NTNU – Trondheim
Norwegian University of
Science and Technology

Integration of hardware accelerators on the SHMAC platform

Marton Leren Teilgård

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Per Gunnar Kjeldsberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Assignment text

Candidate name: Marton Leren Teilgrd

Assignment title: Integration of hardware accelerators on the SHMAC platform

Supervisor: Per Gunnar Kjeldsberg

Assignment text:
The Single-ISA Heterogeneous MAny-core Computer (SHMAC) is an ongoing research project within the Energy Efficient Computing Systems (EECS) strategic research area at NTNU. SHMAC is planned to run in an FPGA and be an evaluation platform for research on heterogeneous multi-core systems. Due to battery limitations and the so called Dark silicon effect, future computing systems in all performance ranges are expected to be power limited. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited heterogeneous systems.
In an autumn project, a first methodology for accelerator selection and design for the SHMAC platform has been developed. There are multiple ways for such accelerators to be integrated on the SHMAC platform.
The main parts of this assignment are as follows:

- Study different approaches found in the literature for integration of hardware accelerators on heterogeneous multiprocessor platforms.
- Discuss different possibilities for accelerator integration on the SHMAC platform.
- Develop one or more technique for integration of the accelerators on the SHMAC platform and evaluate the effectiveness through implementation.
- Evaluate performance, energy, and area overhead for alternative accelerator integration techniques.

Abstract

The historical trend of rampant processor performance gain has slowed down in recent years due to the Dark Silicon Effect, which arises when Moore's law meets the breakdown in Dennard scaling for sub-130nm architectures. This effect has caused the industry to move into heterogeneous multicore architectures in an attempt to utilize this ”dark silicon”. Heterogeneous systems offer the ability to increase performance in applications by implementing accelerators specifically designed to the application. The SHMAC project aims to create a platform for research into heterogeneous systems, where an FPGA platform can provide quick implementation of systems for evaluation.

This thesis proposes a system where an accelerator quickly can be implemented into the SHMAC platform through a set of three different Interface Modules(IFM), and be controlled by the Amber Core through instructions in the ARM ISA. Furthermore the thesis proposes a script based system that generates an accelerated Amber Tile ready for integration into the SHMAC platform. Accelerators that are to be implemented with the IFMs need to meet specific criteria for interface. In order to define these, this thesis proposes a General Accelerator Interface designed to accommodate a wide range of diverse accelerators.

Sammendrag

Historisk sett har utviklingen innen prosessorytelse steget raskt og jevnt. I den senere tid har derimot ”The Dark Silicon Effect”(Den mrke silisium-effekten) bremset utviklingen. Denne effekten oppstr nr Moore’s lov og sammenbruddet i Dennard skalering mtes i produksjonsmetoder under 130nm. Prosessorindustrien prver motvirke denne utviklingen ved utvikle heterogene flerkjerneprosessorer. Heterogene systemer tilbyr muligheten til forbedre ytelsen i gitte oppgaver ved lage oppgavespesifikke akselleratorer som er i stand til utfre ofte gjentatte utregninger i oppgaven. SHMAC-prosjektet jobber med ml om utvikle en forskningsplattform for heterogene systemer der bruk av en FPGA kan gi rask implementering av foresltte systemer.

Denne oppgaven foreslr et system der en gitt akselllerator raskt kan implementeres i SHMAC-plattformen gjennom et sett av tre forskjellige grensesnittsmoduler(IFM) som kan kontrolleres av Amber-kjernen ved hjelp av instruksjoner i ARM ISAet. Videre foresls det ett skriptbasert system som genererer en Amber-flis som er klar for integrering med SHMAC-plattformen. Akselleratorer som skal implementeres med en IFM m mte spesifikke kriterier nr det kommer til grensesnitt. Denne oppgaven definerer et Generelt Akselllerator-grensesnitt laget for konkretisere disse kriteriene, samtidig som det er beregnet for vre brukbart for et bredt spekter av akselleratorer.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for a master's degree.

This work has been performed at the Department of Electronics and Telecommunications, NTNU, Trondheim, with Per Gunnar Kjeldsberg as supervisor.

Acknowledgements

I want thank first and foremost my supervisor Per Gunnar Kjeldsberg for his extensive support and guidance. Without his help I would still be lost in the jungle of literature and articles. I would also like to thank Yaman Umuroglu and Nikita Nikitin at the Department of Computer and Information Science for their tips, guidance and inspiration. I also need to thank Hkon Wikene for guiding me through the magic of the Makefile, Magnu Moreau for his help with the report and the boys at B121 for their companionship. Last but not least I would like to thank my girlfriend Ida Bakke for her moral support during the last couple of months.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ATF	Amber Test Framework
$CCore$	Conservation Core
$EECS$	Energy Efficient Computing Systems
$FIFO$	First In First Out
$FPGA$	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
FU	Functional Unit
IFM	InterFace Module
$SHMAC$	Single-ISA Heterogeneous MAny-core Computer
WMA	Wihbone Master Arbiter
WMB	Wishbone Master Buffer
ASIC	Application Specific Integrated Circuit
RISC	Reduced Instruction Set Computing
SoC	System-on-Chip

Chapter 1

Introduction

1.1 Motivation

In the 60 years that have passed since the first general purpose electronic computer was created, we have seen a massive increase in computing performance. The development has, until about 2002 followed Moore's law[1] with a performance growth of about 52% annually.

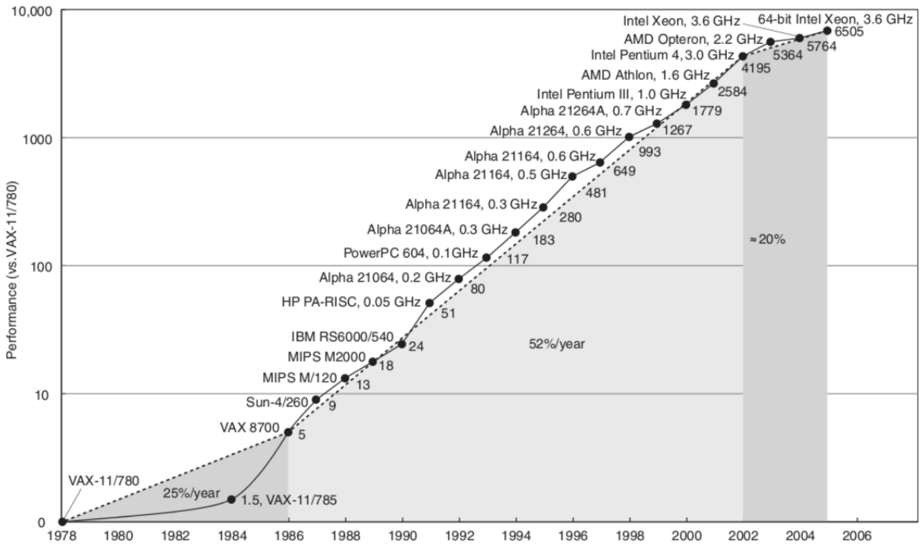
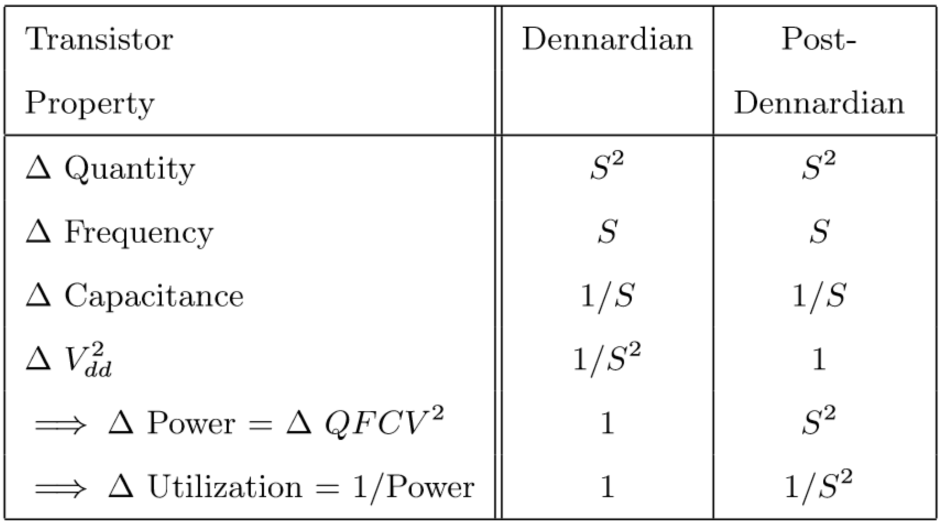


Figure 1.1: Historical Processor Performance [2, p. 3]

Figure 1.1 shows the historical development of processor performance from 1978 until

2005. As seen in the figure, the growth in processing power diminishes from 2002 and onwards. According to Hennessy and Patterson[2] the slowed growth rate is caused by the limitations of power, the available Instruction Level Parallelism and memory latency.

The limitations of power are an effect of the breakdown of Dennardian scaling[3]. The continued decrease in production size, in order to keep up with Moore's law, has historically been possible with a fixed power budget due to the effect known as Dennard, or Dennardian, scaling[4]. However, as production size has decreased past the 130 nm level, the power has moved in to the Post-Dennardian Scaling[3] shown in Figure 1.2.



