx XC2VP20 device resource utilization. The utilization of I/O blocks is shown on the first row. In addition, the utilization of hardware multipliers, RAM blocks, and logic slices is also displayed.

resource	utilized	total	utilization
I/Os	89	404	22 %
MUL18X18s	3	88	3 %
RAMB16s	12	88	13 %
SLICES	5132	9280	55 %

Table 5.5: Summary of Xilinx device resource utilization.

The logic capacity of the XC2VP20 device was ample for TTA DCT32 processor. However, the processor could not be fitted into a smaller XC2VP7 device due to insufficient resources.

5.6 Different TTA Processors

Four different TTA processors were synthesized and their critical paths were analyzed in order to discover whether the common architecture of TTA processors could be optimized for FPGA synthesis and fitting. The critical paths in FPGA implementations may be different from those of ASIC implementations, as routing delays often dominate in FPGAs. The critical path is located either in the global control unit or the interconnection structure, since TTA processors contain various FUs with different amount of pipeline stages. A critical path which is located between the FU input and output registers is never acceptable, as every FU can be pipelined with sufficient amount of registers, thus dividing the long combinatorial paths and cutting the critical path of the FU.

Table 5.6 shows the resources of five tested TTA processors. Processor 1, the DCT32 processor, was synthesized and its resources were presented in Section 5.3. It was selected as a reference processor, and its resources (FUs, RFs, and data buses) were doubled in turns to create five different TTA processors for discovering possible dependencies between the maximum attainable clock frequency and processor resources. All tested processors are fully connected, i.e., every I/O socket is connected to all data buses.

resource	processor 1	processor 2	processor 3	processor 4	processor 5
multiplier FU	1	2	1	2	2
adder/subtractor FU	2	4	2	4	4
shifter FU	1	2	1	2	2
comparator FU	1	2	1	2	2
sign extension FU	1	2	1	2	2
load/store FU	1	2	2	2	2
control FU	1	1	1	1	1
register file	8	8	8	8	16
total FUs & RFs	16	23	17	23	31
databuses	5	5	10	10	10
instruction width	123	123	214	214	224

Table 5.6: Resources of the evaluated TTA processors.

As the compilation fitting process is random and based on a user-entered seed value, the timing results vary between compilations. Seed values affect the random choices made by Quartus II software during the fitting process. Obviously, the seed sweep, i.e., the search for the best fitting result when a different seed value is used in every fitting process, does

not affect resource utilization results. Table 5.7 shows the results of a ten-seed sweep performed by Altera Design Space Explorer (DSE).

seed	clk period (ns)	clk freq (MHz)
1	29.920	33.42
2	29.929	33.41
3	31.013	32.24
4	29.509	33.89
5	28.723	34.82
6	32.172	31.08
7	30.977	32.28
8	30.277	33.03
9	29.357	34.06
10	29.965	33.37

Table 5.7: Differences between compilations.

The results in Table 5.7 are obtained from compilations of the processor number 2. However, the true critical path is typically roughly 1.5 ns shorter than the path the DSE reports. The difference in clock frequency between the reported best (seed = 5) and worst case (seed = 6) is 2.74 MHz, i.e., the best case is approximately 9 % better than the worst case. Therefore, all processors were compiled using the Design Space Explorer with 10 seeds, and only the best cases of these compilations were taken into account in the comparison. Though using more seeds would produce better results up to a certain point, the seed sweep increases compilation time in a linear fashion and becomes impractical if the sweep lasts for days. For example, the sweep in Table 5.7 took 20 hours to complete.

proc. #	processor details	size (LEs)	period (ns)	freq (MHz)	critical path source reg	critical path dest. reg
1	reference	10 871	26.215	38.15	fu_eq_gt_gtu:fu25 r1reg	rf_2wr_2rd:rf8 r2_reg
2	2*FU	14 099	27.042	36.98	fu_eq_gt_gtu:fu25 r1reg	rf_2wr_2rd:rf8 r2_reg
3	2*bus	17 486	32.337	30.92	fu_eq_gt_gtu:fu24 r1reg	rf_2wr_2rd:rf8 r1_reg
4	2*FU, 2*bus	22 166	33.468	29.88	fu_eq_gt_gtu:fu24 r1reg	rf_2wr_2rd:rf8 r1_reg

Table 5.8: Resource utilization and timing results.

Table 5.8 shows the results obtained from the compilations. The processors were numbered similarly to as in Table 5.6. However, when compiling the largest processor (processor 5), the selected FPGA device ran out of routing resources, and thus the design could be neither fitted, nor analyzed. The resource utilization (size) of each processor is shown as logic elements (LEs). The minimum clock period and the maximum clock

frequency, as well as critical path source and destination registers are also listed for each processor.

The timing results in Table 5.8 clearly indicate that the critical path is always similar to the case with the DCT32 reference processor as shown in Figure 5.7 and Table 5.3. Thus, the critical path is independent of processor size and structure, i.e., is only related to the processor architecture.