Transistor Property	Dennardian	Post-Dennardian
Δ Quantity	S^2	S^2
Δ Frequency	S	S
Δ Capacitance	$1/S$	$1/S$
Δ V_{dd}^2	$1/S^2$	1
$\implies \Delta$ Power $= \Delta\ QFCV^2$	1	S^2
$\implies \Delta$ Utilization $= 1/$Power	1	$1/S^2$

Figure 1.2: Dennardian vs. Post Dennardian scaling [3]

This breakdown in power scaling causes what is called the Dark Silicon Effect[5]. This term means that not all transistors on a System on Chip(SoC) device can be powered at the same time due to the increased power demand and the difficulty to dissipate the heat related to the power increase. This has created a move in to heterogeneous multicore systems.

Ying Zhang et al.[6] proposes exploiting the dark silicon by utilizing heterogeneity in multicore processors. When there is a surplus of area that can not always be used, it is possible to create specialized hardware units that can perform specific calculations in order to accelerate certain applications. W.Wolf[7] describes an accelerator as a processing element connected to a general purpose processor, able to increase performance for specific applications. Recent work in the field of heterogeneous multicore systems move towards automated and highly generalized acceleration target detection and integration [8],[9],[10]. The SHMAC project at the Norwegian University of Science and Technology is part of this movement in to heterogeneous multicore systems.

1.2 the SHMAC project

The Single-ISA Heterogeneous MAny-core Computer(SHMAC) project is an ongoing project at the Energy Efficient Computing Systems(EECS) research area at the Norwegian University of Science and Technology. The project proposes an infrastructure for exploring heterogeneous systems at all abstraction levels in an attempt to answer, in regard to heterogeneous hardware, the research question:

- How should a heterogeneous processor architecture be designed in terms of core composition, accelerators, interconnect and memory system?

Rusten and Sortland[11] have created a prototype for the SHMAC project that implements a reconfigurable tile structure on an FPGA. The SHMAC project plan[12] describes how the development has continued further, implementing the Amber Tile that includes an ARM ISA[13] Amber Core[14], hosted by OpenCores[15]. This tile is the main processing tile for the project moving forward, and it is the tile that will control any accelerators implemented in the SHMAC platform. The most important aspects of the SHMAC platform, for this thesis, are explained in Chapter 3.

In order to effectively research the impact of accelerators in heterogeneous systems, it is vital to have a fast and easy way to implement and control accelerators of any type. This thesis proposes an answer to this problem with a set of memory mapped or on-core INterface Modules(IFM) and a Tile Generator that generates new SHMAC tiles with an IFM, modified from the Amber Tile.

1.3 Contributions

The thesis makes several contributions to the SHMAC project, listed below.

- a comprehensive definition of a General Accelerator Interface, which forms the basis for development of IFMs and future accelerators for the SHMAC project. (Chapter 4)
- several alternative options for integrating an accelerator on the SHMAC platform, through InterFace Modules(IFM).(Chapter 5)
- a Python script that generates a modified tile type for a given accelerator, named TileGenerator (Chapter 6)

1.4 Thesis Organisation

This thesis is organized in chapters. A brief description of each chapter is listed below.

Chapter 2: Theory explains concepts from literature relevant to this thesis as well as those components of the Amber Tile that are not developed in the SHMAC project.

Chapter 3: SHMAC Parent System gives an introduction to the SHMAC architecture and details the Amber Tile, which is the basis for the work in this thesis.

Chapter 4: Definition of a General Accelerator Interface describes the general accelerator interface developed in order to design the IFM modules

Chapter 5: Design of the Interface Modules explains in detail how the IFMs are designed, as well as an evaluation of different levels of accelerator integration

Chapter 6: IFM and tile scripted generation explains the design and functionality of the Tile Generator in detail

Chapter 7: Verification and Overhead explains how the designs were verified and details the overhead associated with the IFMs

Chapter 8: Using the IFM system contains a comprehensive user guide for the TileGenerator, the programming of the IFMs and the interface towards the accelerator. This Chapter is designed to help accelerator designers and programmers utilize the IFM system.

Chapter 9: Discussion and Future Work contains a discussion of the proposed IFMs and scripted tile generator, as well as suggestions on what can be improved further in future work.

Chapter 10: Conclusion draws conclusions based on the work described in this thesis.

Chapter 2

Theory

This chapter explains concepts from literature relevant to the work with this thesis, as well as the components of the Amber Tile not developed in the SHMAC project.

2.1 ARM Amber core

The ARM Amber core is a 32 bit RISC processor hosted at OpenCores [15], a website dedicated to develop and distribute open source hardware. It is fully compatible with the ARM v2a instruction set[14], and is an implementation of the ARM7 family. The general ARM7 core is shown in Figure 2.1.There are two versions of the Amber core, Amber23 and Amber25. The SHMAC platform incorporates the Amber25 core. Amber25 has a 5 stage pipeline, an internal 32-bit Wishbone bus and separate instruction and data caches. The Amber architecture supports 2, 3, 4 and 8 way caches, in the SHMAC implementation both cashes are 3-way.

2.1.1 pipeline architecture

Figure 2.2 shows the five stages of the Amber 25 pipeline.

The pipeline consists of the three stages in the Amber 23 structure, **Fetch**, **Decode** and **Execute**, with the addition of two more stages, **Memory** and **Write Back**. The **Fetch** stage does a cache try with the instruction address. If there is a cache miss, the Amber core is stalled while the cache fetches the instruction from the system memory. The **Decode** stage interprets the instruction and sets datapath control signals. The stage contains a Finite State Machine (FSM) to handle multi cycle instructions and interrupts. In the **Execute** stage any ALU operations are executed on the operands in the register bank. The next address for the Fetch stage is generated in this stage. The **Memory** stage handles any memory operations, and the **Write back** stage handles the update of the register bank with new data. A more detailed description of the pipeline can be found in the Amber Core Specification[14].

2.1.2 Instruction set

The ARM v2a instruction set[13] is an older version of the ARM instruction set. All instructions are one-cycle instructions, apart from the coprocessor instructions, multiply and multiply accumulate. The coprocessor register transfer instructions take two cycles. The Amber core utilizes Booth's algorithm to calculate the multiply instructions in a 34 cycle operation.

2.1.3 coprocessor support

The ARM v2a instruction set[13] supports several instructions for communication and control with coprocessors. However, in the Amber 25 core only a subset of these are implemented. There are already fully implemented instructions for passing data between the Amber core registers and a coprocessor. In addition the instruction set includes instructions for data transfer between coprocessor and memory and for passing operation arguments from the amber core to a coprocessor.

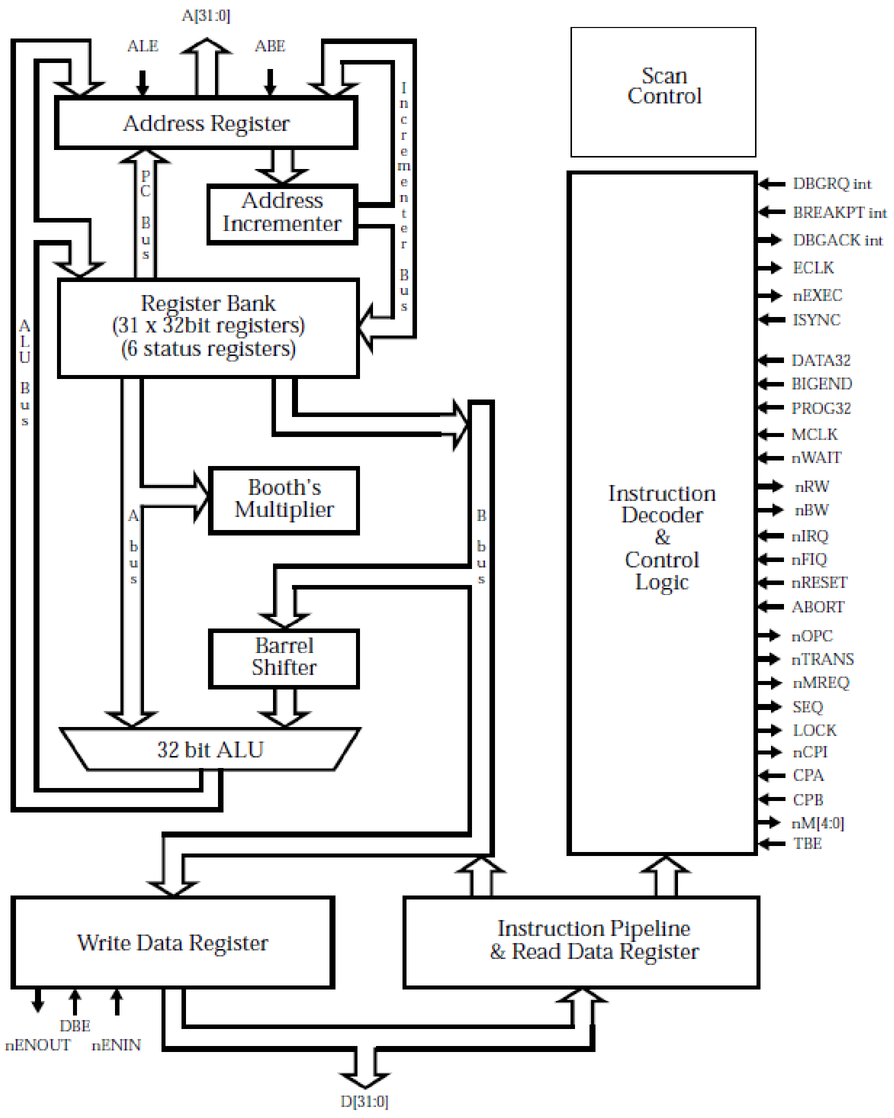


Figure 2.1: ARM7 core diagram. [13, p. 7]

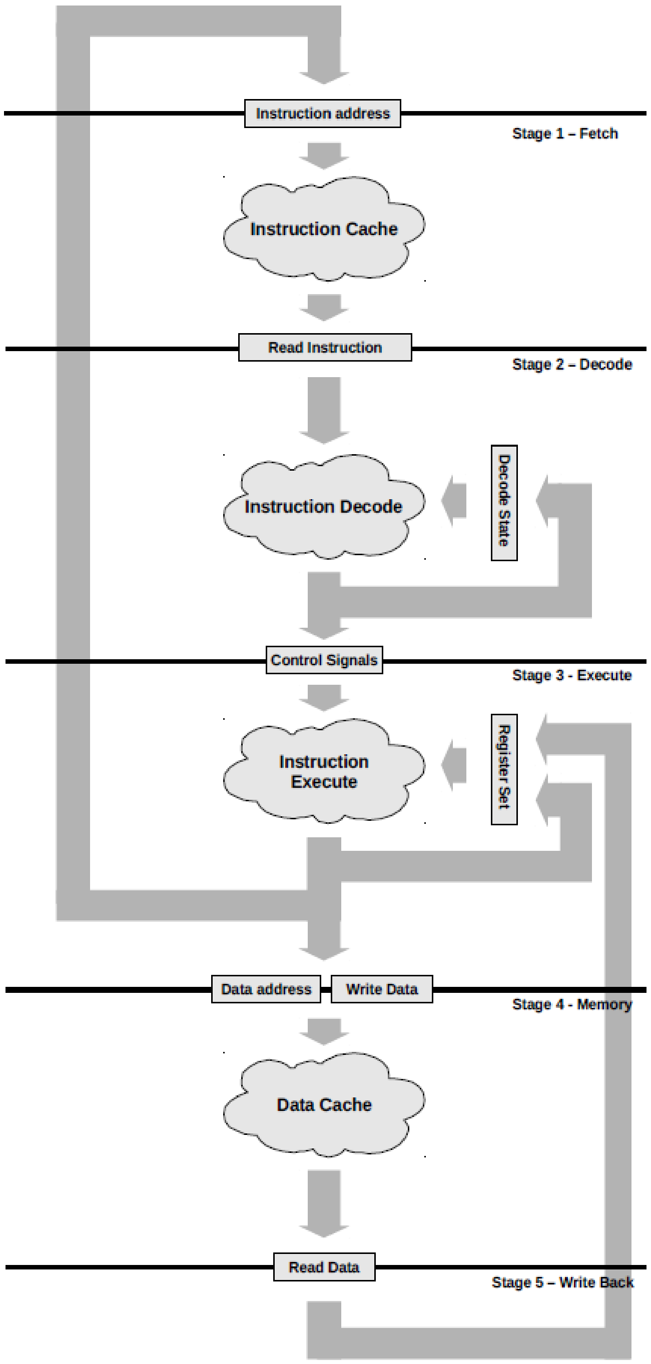


Figure 2.2: Amber 25 pipeline.[14, p. 5]

2.2 Wishbone bus

The Wishbone[16][17] bus is an open source System-on-Chip(SoC) Interconnect Architecture hosted by OpenCores[15]. The Wishbone bus is used both in the Amber core and in the tile architecture on the SHMAC platform. The Wishbone bus specification declares two bus interfaces, **Master** and **Slave**. The master interface is a core capable of initiating a bus cycle, the slave is a core capable of receiving a bus cycle. The connections between the different interfaces can be established in a large number of ways, examples include point-to-point transfers and arbitrated bus systems with several **Master** and **Slave** units. Figure 2.3 shows how two units, which both have **Master** and **Slave** interfaces can communicate.

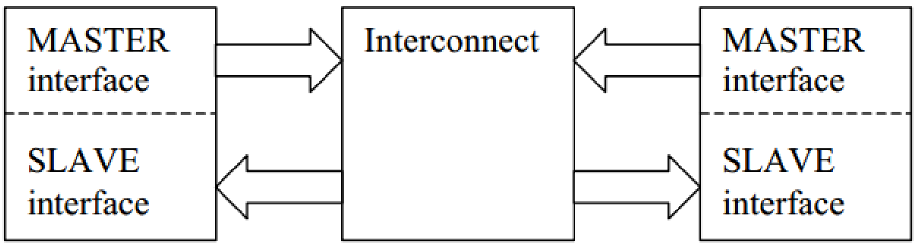


Figure 2.3: Connection between two modules with both **Master** and **Slave** Wishbone interfaces. [17, p. 1]

2.2.1 Wishbone Signals

The Wishbone bus system defines several signals common to all interfaces. The list below describes the signals most relevant to this thesis and the SHMAC system. Unless otherwise stated, the **Master** interface has the *SIGNAL_O* port and the **Slave** interface has the *SIGNAL_I* port.

ADR_I/ADR_O This is a bus signal that indicates the address of the data requested by the **Master**.

SEL_I/SEL_O This is a bus signal indicating where on the data port there will be valid data. The size of this bus is determined by the granularity of the data port.

WE_I/WE_O This signal indicates if the data transfer cycle is a write where data passing from **Master** to Slave, or a write where data is passed from **Slave** to **Master**

CYC_I/CYC_O This signal indicates a valid bus cycle in progress. This signal can be held high during a block transfer of several data transfers.

STB_I/STB_O This signal indicates a valid data transfer data cycle. In the case of the SHMAC platform, this signal is used in combination with the ADR_ signal to arbitrate between different modules with **Slave** interface.

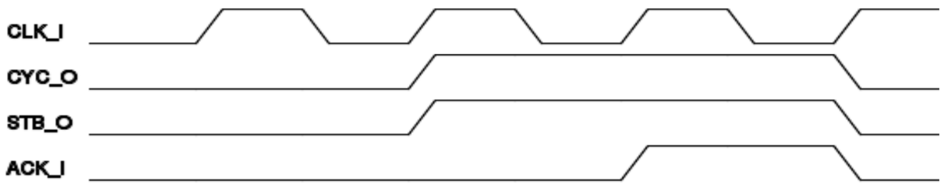


Figure 2.4: Standard transfer protocol, synchronous **Slave** [16, p. 33]

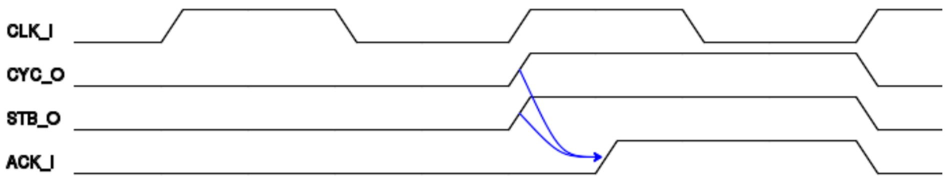


Figure 2.5: Standard transfer protocol, asynchronous **Slave** [16, p. 34]

ACK_I/ACK_O The ACK signals are reversed in comparison to the other signals, with the ACK_O in the **Slave** interface and the ACK_I in the **Master** interface. It is used to acknowledge a valid data transfer.

DAT_I/DAT_O Both **Master** and **Slave** interfaces have both of these ports. They are bus signals transferring the data in or out of the module, respectively.

2.2.2 Wishbone Cycle

The Wishbone specification defines two types of standard transfer protocols, synchronous and asynchronous **Slave**. Figures 2.4 and 2.5 shows these in waveforms, from the perspective of the **Master** interface.

In both cases the data is read and STB_O is negated on the first rising clock edge that the ACK_I signal is high. These standard transfer cycles are known as *SINGLE READ-/WRITE Cycles* in the Wishbone specification[16]. The Wishbone specification also defines another type of transfer cycle that is relevant to the SHMAC platform, the *BLOCK READ/WRITE Cycles*. Figure 2.6 shows a block transfer cycle. The **Master** can indicate such a block transfer by holding the CYC_O signal high after the STB_O signal is negated, and keeping it high through several consecutive transfer cycles.

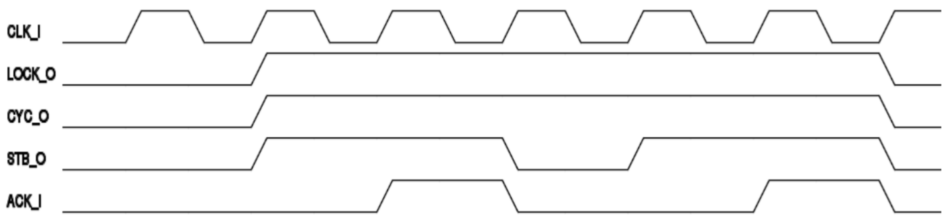


Figure 2.6: Block transfer protocol, synchronous **Slave** [16, p. 47]

2.3 Hardware Accelerators

Hardware accelerators are used in multiprocessor systems for a large number of applications, ranging from scientific and business applications to private computers and hand held devices. W. Wolf [7, p. 356] describes an accelerator as a **processing element** that can give large performance increases for applications that use a lot of time in small sections of code. This can be code that is exceptionally slow to execute, applications that only focus on one small task or code that is repeated often in for example a nested loop. Where W. Wolf[7] mostly focuses on speedup and execution time, G. Venkatesh et al. [8] focus on energy conservation, even at the cost of slower execution. Accelerators are described in both [7] and [8] as CPU-near Application Specific Integrated Circuits (ASIC) that do some computational work on data provided by the CPU or through memory bus.