Figure 5.8 shows the dependency of the clock frequency on the processor resources. The four size-frequency points on the graph are taken from the results in Table 5.8. It can be noticed that increasing the amount of FUs does not significantly affect the clock frequency. However, increasing the amount of data buses results as significant clock frequency degradation.

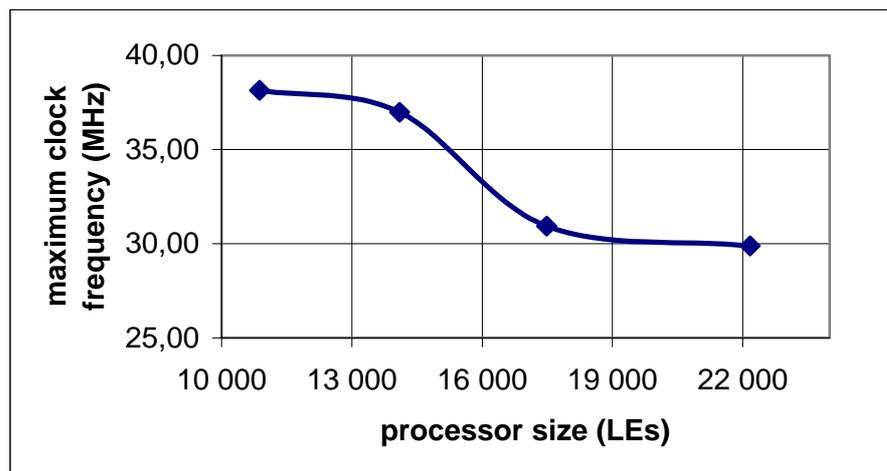


Figure 5.8: Maximum clock frequency versus processor size.

The compilations, the results of which are shown in Table 5.8, clearly show that the Boolean register bypass line, shown in Figure 5.7, is the bottleneck of the TTA architecture in the FPGA implementation. The bypass line was removed in order to locate the next critical path. However, the removal of this vital line without any other modifications to the control unit results as an inoperable processor. This test was carried out solely for timing analysis purposes.

After removing the Boolean register bypass line, the length of the critical path in the modified DCT32 processor, which is based on processor 1 in Table 5.6, was reduced to 21.0 ns, yielding an operating frequency of 47.6 MHz. The clock frequency increase was approximately 10 MHz (25 %) compared to the unmodified DCT32 processor. Similarly, when tested with a modified processor 4 in Table 5.6, the clock frequency increased 6.0 MHz (20 %) to 35.9 MHz. In both processors, the new limiting factor for the clock frequency increase was the bus structure, i.e., the interconnection network.

6. CONCLUSIONS

This thesis presented the common concepts of implementing a TTA processor on an FPGA technology. An implementation of the TTA DCT32 processor as a slave processor was successfully demonstrated and accurately documented. Two similar FPGA devices of different brands were compared, and no significant difference in performance was found. In addition, FPGA timing properties of the TTA were examined and a need for improvement was discovered.

The Boolean register bypass line was revealed to be the common FPGA bottleneck in a TTA. This problematic line is located in the global control unit of the processor. A solution for the bottleneck, the removal of the bypass line, was also analyzed. The solution was discovered to be fairly efficient, providing a clock frequency increase of approximately 25 percent. When this bottleneck was artificially removed, the clock frequency was limited by the processor interconnection structure.

The timing results of the TTA DCT32 processor FPGA implementation were as expected. The maximum clock frequency of the Altera APEX 20KE1000E FPGA realization was approximately one fifth of the corresponding ASIC implementation, resulting as a clock frequency of about 37.5 MHz. After pipelining the multiplier FU and removing the Boolean register bypass line, the maximum clock frequency peaked at 48.0 MHz.

The DCT32 processor was also fitted to a Xilinx Virtex-II Pro XC2VP20 FPGA device, and a maximum clock frequency of 37.8 MHz was obtained. The critical path in the Xilinx implementation was the same as in the Altera implementation. The timing results obtained with Altera and Xilinx FPGA devices were surprisingly similar, even though comparable

results could have been expected. After all, a Xilinx Virtex-II Pro series FPGA device was selected for the comparison for the exact reason that its architecture and technology resemble that of the Altera APEX 20KE series.

As the timing properties of different TTA processors were analyzed, a strong dependency between the amount of data buses in the processor and the maximum clock frequency was found. The clock frequency suffered a significant drop when the amount of data buses was increased. The amount of FUs in the processor, on the other hand, only had a negligible effect on the clock period. The dependency between the amount of data buses and the clock frequency, however, should be analyzed with more test cases in order to increase its accuracy and reliability. In addition, this analysis could be extended to cover more TTA-related options, such as I/O socket connectivity level or different combinations of register files and FUs.

According to the results obtained during this thesis work, there is no need to individually optimize the global control unit of TTA processors for ASIC and FPGA technologies. However, the GCU requires modifications, as the factor that restricts the clock frequency should be the data path of a processor, instead of the control logic.

The results presented in this thesis can be used as guidelines when designing a TTA processor for FPGA technology. In addition, this thesis provides valuable information on designing, implementing, and debugging designs on FPGA technology.