Accelerator examples from theory

In order to create a general interface it is necessary to describe some examples of accelerators and try to find commonalities and general traits between them. I will here describe some accelerators found in literature.

W. Wolf[7] gives a general description of an accelerator without any specifics, as shown in Figure 2.7.

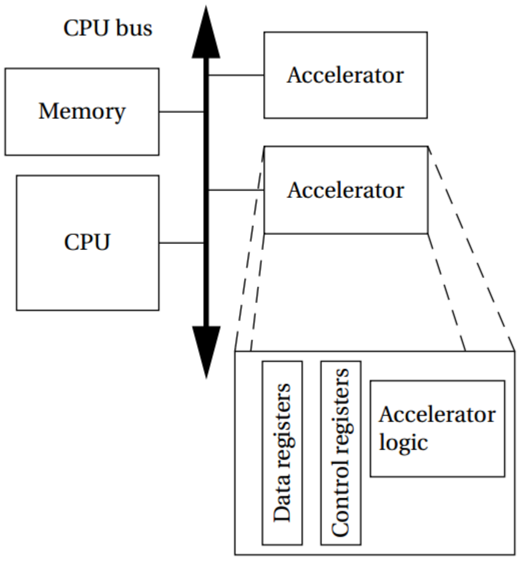


Figure 2.7: Accelerator description and interface. W. Wolf[7, p. 356]

This is an accelerator interfaced through the CPU bus as an I/O device. It uses control registers to accommodate a control interface for the CPU. This can be used to check states, set options and give start signals. In addition there are data registers, used to hold and pass data needed by the accelerator. The *accelerator logic* module in Figure 2.7 is designed to effectively execute a given task, for example a small section of code that is often used.

In the paper ”Quantifying Acceleration”[18] the authors B.Reagen et al. do not focus too much on the design of the accelerator, but they write that they focus on a type of accelerator that specializes in certain limited tasks, with it’s own memory system and created for the large design space.

Although the authors of the paper ”Conservation Cores: Reducing the Energy of Mature Computations”[8], G. Venkatesh et al. separates their Conservation Cores(CCore) from the term accelerators due to the focus on energy saving at the cost of performance and area, in essence they are the same thing. They implement their CCores as a set of ASICs closely connected to the CPU, with a shared cache. They are logical accelerators that implement a specialized hardware structure to calculate often used code.

In my semester project for the fall of 2013, ”Evaluation of basic block accelerators for use on the SHMAC platform” [19], co written with Sunniva Nergaard Berg, we look in to several accelerator targets. The project describes several possible acceleration targets from a set of benchmarks, and includes a design of a multiply-add accelerator for the SHMAC system. The accelerator in this project is a hardware implementation of a simple equation that the parent system[14] does slowly, with input and output variables stored in registers that are updated on every clock, in addition to a **start** port. To summarize, it is a specialized, clocked hardware unit that takes input data and control signals from a parent system and produces output data to be read back. The future work section of this paper also points to the possibility of adding more functionality, such as the ability to subtract in stead of adding, by setting control options from the parent system.

2.4 CPU to accelerator interface

There are many different ways of interfacing an accelerator from a CPU system in literature. The optimal interface is dependent on such issues as what the accelerator does, how fast it operates, the level of control necessary, how much data it requires, what infrastructure the CPU and the parent system use and what resources are available in terms of area and energy. I will now look into some examples of accelerator interfaces found in literature.

W. Wolf[7] gives in his book two examples of CPU to accelerator interface, both connected to the CPU bus and gives the CPU control through control registers. One of these uses direct data passing from the CPU to the accelerator and the other interface is able to access the memory directly through the bus, with only control signals passed from the CPU. The first type is shown in Figure 2.7, and the other is shown in Figure 2.8.

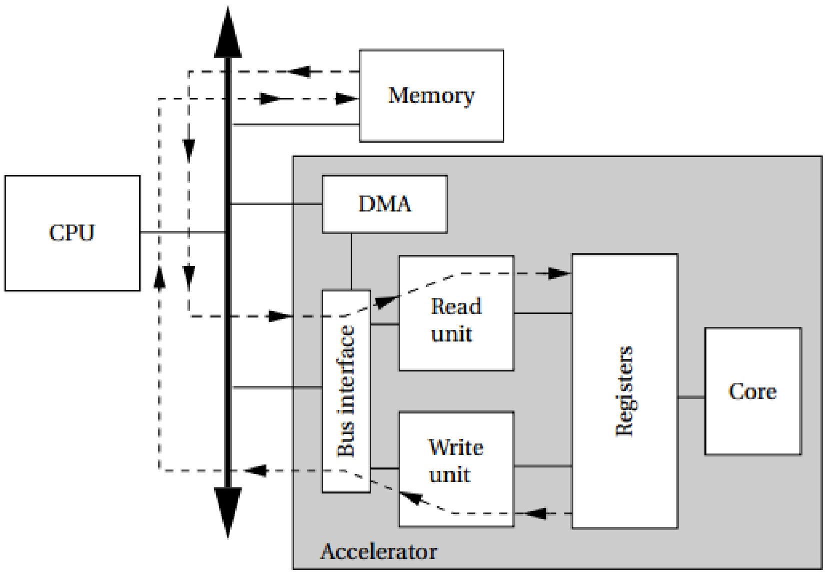


Figure 2.8: Accelerator interface with memory access. W. Wolf[7, p. 358]

The interface shown in Figure 2.8 has a higher complexity and requires more area, but the memory access makes it well suited for accelerators with a high data demand, as it frees up the CPU to work in parallel with the accelerator.

G.Venkatesh et al.[8] takes a slightly different approach in interfacing their CCores to the parent CPU. They use a more CPU near connection, where the accelerators, called CCores, share the L1 data cache with the CPU. To allow the parent CPU to control the accelerators they use Scan Chains[8].

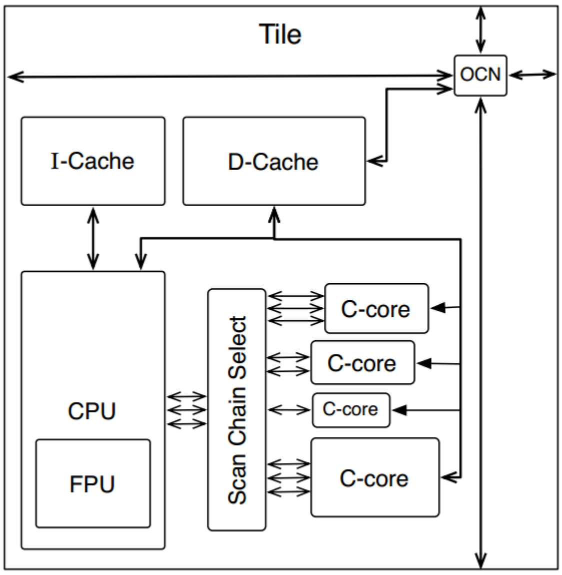


Figure 2.9: CPU tile with CCore interface. G. Venkatesh et al.[8, p. 3]

These scan chains gives the CPU a high level of control, giving it access to read and write arguments and states, perform context switching and patching of the accelerators. Figure 2.9 shows a tile with several CCores interfaced to one parent CPU. The tile shown in the figure is one of many in a multicore system.

In the paper called ”Dynamically Specialized Datapaths for Energy Efficient Computing” [9], the authors V. Govindaraju et al. take an even more core near approach to integrate their accelerators, referred to as DySER blocks, with the parent CPU. They place them in the pipeline of the CPU as a set of Functional Units(FU) and switches, controlled directly by the CPU’s Decode and Execute pipeline stages. In this way they insert highly specialized and controllable hardware directly into the CPU architecture, and the DySER blocks integrates completely in to the CPU module.

P. M. Stillwell jr. et al. proposes a very independent interface in the article ”HiPPAI: High Performance Portable Accelerator Interface” [10]. They utilize a high level application function and accelerators working in the virtual address domain with a special purpose hardware interface to control and use the accelerators. In the terms used earlier, these accelerators can almost be seen as specialized cores, or functional units on the SoC device.

Chapter 3

SHMAC parent system

In this chapter I will go through the SHMAC system, the Amber tile and the Amber core to provide the necessary understanding of the system for which the IFM system is meant. This chapter contains an overview of the previous work completed in the SHMAC project that are relevant for this thesis.