REFERENCES

- [1] H. Corporaal, "Microprocessor Architectures: From VLIW to TTA," Chichester, UK: John Wiley & Sons, 1997.
- [2] J. Sertamo, "Processor Generator for TTA," M.Sc. Thesis, Tampere University of Technology, Tampere, Finland, 2003.
- [3] J. Sertamo, H. Isännäinen, "MOVE TTA – ARM7TDMI co-simulation," Project report, Tampere University of Technology, Finland, 2004.
- [4] AMBA Specification Rev. 2.0, ARM Limited, 1999.
- [5] H. Corporaal, "Transport Triggered Architectures: Design and Evaluation," Doctoral thesis, University of Delft, the Netherlands, 1995.
- [6] Excalibur EPXA10 DDR Development Board Hardware Reference Manual, version 1.0, Altera Corporation, 2002.
- [7] Excalibur Devices Hardware Reference Manual, version 3.1, Altera Corporation, 2002.
- [8] ARM922T Technical Reference Manual, Revision 0, ARM Limited, 2000-2001.
- [9] AMBA Specification, Revision 2.0, ARM Limited, 1999.
- [10] S. Brown, J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," technical paper, University of Toronto, Canada, 1996.
- [11] APEX 20K Programmable Logic Device Family Data Sheet, Rev. 5.1, Altera Corporation, 2004.
- [12] Excalibur Stripe Simulator User Guide, Version 1.5, Altera Corporation, 2003.
- [13] Excalibur Software Debugging Solutions, Application Note #196, Version 1.2, Altera Corporation, 2002.
- [14] Excalibur Bus Functional Model User Guide, Version 1.2, Altera Corporation, 2002.

-
- [15] Spartan II Development System: Introduction to FPGA Technology, Trenz Electronic, Bünde, Germany, 2001.
 - [16] Quartus II Handbook Volume 1: Design & Synthesis, Rev. 5v1-2.0, Altera Corporation, 2004.
 - [17] Langen, Niemann, Pormann, Kalte, Rückert, “Implementation of a RISC Processor Core for SoC Designs – FPGA Prototype vs. ASIC Implementation,” technical paper, University of Paderborn, Germany, 2002.
 - [18] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, v4.0, Xilinx, Inc., 2004.

APPENDIX B

VHDL Source Code of the Memory Controller Module

```
-----
-- Memory controller for TTA / ARM Co-design
-----
-- TUT / DCS, FlexDSP
-- Miika Niiranen <miika.niiranen@tut.fi>
-----
-- Details:
-- A memory controller that takes care of all memory transfers. It combines
-- memories so that they are visible to AHB address space.
-----
-- Memory addresses (from ARM/AHB side):
-- * data memory      = 0x???00000 (base address)
-- * instruction memory = 0x???10000 (base address)
-- * control register = 0x???20000
-----
-- The design requires at least a 256K PLD region in Quartus II memory map.
-----
-- NOTES:
-- On AHB write:
-- * address and control signals need to be delayed
-- * write data will be available on the next clock cycle
-----
-- On AHB read:
-- * address and control signals need to be delivered immediately
-- * read data must be available on the next clock cycle
-----
-- ARM indexes memory in 8-bit words -> When dealing with 32-bit words in
-- software, bits HADDR(1..0) are always "00".
-- Therefore, during IMEM write, IMEM array is selected upon bits HADDR(3..2)
-- MOVE DCT32 indexes data memory in 32-bit words and instruction memory
-- in 128-bit words.
-----
-- Created:      2004-08-05
-- Last modified: 2004-08-05
-----
-- Modifications:
-----

library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity memory_controller is
    port (
        clk : in std_logic;
        rstx : in std_logic;

        -- AHB interface
        HWRITE : in std_logic;
        HADDR  : in std_logic_vector(31 downto 0);
        HTRANS : in std_logic_vector(1 downto 0);
        HWDATA : in std_logic_vector(31 downto 0);
        HREADY : out std_logic; -- indicates the AHB master when the bus is busy
        HRDATA : out std_logic_vector(31 downto 0);

        -- data memory interface
        dmem_en_x : out std_logic;
        dmem_wr_x : out std_logic;
        dmem_addr : out std_logic_vector(10 downto 0);
        dmem_bit_wr_x : out std_logic_vector(31 downto 0);
        dmem_data_out : out std_logic_vector(31 downto 0);

        -- instruction memory interface
        imem_data_out : out std_logic_vector(31 downto 0);
        imem_we_x : out std_logic_vector(3 downto 0); -- imem write enable (active low), 3 32-bit arrays
        imem_en_x : out std_logic_vector(3 downto 0); -- imem enable (active low), 3 32-bit arrays
        imem_addr : out std_logic_vector(9 downto 0);

        -- movecore interface
        move_imem_addr : in std_logic_vector(9 downto 0);
        move_dmем_addr : in std_logic_vector(10 downto 0);
        move_dmем_en_x : in std_logic;
        move_dmем_we_x : in std_logic;
        move_imem_en_x : in std_logic;
        move_dmем_d_in : in std_logic_vector(31 downto 0);
        move_dmем_bit_wr_x : in std_logic_vector(31 downto 0);

        -- control register interface
        cntl_data_out : out std_logic_vector(31 downto 0); -- data from AHB to control reg
        cntl_we : out std_logic; -- control register write enable

        -- memory input signals
        cntl_data_in : in std_logic_vector(31 downto 0);
        dmем_data_in : in std_logic_vector(31 downto 0);
        imem_data_in : in std_logic_vector(127 downto 0);

        -- IRQ output
        IRQ_out : out std_logic
    );
end memory_controller;

architecture rtl of memory_controller is
    signal ahb_dmем_addr : std_logic_vector(10 downto 0);
    signal ahb_dmем_en_x : std_logic;
    signal ahb_dmем_we_x : std_logic;
    signal ahb_dmем_bit_wr_x : std_logic_vector(31 downto 0);
    signal ahb_imem_addr : std_logic_vector(9 downto 0);
end architecture;
```

```

signal ahb_imem_we_x : std_logic_vector(3 downto 0);
signal ahb_imem_en_x : std_logic_vector(3 downto 0);
signal ahb_cntl_we : std_logic;

-- latched control signals
signal HADDR_reg : std_logic_vector(31 downto 0);
signal HWRITE_reg : std_logic;
signal HTRANS_reg : std_logic_vector(1 downto 0);

signal ahb_dmem_addr_reg : std_logic_vector(10 downto 0);
signal ahb_dmem_en_x_reg : std_logic;
signal ahb_dmem_we_x_reg : std_logic;
signal ahb_imem_addr_reg : std_logic_vector(9 downto 0);
signal ahb_imem_we_x_reg : std_logic_vector(3 downto 0);
signal ahb_imem_en_x_reg : std_logic_vector(3 downto 0);
signal ahb_dmem_bit_wr_x_reg : std_logic_vector(31 downto 0);

begin

-- as the control signals are simply just ORed together (active high,active low
-- signals are ANDed), the idle side control signals must be kept in inactive state
dmem_en_x <= move_dmem_en_x and ahb_dmem_en_x; -- active low -> AND
dmem_wr_x <= move_dmem_we_x and ahb_dmem_we_x; -- active low
dmem_bit_wr_x <= move_dmem_bit_wr_x and ahb_dmem_bit_wr_x; -- active low

imem_data_out <= HWDATA;
imem_we_x <= ahb_imem_we_x;
imem_en_x <= move_imem_en_x&move_imem_en_x&move_imem_en_x&move_imem_en_x and ahb_imem_en_x;
-- NOTE: if move_imem_en_x = '0' then all imem arrays must be enabled...

cntl_data_out <= HWDATA;
cntl_we <= ahb_cntl_we;

-- send interrupt request when the MSB of the control register is '1'
IRQ_out <= cntl_data_in(31);

-- MUX that connects address to instruction memory
mux_imem_addr : process(move_imem_en_x, ahb_imem_addr, move_imem_addr)
begin
    case move_imem_en_x is
        when '0' => imem_addr <= move_imem_addr;
        when '1' => imem_addr <= ahb_imem_addr;
    end case;
end process;

-- MUX that connects address to data memory
mux_dmem_addr : process(move_dmem_en_x, ahb_dmem_addr, move_dmem_addr)
begin
    case move_dmem_en_x is
        when '0' => dmem_addr <= move_dmem_addr;
        when '1' => dmem_addr <= ahb_dmem_addr;
    end case;
end process;

-- this process latches the AHB control signals
regs : process (clk, rstx)
begin
    if rstx = '0' then
        HADDR_reg <= (others => '0');
        HWRITE_reg <= '0';
        HTRANS_reg <= "00";
    elsif clk'event and clk = '1' then
        HTRANS_reg <= HTRANS;
        if HTRANS /= "00" then -- HTRANS not IDLE, latch new control signal values
            HADDR_reg <= HADDR;
            HWRITE_reg <= HWRITE;
        else -- HTRANS = IDLE -> keep existing control signal values
            HADDR_reg <= HADDR_reg;
            HWRITE_reg <= HWRITE_reg;
        end if;
    end if;
end process;

-- MUX: this process selects the data to be written to data memory
dmem_data : process(move_dmem_we_x, move_dmem_d_in, HWDATA)
begin
    if move_dmem_we_x = '0' then
        dmem_data_out <= move_dmem_d_in;
    else
        dmem_data_out <= HWDATA;
    end if;
end process;

-- this process selects the appropriate control signals
control : process (HWRITE_reg, HADDR_reg, HTRANS_reg, HWRITE, HADDR, HTRANS, dmem_data_in, imem_data_in,
    cntl_data_in, move_imem_en_x)
begin
    if HWRITE_reg = '1' and HTRANS_reg /= "00" then -- write was initiated on the previous clock cycle
        -- however, HWDATA was not valid until now -> enable appropriate memory
        case HADDR_reg(17 downto 16) is -- select base address (dmem, imem, or cntl_reg)
            when "00" => -- write to dmem
                -- enabled
                ahb_dmem_addr <= HADDR_reg(12 downto 2);
                ahb_dmem_en_x <= '0';
                ahb_dmem_we_x <= '0';
                ahb_dmem_bit_wr_x <= (others => '0');
                -- disabled
                ahb_imem_addr <= (others => '0'); -- disabled items default to '0'
                ahb_imem_we_x <= "1111"; -- do not write to any of the 32-bit imem arrays
                ahb_imem_en_x <= "1111"; -- AHB does not need to enable imem arrays
                ahb_cntl_we <= '0';
            when "01" => -- write to imem
                -- enabled
                ahb_imem_addr <= HADDR_reg(13 downto 4); -- bits 1..0 select the 32-bit array
                -- ARM indexes memory in 8-bit words, where MOVE indexes 128-bit words
                case HADDR_reg(3 downto 2) is -- select imem array, one-hot encoding
                    -- 32-bit memory arrays
                    when "11" => -- LSB comes last
                        ahb_imem_we_x <= "1110";
                        ahb_imem_en_x <= "1110";
                    when "10" =>
                        ahb_imem_we_x <= "1101";
                        ahb_imem_en_x <= "1101";
                    when "01" =>
                        ahb_imem_we_x <= "1011";
                end case;
            end case;
        end case;
    end if;
end process;