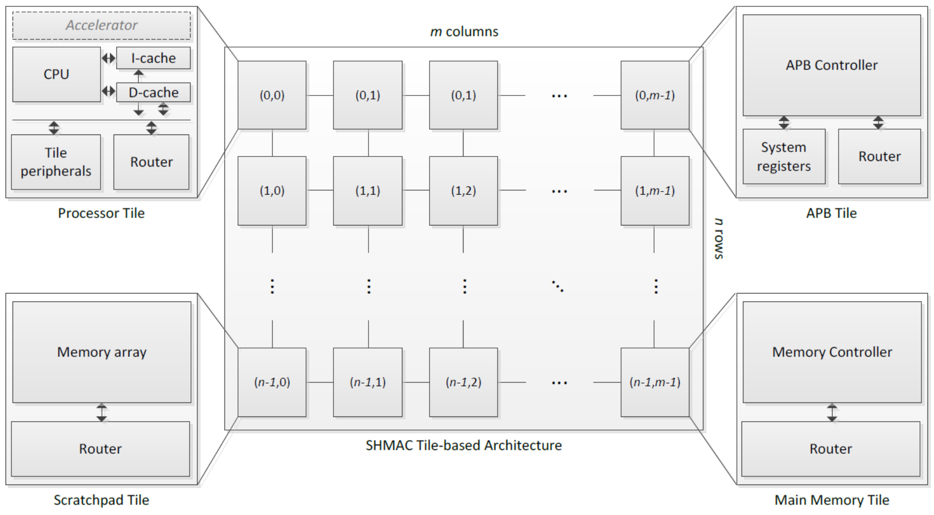


Figure 3.1: High level architecture SHMAC [12, p. 5]

3.1 SHMAC system overview

The Single-ISA Heterogeneous MAny-core Computer(SHMAC) project is a research project meant to provide a fast and easy way to develop and test heterogeneous processing systems. As stated in the SHMAC project plan[12], ”The key idea is to create a flexible framework in which different heterogeneous processors can be created from a collection of processing elements and accelerators.” This is a research project with a heavy focus on hardware/software co-design, meant to do research on heterogeneous systems at all abstraction levels. The SHMAC system is currently being developed on a FPGA platform, to enable fast testing and verification of different heterogeneous systems. The FPGA device used is a xc5vlx330 chip from the Virtex 5 family of Xilinx FPGAs.

SHMAC Tile Architecture

SHMAC is a tile based architecture, made up of several tile types laid out in a rectangular grid, with each tile identified by its position in the grid. The identification is on the form *(n,m)* where *n* denotes the row and *m* denotes the column. Figure 3.1 shows the tile structure of the SHMAC system.

The different tiles are connected to each other using a mesh interconnect system. This means that each tile has connections to its neighbours in up to four directions, north, south, east, west. An edge tile only has connections in the directions where there is neighbours, a non-edge tile has connections in all four directions. Every type of tile has a router that handles the flow of data.

The SHMAC project has the potential for a large amount of different tiles, but a small

Table 3.1: SHMAC system memory map

Start	End	Description
0000 0000	0000 001F	Exception Table
0000 0020	F7FF FFFF	Main Memory
F800 0000	FFFD FFFF	BRAM Memory
FFFE 0000	FFFE FFFF	Tile Registers
FFFF 0000	FFFF FFFF	System Registers

set of tile types are already defined and are described in the list below.

CPU core Also known as Amber Tile. The main processor tile of the SHMAC project, containing a modified Amber ARM compatible core and some extra units. See Section 3.2 for more details.

APB I/O tile, implements I/O support over an APB bus

Main Memory a tile that communicates with the ZBM RAM of the parent FPGA platform

Scratchpad This tile is a RAM tile using the FPGA internal BRAM

Table 3.1 shows the memory map of the SHMAC system.

3.2 Amber Tile

The Amber Tile is the main CPU tile in the SHMAC system[12]. A high level architecture schematic is shown in Figure 3.2. The tiles main components are a router and an Amber core. The router handles the off tile communication, in addition to passing packages. The Amber core is a modified version of the OpenCores Amber core described in Section 2.1. In addition to these two main components, the Amber tile includes a set of tile registers, an Interrupt controller and a Timer Module. The tile registers are used to store information about the specific tile, such as processor ID and tile coordinates. The Interrupt controller sorts all interrupt sources and relay interrupts to the Amber core. The Timer Module contains three separate timers that can be controlled by the Amber core and it is able to set interrupt signals through the interrupt controller.

The tile components are connected by a 128 bit Wishbone bus. The Amber core controls the different tile modules through memory mapping. The tile memory map is shown in Table 3.2.

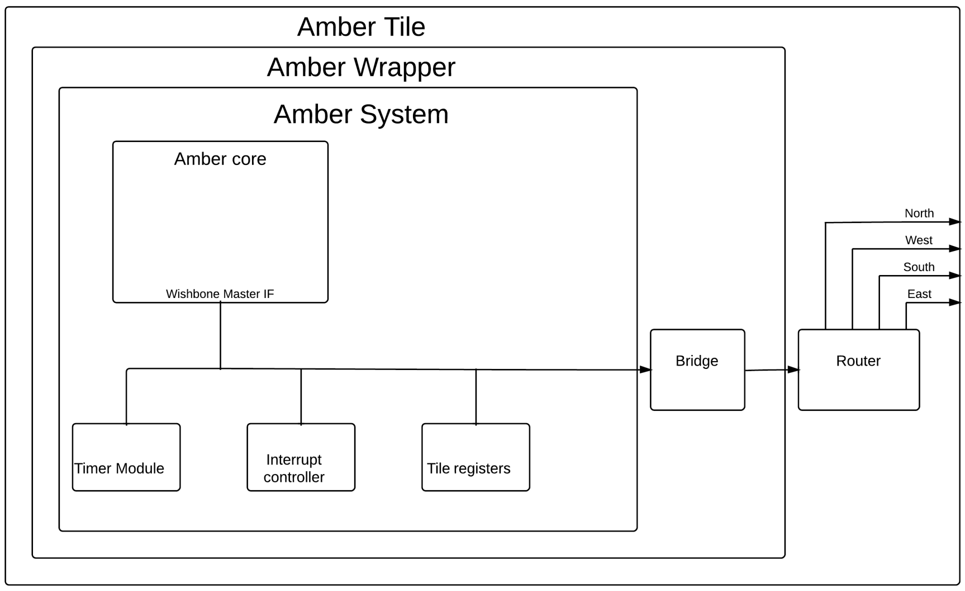


Figure 3.2: Overview of the Amber Tile

Table 3.2: Amber Tile memory map

Start	End	Description
FFFE 0000	FFFE 0FFF	Tile Register Module
FFFE 1000	FFFE 1FFF	Timer Module
FFFE 2000	FFFE 2FFF	Interrupt Controller

3.3 Development and test tools

Xilinx ISE

Xilinx ISE(Integrated Software Environment)[20] is a HDL synthesizer and design tool that is used in this project due to the fact that the project is using a Xilinx FPGA. ISE offers HDL synthesis, simulation, RTL diagram generation and timing analysis. ISE supports several OS platforms, including MAC OS, Windows and several Linux distributions. All development and most of the tests run during the work with this thesis has been done with Xilinx ISE.

ISim

In order to run simulations, Xilinx ISE uses the simulation tool ISim. ISim is an advanced HDL simulator with a project navigator and waveform window. It is used for verification and debugging of complex systems, with its main functionality tightly bound to the ISE environment and the Xilinx FPGAs.

Amber test framework

Included in the Amber OpenCores project there is a set of tests and testbenches meant to verify and debug the functionality of the core[21]. These have been modified further to work with the SHMAC project and are a part of the SHMAC repository. The Amber test framework uses a Linux based Gnu compiler and ISim to run the assembly level tests, verify and debug.

Chapter 4

Definition of a General Accelerator Interface

In order to create a general interface module for communicating with an accelerator, even though the goal of this thesis is to create a set of as general as possible IFMs, some limitations and definitions have to be made. In this section I will propose a definition for a general accelerator in terms of ports and explain the reasoning behind this definition. Communication cycles and timing waveforms for communication between accelerator and IFM can be found in Section 8.3. Any accelerator conforming with these definitions will be possible to implement in to the SHMAC platform through one or more of the IFMs proposed in this thesis, depending on the limitations of the specific IFM.

4.1 Interviews with SHMAC accelerator designers

One of the approaches taken in order to determine what is required in a accelerator interface was to interview several accelerator designers currently working in the SHMAC project. This section consist of descriptions of three of them.

Audun Lie Indegaards [22] accelerator is a Floating Point Unit (FPU) with variable word width and exception handling. At the time of synthesis the designer can determine the word width and the propagation time through the accelerator. It is a accelerator that takes input data and control signals including a start signal and some operational options from the parent system, calculates the values and sets a ready signal when it is done. It can be synthesized as a asynchronous module that can function with a propagation time of less than one cycle and so function without clock, but in most cases the accelerator is clocked.

Einar Johan Tran Smen[23] has created a accelerator with a very different functionality, namely a Game of Life simulator, that can amongst other things be used as a pseudo random number generator. It can be created in several sizes, decided at the time synthesis. It functions by shifting in input data and rule sets through a number of input ports and by using an options input to determine what type of data enters the accelerator. It also uses the options input to give start and read commands, and control which data is to be read.

Sunniva Nergaard Berg[24] is working on a accelerator that performs several arithmetic operations on two matrices nested in two for loops. It is part of an algorithm for Epileptic seizure prediction. It has large demands on memory access, as it requires 7 + 2048 32 bit input values,and produces 2008 32 bit output values per calculation. It uses a start signal to indicate start of operation, and signals completion with a ready signal.

4.2 General accelerator traits

Based on the accelerators described in Sections 4.1 and 2.3, I have outlined a set of traits common in accelerators. These are important to design the IFMs, and are described here.

Accelerators does computations on data As expected, but it is an essential characteristic for accelerators. Accelerators take input data, perform some form of calculation on the data and produce output data.