```

```

        ahb_imem_en_x <= "1011";
    when "00" => -- MSB comes first
        ahb_imem_we_x <= "0111";
        ahb_imem_en_x <= "0111";
    when others =>
        ahb_imem_we_x <= "1111";
        ahb_imem_en_x <= "1111";
    end case;
    -- disabled
    ahb_dmem_addr <= (others => '0');
    ahb_dmem_en_x <= '1';
    ahb_dmem_we_x <= '1';
    ahb_dmem_bit_wr_x <= (others => '1');
    ahb_cntl_we <= '0';
when "10" => -- write to control register
    if HADDR_reg(15 downto 0) = X"0000" then -- actually write to control register
        -- enabled
        ahb_cntl_we <= '1';
    else
        ahb_cntl_we <= '0';
    end if;
    -- disabled
    ahb_dmem_addr <= (others => '0');
    ahb_dmem_en_x <= '1';
    ahb_dmem_we_x <= '1';
    ahb_dmem_bit_wr_x <= (others => '1');
    ahb_imem_addr <= (others => '0'); -- disabled items default to '0'
    ahb_imem_we_x <= "1111"; -- do not write to any of the 32-bit imem arrays
    ahb_imem_en_x <= "1111";
when others => -- erroneous address -> all control signals disabled
    ahb_cntl_we <= '0';

    ahb_dmem_addr <= (others => '0');
    ahb_dmem_en_x <= '1';
    ahb_dmem_we_x <= '1';
    ahb_dmem_bit_wr_x <= (others => '1');
    ahb_imem_addr <= (others => '0'); -- disabled items default to '0'
    ahb_imem_we_x <= "1111"; -- do not write to any of the 32-bit imem arrays
    ahb_imem_en_x <= "1111";
end case;

elsif HWRITE = '0' and HTRANS /= "00" then
    -- write was not initiated on the previous clock cycle and is being initiated now
    -- read address is directed immediately to AHB slaves so that we can receive
    -- the read data from the slaves within one clock cycle
    case HADDR(17 downto 16) is -- select memory base (0x80000000, 0x80010000, or 0x80020000)
    when "00" => -- read from data memory
        -- enabled
        ahb_dmem_addr <= HADDR(12 downto 2); -- index 32-bit words
        if HTRANS /= "00" then
            ahb_dmem_en_x <= '0';
        else
            ahb_dmem_en_x <= '1';
        end if;
        ahb_dmem_we_x <= '1';
        ahb_dmem_bit_wr_x <= (others => '1');
        -- disabled
        ahb_imem_addr <= (others => '0'); -- disabled items default to '0'
        ahb_imem_we_x <= "1111"; -- do not write to any of the 32-bit imem arrays
        ahb_imem_en_x <= "1111"; -- AHB does not need to enable imem arrays
        ahb_cntl_we <= '0';
    when "01" => -- Read from instruction memory
        -- enabled
        ahb_imem_addr <= HADDR(13 downto 4); -- bits 1..0 select the 32-bit array
        -- NOTE: 8-bit address -> 32-bit <<2
        if HTRANS /= "00" then
            case HADDR(3 downto 2) is -- select source imem array, one-hot encoding
            when "11" => -- LSB comes last
                ahb_imem_we_x <= "1111";
                ahb_imem_en_x <= "1110";
            when "10" =>
                ahb_imem_we_x <= "1111";
                ahb_imem_en_x <= "1101";
            when "01" =>
                ahb_imem_we_x <= "1111";
                ahb_imem_en_x <= "1011";
            when "00" => -- MSB comes first
                ahb_imem_we_x <= "1111";
                ahb_imem_en_x <= "0111";
            when others =>
                ahb_imem_we_x <= "1111";
                ahb_imem_en_x <= "1111";
            end case;
        else
            ahb_imem_we_x <= "1111";
            ahb_imem_en_x <= "1111";
        end if;
        -- disabled
        ahb_dmem_addr <= (others => '0');
        ahb_dmem_en_x <= '1';
        ahb_dmem_we_x <= '1';
        ahb_dmem_bit_wr_x <= (others => '1');
        ahb_cntl_we <= '0';
    when "10" => -- read from control register
        -- disabled (control register is always enabled)
        ahb_cntl_we <= '0';
        ahb_dmem_addr <= (others => '0');
        ahb_dmem_en_x <= '1';
        ahb_dmem_we_x <= '1';
        ahb_dmem_bit_wr_x <= (others => '1');
        ahb_imem_addr <= (others => '0'); -- disabled items default to '0'
        ahb_imem_we_x <= "1111"; -- do not write to any of the 32-bit imem arrays
        ahb_imem_en_x <= "1111";
    when others => -- all control signals disabled
        ahb_cntl_we <= '0';
        ahb_dmem_addr <= (others => '0');
        ahb_dmem_en_x <= '1';
        ahb_dmem_we_x <= '1';
        ahb_dmem_bit_wr_x <= (others => '1');
        ahb_imem_addr <= (others => '0');
        ahb_imem_we_x <= "1111";
        ahb_imem_en_x <= "1111";
    end case;
else -- neither write nor read is being processed -> disable all control signals
    ahb_dmem_addr <= (others => '0');
    ahb_imem_addr <= (others => '0');

```

```

    ahb_cntl_we <= '0';
    ahb_dmem_en_x <= '1';
    ahb_dmem_we_x <= '1';
    ahb_dmem_bit_wr_x <= (others => '1');
    ahb_imem_we_x <= "1111";
    ahb_imem_en_x <= "1111";
end if;

-- the situation when the bus will be busy
if HWRITE_reg = '1' and HWRITE = '0' then
    HREADY <= '0';
else
    HREADY <= '1';
end if;

end process;

-----
-- Connect appropriate memory input to HRDATA bus according to memory base address from
-- previous clock cycle.
-----
process(HWRITE_reg, HADDR_reg, dmem_data_in, imem_data_in, cntl_data_in)
begin
    -- connect the read data to HRDATA output
    if HWRITE_reg = '1' then
        -- write was initiated, data won't be read, default to '0'
        HRDATA <= (others => '0');
    else -- read operation was initiated on the previous clk cycle
        -- read data from a correct source must be delivered to HRDATA
        case HADDR_reg(17 downto 16) is --_reg(17 downto 16) is -- find out the source
            when "00" => -- data memory
                HRDATA <= dmem_data_in;
            when "01" => -- instruction memory
                case HADDR_reg(3 downto 2) is
                    when "00" =>
                        HRDATA <= imem_data_in(127 downto 96);
                    when "01" =>
                        HRDATA <= imem_data_in(95 downto 64);
                    when "10" =>
                        HRDATA <= imem_data_in(63 downto 32);
                    when "11" =>
                        HRDATA <= imem_data_in(31 downto 0);
                    when others => -- never occurs, for synthesis tool only
                        HRDATA <= imem_data_in(127 downto 96);
                end case;
            when "10" => -- control register
                HRDATA <= cntl_data_in;
            when others => -- should not occur
                HRDATA <= (others => '0');
        end case;
    end if;
end process;
end;