Accelerators execute a set operation This means that accelerators generally execute a very specific operation, and it does so every time it is used. Some accelerators implement the possibility to set options to give small variations on this operation, but in general these are very limited. Normally an accelerator implements a subset of instructions in the parent's ISA or a part of the application's high level code, but this is not necessarily the case.

Accelerators are controlled by a parent CPU All accelerators are in smaller or greater detail controlled by a CPU. This is normally done with control registers and signals such as interrupts, start signals, readable states and options. There are different ways of doing this, examples include memory mapped, connected to CPU bus, coprocessor interfaced and custom IO ports.

Most accelerators are clocked and have reset functionality Most accelerators require a clock signal, both to function internally and to communicate with the parent system. This is however a trait with some exceptions. Accelerators can be created as pure combinatorial circuits, where the propagation of data through the accelerator is not controlled by a clock. These are normally accelerators with functionality that always completes in a fixed time and with limited complexity and control options. The reset functionality is normal in all digital hardware, and is also common in accelerators.

4.3 General interface design

The design of the accelerator interface supported by the IFM structure is done in such a way that it is as general as possible, while giving concise constraints in order to make it possible to build functioning interface solutions for connection to the Amber system. The general traits of an accelerator, as defined in Section 4.2, is the basis for this design.

The accelerator interface is clocked and includes a reset port. Both of these will with the use of an IFM from this project be connected to the system clock and system reset signals. In addition to these system wide signals, the accelerator interface includes a start and a ready port used by the IFMs, and in turn the parent system, to control the accelerator and to give feedback to the system upon completion. These are one bit ports and the signal given to indicate a start signal is one high cycle, synchronous with the system clock. The ready signal is treated asynchronously by the two basic IFMs, with the system reacting to the rising edge of this signal. The For loop IFM reads this signal synchronously on the rising edge of the clock signal. This difference is included to allow the basic IFMs to work with asynchronous accelerators, as they do not require the same strict operational sequence as the For loop IFM. See Section 8.3 for details on the differences in the communication cycles between the IFMs and the given accelerator.

The interface definition also includes a 32 bit options port. This is included to give the parent system the ability to give options to the accelerator. The size is determined due to the ISA's word size of 32 bits. Note that it is possible for an accelerator designer to use one of the input data ports as an extra options port, but not with the For loop IFM. In addition to these control ports the definition includes an undefined number of input and output ports. These are 32 bits wide, again to conform with the ISA word size. The current IFM designs has limitations on the number of input and output ports, but the accelerator interface definition does not.

4.4 Accelerator ports

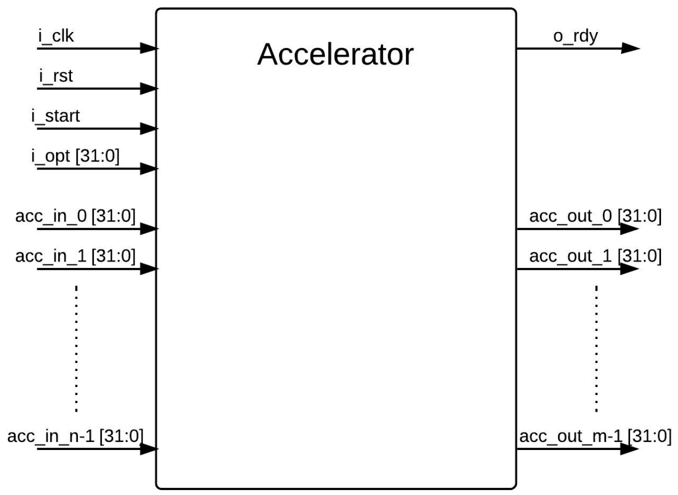


Figure 4.1: Accelerator module with N inputs ports and M output ports

Figure 4.1 shows the general accelerator module. The module is clocked and has a fairly minimal interface. The ports are described in the list below.

i_clk: This port is the system clock connected directly from the Amber core.

i_rst: This port is the system reset connected directly from the Amber core. The reset is active high and is treated synchronously in the Amber system.

i_start: This port is used to give the accelerator a start signal. The signal is a one cycle active high set by the IFM. In the Coprocessor(5.4) and the Slave(5.5) IFMs, this signal is set synchronously with the **i_opt** port. For the For loop(5.6) IFM, the **i_start** port will be set high once every cycle while the **i_opt** is held. This is described in detail in sections 8.3.1 and 8.3.2.

i_opt: This port is a 32 bit input array used to pass options to the accelerator. The options passed to the IFM will be held on this port constantly. As mentioned in the description of the **i_start** port, this port will be set synchronously with the **i_start** port.

o_rdy: This port determines when the interrupt or polling values in the IFMs is set, and when accelerators have new data on the **acc_out** ports. It is treated slightly different in the different IFMs, this is explained in detail in sections 8.3.1, 8.3.1 and 8.3.2, but the general idea is that the output buffers of the IFM are updated when this port goes high, in other words at the rising edge. The **acc_out** ports has to be set before the signal to avoid loss of data.

acc_in_0 to acc_in_n-1: These are the N 32 bit input data ports. There can theoretically be any number of these, but the accelerator design will need to comply with the limitations of the IFM chosen. Table 8.1 summarizes the IFM traits. They are not changed while **i_start** is held high, but can be changed before a calculation is finished, indicated by a high signal on **o_rdy**, in the basic IFMs. The different interactions for the different modules are described in detail in sections 8.3.1, 8.3.1 and 8.3.2.

acc_out_0 to acc_out_m-1: These are the M 32 bit output data ports. As with the input ports, M is only limited by the limitations of the IFM chosen. The data on these ports are clocked in to the output buffer of the IFM on a rising edge on the **o_rdy** port. There are variations regarding the timing of these operations in the different IFMs, see sections 8.3.1, 8.3.1 and 8.3.2 for details.

Chapter 5

Design of the Interface Modules

In this chapter I will elaborate on the design choices and the structure of the different IFMs. The first section gives an overview of a few possible solutions of IFMs for integrating an accelerator with the Amber Tile, and an explanation of the major design decisions made with the implementation of the IFMs. The following sections gives a detailed explanation of the architecture, functionality and verification of the IFMs.

5.1 Different options for IFM integration level

One of the major questions when designing an interface module is deciding the integration level of the accelerator. In this case integration level means on what level in the system architecture the accelerator is connected to the system. An accelerator implemented on it's own tile is at a higher level, architecturally, than an accelerator implemented with a coprocessor interface. To decide this we need to know a few things about the accelerator. An accelerator with a short execution time and low level of independence should be CPU near to avoid considerable communications overhead, while the opposite case of an independent accelerator with long execution time can be placed further away to allow the CPU to operate more independently. Below I have listed four alternatives, with some characteristic attributes, that stood out as options for the SHMAC system due to the architecture and topology of the SHMAC system and the Amber core[14].

- **Coprocessor Interface**
  - Amber CPU internal component, low amount of changes needed
  - direct ISA support
  - small area and time overhead
  - basic control logic
  - one word transfer only
  - high level of CPU control

  - suitable for small, fast accelerators with low data demands and a low degree of operational independence

- **Wishbone slave interface**
  - Amber tile component, low amount of changes needed
  - memory mapped CPU control
  - small area and time overhead
  - basic control logic, bus control needed
  - one word write, four word read
  - high level of CPU control
  - interrupt capability
  - suitable for accelerators with low data demands and a low degree of operational independence

- **Wishbone master interface**
  - Amber tile component, medium amount of changes needed
  - memory mapped CPU control
  - able to access off tile memory
  - medium area overhead, small time overhead
  - advanced control logic
  - low level of CPU control
  - interrupt capability
  - suitable for accelerators with high data demands and a high degree of operational independence

- **Tile interface**
  - Separate Tile component, high level of changes needed
  - memory and tile mapped CPU control
  - able to access off tile memory
  - large area and time overhead
  - advanced control logic
  - low level of CPU control
  - can be a shared resource between several CPUs
  - interrupt capability possible, not finalized in the SHMAC project
  - suitable for accelerators with high data demands, a high degree of operational independence and tolerance for high time delay

As can be seen from the list above, the interface type suited for a given application or accelerator is highly dependent on the properties of said accelerator and the demands of the application. However, the task in this thesis is to create an interface that is as general as possible, which can fit any type of accelerator and application. In order to do this I decided to create more than one interface type, and to make a system where the accelerator designer can choose between several options to find the best suited interface. This would also give an opportunity to test and characterize the different interface possibilities. The need to limit the amount of work required forced the decision to drop the Tile interface at an early stage. This is the most complex of the interface types, and would require large amounts of work. It was dropped to increase the probability of designing a functioning system with several interface options.

5.2 Start signal handling

The start signal of the IFMs are synchronised with writing the **options** register for all the IFM variations. In the Coprocessor and Slave IFMs writing the **options** register immediately triggers a high *start* signal with a duration of one clock cycle on the accelerators **start** port. In the For loop IFM writing the **options** register progresses the FSM and starts the execution of the for loop.

The decision to handle start signalling between CPU and accelerator in this way was made after careful consideration between this method and a method where the start signal was handled separately. The separate start method would have meant that the **options** register and **start** register would have been written separately.

With the separate start method, the programmer would be able to set and change the **options** register without giving the accelerator a *start* signal. For the Coprocessor and Slave IFMs this would have made the interface between CPU and accelerator even more general. In addition, a designer could use the **options** register as an extra input for accelerators that does not require options, and by doing so save area in the design. For the For loop IFM this advantage is not there due to the strict timing of the For loop start procedure. One could however argue that this approach would have made the For loop IFM safer and easier to use with a one transfer one function policy.