```

APPENDIX C

C-Source Code of the Test Program

```
/*
 * C Code for a simple DCT32 application utilizing a TTA processor
 *
 * TUT / DCS, FlexDSP
 * Miika Niiranen <miika.niiranen@tut.fi>
 *
 * Created: 2004-07-08
 * Last modified: 2004-08-18
 *
 * NOTES:
 *
 * TTA processor instruction width is 128 bits and data width 32 bits.
 * IMEM address width is 10 bits, while DMEM address is 11 bits wide.
 * The instruction memory and data memory must be initialized prior
 * to initiating DCT calculation. Otherwise, an undefined error will
 * result.
 *
 * Modifications:
 * * Added FIQ handler function. Removed control register polling loop.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "move_defs.h"
#include "arm_stripe.h"

#define DEBUG 1

int DEBUG; // global debug on/off variable (controlled by menu item #5)

// pointers to memory locations
int *move_dmem = (int*)MOVE_DATA_MEM_ADDR; // 0x80000000
int *move_imem = (int*)MOVE_IMEM_ADDR; // 0x80010000
int *move_ctrl_reg = (int*)MOVE_CTRL_REG_ADDR; // 0x80020000

char *input;
char *data; // received line

void load_instructions(void);
void load_data(void);
void do_dct(void);
void read_dct_data(void);
void write_to_control_reg(void);
void read_from_control_reg(void);
void read_from_imem(void);
int char_vector_to_int(char* vector);

// Process pending FIQ interrupt
// In a real application, this function would probably do something useful... =)
void FIQ_handler(void)
{
    // Clear INT_PLD[0] interrupt by writing to a PLD slave register to deactivate that signal
    *move_ctrl_reg = MOVE_LOCK;

    // notify the user through UART
    printf("Interrupt received. DCT data is ready.\r\n");
    return;
}

int main() {
    DEBUG = 0;

    *move_ctrl_reg = MOVE_LOCK; // lock the HW accelerator

    printf("\r\n");
    printf("*****\r\n");
    printf("A simple test program for ARM922T / TTA DCT design *\r\n");
    printf("*****\r\n");

    while (1) {
        printf("\r\nAvailable options:\r\n");
        printf("1) Load instructions\r\n2) Load data\r\n");
        printf("3) Start processing\r\n4) Read processed data\r\n");
        if (!DEBUG) {
            printf("5) Enable debugging\r\n");
        }
        if (DEBUG) {
            printf("5) Disable debugging\r\n");
            printf("6) Write to TTA DCT32 control register\r\n");
            printf("7) Read TTA DCT32 control register\r\n");
            printf("8) Read instruction memory contents\r\n");
        }
        printf("0) Quit\r\n");
        printf("\r\nPress a number and ENTER to continue... > ");
        gets(input);

        if (input[0] == '1') {
            load_instructions();
            printf("\r\nInstructions loaded.\r\n");
        }
        else if (input[0] == '2') {
            load_data();
            printf("\r\nData loaded.\r\n");
        }
        else if (input[0] == '3') {
            do_dct();
        }
    }
}
```

```

        printf("\r\nDCT processing started.\r\n");
    }
    else if (input[0] == '4') {
        read_dct_data();
        gets(input);
    }
    else if (input[0] == '5') {
        if (DEBUG) {
            DEBUG = 0;
            printf("\r\nDebugging disabled.\r\n");
        }
        else {
            DEBUG = 1;
            printf("\r\nDebugging enabled.\r\n");
        }
    }
    // The following three options are only available when debugging is enabled,
    // because the user should not be able to write to the control register or
    // to read instruction memory during normal operation.
    else if (input[0] == '6' && DEBUG) {
        write_to_control_reg();
        printf("\r\nControl register written.\r\n");
    }
    else if (input[0] == '7' && DEBUG) {
        read_from_control_reg();
        printf("\r\nControl register read.\r\n");
    }
    else if (input[0] == '8' && DEBUG) {
        read_from_imem();
        printf("\r\nInstruction memory contents read.\r\n");
    }
    else if (input[0] == '0') {
        printf("\r\nQuitting program...\r\n");
        return 0;
    }
    else {
        printf("\r\nInvalid input.\r\n");
    }
}
}
//-----
// This function receives instructions from UART and updates instruction memory
// accordingly. Instructions must be loaded in four 32-bit groups due to HW
// restrictions.
void load_instructions() {
    int index = 0; // counts the number of received instruction vectors
    int subindex = 0; // counts the four 32-bit groups (4x32=128)
    int data_int3, data_int2, data_int1, data_int0;

    // all char vectors are terminated with NULL (0) as a last character
    char data3[32], data2[32], data1[32], data0[32];
    char temp1, temp2;

    move_imem = (int*)MOVE_IMEM_ADDR;

    printf("\r\nSend instruction memory contents and press ENTER\r\n");

    while (getchar() == 34 && index < 1024) { // first char = '"'
        scanf("%32c%32c%32c%32c%c%c", data3, data2, data1, data0, temp1, temp2);

        data3[32] = 0; // terminate strings with NULL for char_vector_to_int conversion
        data2[32] = 0;
        data1[32] = 0;
        data0[32] = 0;

        if (DEBUG) {
            printf("\r\nDEBUG: data3 = %s", data3);
            printf("\r\nDEBUG: data2 = %s", data2);
            printf("\r\nDEBUG: data1 = %s", data1);
            printf("\r\nDEBUG: data0 = %s\r\n", data0);
        }

        data_int3 = char_vector_to_int(data3);
        data_int2 = char_vector_to_int(data2);
        data_int1 = char_vector_to_int(data1);
        data_int0 = char_vector_to_int(data0);

        if (DEBUG) {
            printf("\r\nDEBUG: data_int3 = 0x%X, data_int2 = 0x%X", data_int3, data_int2);
            printf("\r\nDEBUG: data_int1 = 0x%X, data_int0 = 0x%X\r\n", data_int1, data_int0);

            printf("\r\nDEBUG: move_imem = 0x%p\r\n\r\n", move_imem);
        }

        *move_imem = data_int3;
        move_imem += 1;
        *move_imem = data_int2;
        move_imem += 1;
        *move_imem = data_int1;
        move_imem += 1;
        *move_imem = data_int0;
        move_imem += 1;

        //scanf("%c%c%c%32c%32c%32c%32c%c%c", temp0, temp0, temp0, data3, data2, data1, data0, temp1, temp2);
        scanf("%c%c", temp1, temp2); // empty rx buffer before reading new line
        index++;
    }
    return;
}
//-----
// This function loads the data to be transformed into data memory.
// Works OK, 04-07-27
void load_data() {
    int index = 0; // counts the number of received data vectors (to be safe)
    int data_int; // holds the data to be written (in integer format)
    move_dmem = (int*)MOVE_DATA_MEM_ADDR; // initialize pointer to dmem
    printf("\r\nSend data memory contents and press ENTER\r\n");
    gets(data); // get char vector (entire line)
    data[32] = 0; // insert null termination
    while (*data != 10 && *data != 13 && index < 2048) { // double-check the table index
        // 10 = LF, 13 = CR (see ASCII table)
        data_int = char_vector_to_int(data); // convert char-type "bit" vector to integer
        *move_dmem++ = data_int; // write data to dmem and increment pointer after writing
    }
}