However, the synchronised start method alsohas its advantages. For all three IFMs the synchronised start method saves one interaction per start procedure. This saves time and energy, which are vitally important resources for any SoC or processing system.

As already stated, the decision was made to go with the synchronised start method. I consider the saved time and energy, achieved by limiting the necessary CPU to accelerator interactions as much as possible, to far outweigh the advantages in generalisation, easy programming model and area requirements of the separate start method. In addition to this one could argue that with some clever design an accelerator designer could achieve the separate start functionality for the Coprocessor and Slave IFMs. By using an extra input port as an options port in the accelerator design, you could achieve this functionality at the cost of some area. The area cost would be the logic and wires connected to the now unused options port.

5.3 Completion notification

The IFMs handle Completion handling differently from each other. The Slave and For loop IFMs both use an **interrupt** port connected to the tile's Interrupt Controller, while the Coprocessor IFM use a **poll** register. The Slave IFM sends a one cycle high *interrupt* signal as a direct response to the rising edge of the *ready* signal from the accelerator, the For loop IFM sends a one cycle high *interrupt* signal on the completion of the loop execution. The Interrupt Controller receives these signals and gives a one cycle high *interrupt* signal to the Amber Core. The Coprocessor IFM uses a one bit register to indicate completion. This register is written low when the *start* signal is given to the accelerator, and is written high upon the first following *ready* signal from the accelerator. The *poll* register is readable for the controlling CPU.

These methods for handling the *ready* signalling and secure completion of calculations were decided upon after evaluating several options. One option is stalling the CPU pipeline. While this method can offer some benefits for accelerators with a short execution time and applications that does not benefit from parallelization by saving energy usage by hindering unnecessary switching activity, this method does not keep in line with the idea of generalisation. This method would severely limit the different accelerators able to use the IFM. In addition to the generalisation issues, this method is only really viable for the Coprocessor IFM due to the Amber Tile architecture, and this would require substantial changes in the *Decode* and *Execute* modules of the Amber Core. These changes would create difficulties for any future work on optimizing the Amber Core.

The interrupt method has great merits when it comes to generalisation. It creates a ”fire and forget” use case where the CPU can work on different tasks while the accelerator executes. In addition it allows the accelerator designer to utilize several consecutive *ready* signals, where all signals will send an *interrupt* signal to the CPU. This method requires some changes to logic and Tile architecture, but uses existing functionality in both the Amber Core and the Amber Tile.

The poll methods main attribute is the small amount of changes necessary to the Amber Tile structure. It only requires the logic necessary to read a one bit register through the IFM interface in order to function. An implementation of an IFM with the poll methods keeps the architecture changes to a minimum. This method does however limit the generalisation characteristics of the IFM, as it requires a *start* signal to set the **poll** register low, making the system only able to react to one *ready* signal per *start* signal. This method also limits the possibilities of utilizing parallelization in the accelerated tile. The CPU is required to check if the accelerator is done, often with a continuous loop performing read operations. This costs time and energy the CPU could use for other tasks.

Evaluating these different methods, I decided that in order to approach the goal of an

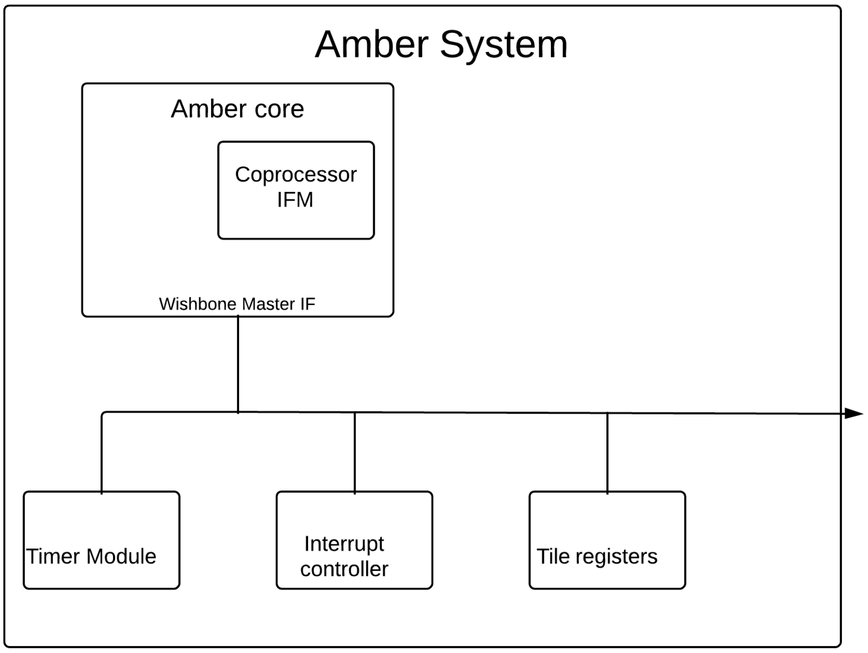


Figure 5.1: Amber System with the Coprocessor IFM

as general as possible interface system, the best option was the interrupt method. This was then implemented in the Slave and For loop IFMs. However, when looking at the architectural changes of the Amber Core required to allow the Coprocessor IFM to send a *interrupt* signal to the Interrupt Controller, I decided that the poll method was better suited to the Coprocessor IFM. While the goal of a high level of generalization is important, the necessity of keeping the changes to the Amber Core as small as possible, in order to not hamper any further development of the CPU architecture, took precedence here.

5.4 Coprocessor Interface Module

The Coprocessor Interface Module is an accelerator interface designed to be as core near as possible and require the smallest possible changes to the Amber Tile structure. The ARM ISA[13] includes several instructions for coprocessor data passing and control, but only a subset of these are implemented in the Amber Core. The Amber Core implements a simple coprocessor designed to handle cache control and the basic register to coprocessor one word data transfer instructions(*mrc* and *mcr*). Figure 5.2 illustrates how the Coprocessor IFM expands the coprocessor module without making any changes to the interactions with the Amber Core. The Coprocessor IFM offers a small and core near accelerator interface

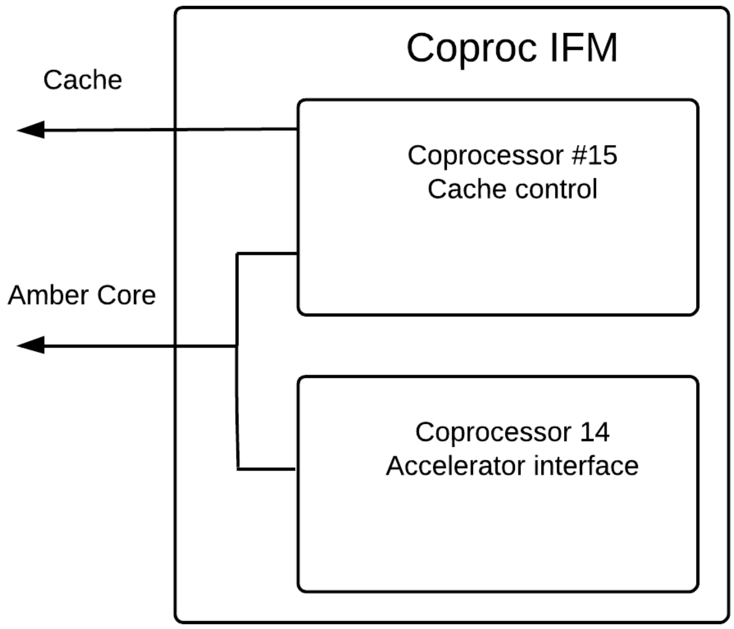


Figure 5.2: The coprocessor module replacing the standard module in the Amber Core

while limiting the changes to the Amber Core and Amber Tile architecture.

5.4.1 Implementation

Figure 5.3 shows the top level schematic for the Coprocessor IFM. As mentioned in section 5.4, the Amber Core only implements a small subset of the ARM ISA's coprocessor instructions. As a consequence of this several of the interconnects to the Amber Core serves no function. This includes the following ports: **i_copro_opcode1**, **i_copro_opcode2** and **i_copro_crm**. In addition, the ports named **o_cache_enable**, **o_cache_flush**, **o_cacheable_area**, **i_fault**, **i_fault_status** and **i_fault address** are only used by the existing coprocessor unit, and are not explained further in this thesis. See [14] for details. The rest of the ports are described below.

i_clk, i_rst, i_core_stall Standard core wide control signals, synchronous to the entire Amber Core. **i_core_stall** stops all operation

i_copro_crn Four bit coprocessor register number. Addresses the register written or read by the Amber Core