```

```

        if (DEBUG) {
            printf("\r\nDEBUG: Received line: %s = 0x%x\r\n", data, data_int);
        }
        index++;
        gets(data); // get the next char vector (entire line..)
        data[32] = 0; // .. and insert null termination
    }
    move_dmemb = (int*)MOVE_DATA_MEM_ADDR; // initialize pointer to the beginning of the dmemb
    return;
}
//-----
void do_dct() {
    *move_ctrl_reg = MOVE_INIT;
    if (DEBUG) {
        printf("\r\nDEBUG: *move_ctrl_reg = 0x%X\r\n", *move_ctrl_reg);
    }
    return;
}
//-----
// This function reads 1024 (max. 2048) 32-bit words from move data memory.
void read_dct_data() {
    int index = 0;
    move_dmemb = (int*)MOVE_DATA_MEM_ADDR;
    printf("\r\nEnable logging and press ENTER when ready to receive...\r\n");
    gets(input);
    while (index < 2048) { // this line defines the amount of words read
        printf("%u\r\n", *move_dmemb);
        move_dmemb++;
        index++;
    }
    move_dmemb = (int*)MOVE_DATA_MEM_ADDR; // reset the pointer
    return;
}
//-----
// This function writes a 32-bit word to the TTA DCT32 control register
void write_to_control_reg() {
    int data_to_ctrl_reg;
    char temp; // removes <ENTER> from input stream
    move_ctrl_reg = (int*)MOVE_CTRL_REG_ADDR; // 0x0002FFFF
    printf("\r\nData (decimal) to be written to control register: ");
    scanf("%u%c", &data_to_ctrl_reg, temp); // receive a 32-bit hexadecimal integer
    *move_ctrl_reg = data_to_ctrl_reg; // write received data to control reg
    printf("\r\nValue %u was written to control register.", data_to_ctrl_reg);
    return;
}
//-----
// This function reads the contents of the TTA DCT32 control register
void read_from_control_reg() {
    move_ctrl_reg = (int*)MOVE_CTRL_REG_ADDR; // 0x0002FFFF
    printf("\r\nControl register contains data = 0x%X at address = 0x%p",
        *move_ctrl_reg, move_ctrl_reg); // address should always be 0x2FFFF
    return;
}
//-----
// This function reads 256 (max. 512) 128-bit words in 32-bit pieces
// from MOVE IMEM (from address 0 onwards)
void read_from_imemb() {
    char *temp;
    int index = 0;
    int array3, array2, array1;
    move_imemb = (int*)MOVE_IMEM_ADDR;
    printf("\r\nEnable logging and press ENTER when ready to receive...\r\n");
    gets(input);
    while (index < 512) { // this line defines the amount of words read
        array3 = *move_imemb; // get the first 32 bits
        move_imemb += 1;
        array2 = *move_imemb; // ...
        move_imemb += 1;
        array1 = *move_imemb;
        move_imemb += 1;
        if (DEBUG) {
            printf("[0x%X] %X %X %X\r\n", index, array3, array2, array1, *move_imemb);
        }
        else {
            printf("%X %X %X %X\r\n", array3, array2, array1, *move_imemb);
        }
        move_imemb += 1;
        index++;
    }
    move_imemb = (int*)MOVE_IMEM_ADDR; // reset the pointer
    gets(temp);
    return;
}
//-----
// This function converts either a 32-bit or a 128-bit char-type
// bit vector to a double. Requires vectors terminated with NULL
// as inputs. Tested, works OK. 04-07-27
int char_vector_to_int(char* vector) {
    int x; // for-loop variable
    long double data_int = 0; // stores the integer value of the bit vector
    int i = 0; // index variable

    while (vector[i] != 0) {
        // while not NULL (terminate str), check the amount of bits (32 or 128)
        i++;
    }
    switch (i) {
        case 32: // bit vector is 32 bits long
            for (x = 0; x < 32; x++) {
                // convert from char to double and calculate bit weight
                data_int += (vector[x] - 48) * pow(2, (31-x));
            }
            break;
        default: // should not occur, as the bit width should be 32 or 128
            printf("\r\nERROR: Undefined vector range of %i bits.\r\n", i);
            data_int = -1;
            break;
    }
    return (unsigned int)data_int;
}
// unsigned int required because we deal with large positive numbers (>2^31)
}

```