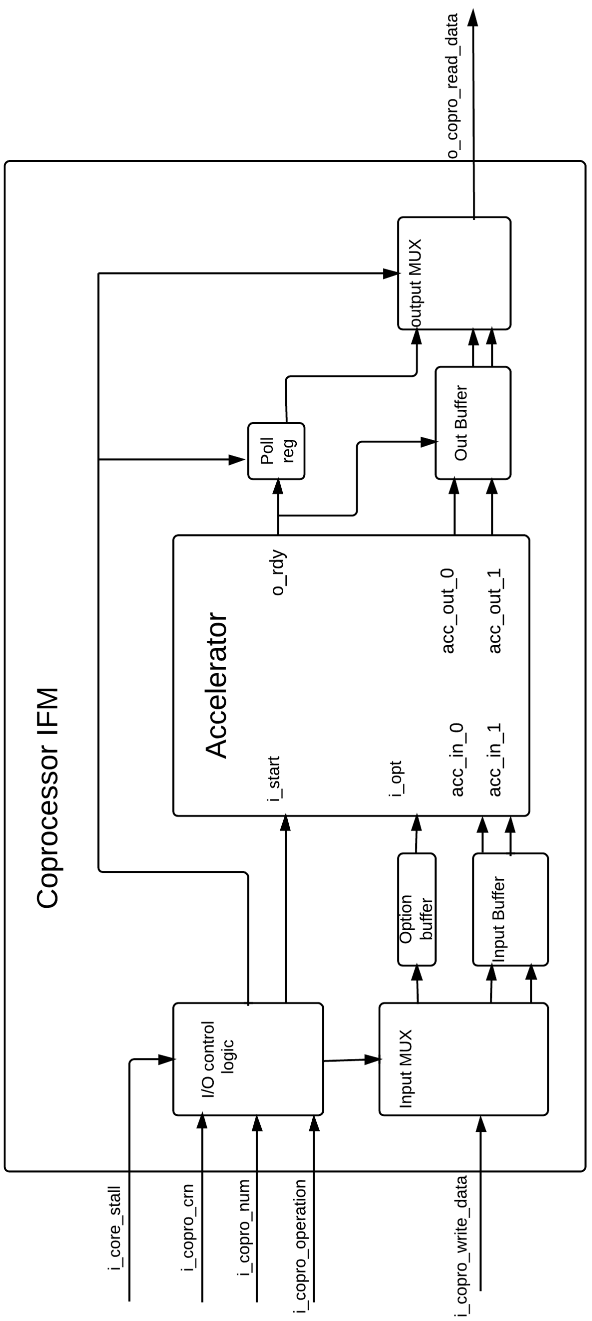


Figure 5.3: Schematic of the coprocessor IFM interfacing a 2 input 2 output accelerator. Note that signals and components not relevant to the IFM is omitted.

i_copro_num Four bit coprocessor number. Indicates the coprocessor called by the instruction

i_copro_operation Two bit operation number. 2'd2 indicates a write(*mcr*) instruction, 2'd1 indicates a read(*mrc*) instruction, 2'd0 indicates no operation

i_copro_write_data, o_copro_read_data 32 bit data transfer buses. Transmits one word of data to or from the coprocessor, respectively

In addition to the interface towards the Amber Core the Coprocessor IFM incorporates an interface for the accelerator. For a detailed definition of the general accelerator interface used in the IFM, see Section 4.3. The Coprocessor IFM uses several registers as interface buffers for the accelerator, listed below:

acc_start: 1 bit register used to give the accelerator a one cycle high *start* signal

poll: 1 bit register used to indicate a *ready* signal from the accelerator following a *start* signal. Coprocessor register number 15, read only

acc_opt: One 32 bit options register, connected to the accelerator's options port. Coprocessor register number 15, write only

acc_in: An array of **N** 32 bit registers, where **N** is the number of input ports to the accelerator. Coprocessor register numbers 0 to 14, write only

acc_out: An array of **M** 32 bit registers, where **M** is the number of input ports to the accelerator. Coprocessor register numbers 0 to 14, read only

The next two subsections explains the implementation of the Coprocessor IFM, sorted by actions initiated by the Amber Core and by the interfaced accelerator.

5.4.1.1 Amber Core initiated actions

The Amber Core can in essence only do two things, write or read a register. The Coprocessor IFM expands on this by allowing the writing of one of these registers to set the **acc_opt** register, give the accelerator a *start* signal and setting the Coprocessor IFMs **poll** register low. Note that a high signal on the **i_core_stall** port stops the Coprocessor IFM from responding to any input from the Amber Core. The only exception to this is a *reset* signal, which takes precedence.

A write operation is initiated by the Amber Core by setting the appropriate values to the **i_copro_num** and **i_copro_crn** ports and setting the value 2'd2 on the **i_copro_operation** port. On the next rising edge of the clock the value on the **i_copro_data** is stored in the corresponding register. If the register number on **i_copro_crn** denotes an accelerator input port, the corresponding register in **acc_in** is written. If the register number denotes the options register the **acc_opt** register is written with the data value of **i_copro_data**. In addition the **acc_start** register is set high for one cycle and the **poll** register is set low.

A read operation is initiated by the Amber Core by setting the appropriate values to the

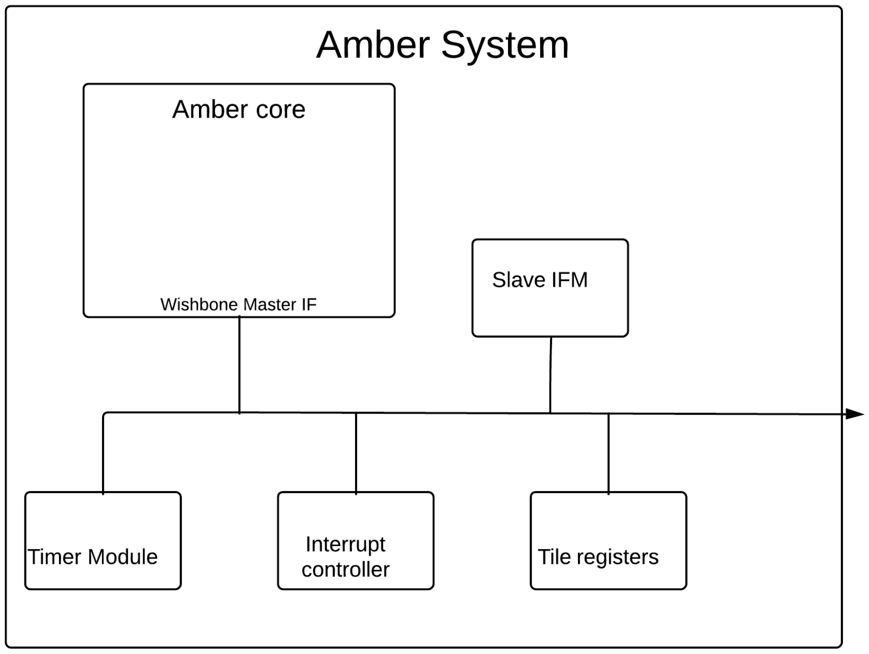


Figure 5.4: Amber System with the Slave IFM

i_copro_num and **i_copro_crn** ports and setting the value 2'd1 on the **i_copro_operation** port. On the next rising clock the value of the corresponding register will be set on the **o_copro_data** port. This includes the **poll** and the **acc_out** registers.

5.4.1.2 Accelerator initiated actions

The accelerator can only initiate one action. By setting a one cycle high *ready* signal on the **o_rdy** port, it initiates a store output action. On the consecutive rising edge of the clock the value of the accelerator's **o_acc_out** ports is stored in the **acc_out** registers. In addition, the **poll** register is set high.

5.5 Slave Interface Module

The Slave IFM is an accelerator interface designed to be as simple as possible, providing the same passive control regime as the Coprocessor IFM. The Slave IFM is memory mapped and connected to the Amber Core's Wishbone bus with a slave Wishbone bus controller. Figure 5.4 shows the Slave IFM's in the context of the Amber Tile architecture. The IFM is able to handle both four word writes and reads over the 128 bit Wishbone bus. Note that at the time of writing the Amber Core is only capable of doing four word Wishbone reads, not writes.

5.5.1 Implementation

Figure 5.5 is the schematic of the Slave IFM. The ports are described in the list below.

o_irq Interrupt signal connected to the Amber Tile Interrupt Controller.

Wishbone Slave Interface The remaining ports in the schematic is the Wishbone Interface. See See Section 2.2 for details

clk and rst(omitted from figure) Standard system wide control signals, synchronous to the entire Amber Tile.

In addition to the interface towards the Amber Tile Wishbone bus the Slave IFM incorporates an interface for the accelerator. For a detailed definition of the general accelerator interface used in the IFM, see section 4.3. The Slave IFM uses several registers as interface buffers for the accelerator, listed below:

acc_start: One bit register used to give the accelerator a one cycle high *start* signal

o_irq: One bit register used to control the *interrupt* signal to the Amber System.

acc_opt: One 32 bit options register, connected to the accelerator's options port. Tile memory address 0x37FC, write only

acc_in: An array of **N** 32 bit registers, where **N** is the number of input ports to the accelerator. Tile memory addresses 0x3000 to 0x37F8, write only

acc_out: An array of **M** 32 bit registers, where **M** is the number of input ports to the accelerator. Tile memory addresses 0x3800 to 0x3FFC, read only

The next two subsections explains the implementation of the Slave IFM, sorted by actions initiated by the Amber Core and by the interfaced accelerator.

5.5.1.1 Amber Core initiated actions

The Amber Core can in essence only do two things, write or read a register. The Slave IFM expands on this by allowing the writing of one of these registers to set the **acc_opt** register and give the accelerator a *start* signal. The Slave IFM is memory mapped, so both actions are completed through memory operations through the Amber Tile's Wishbone bus.

5.5.1.2 Accelerator initiated actions

The accelerator can only initiate one action. By setting a one cycle high *ready* signal on the **o_rdy** port it initiates a store output action. On the consecutive rising edge of the clock the value of the accelerator's **o_acc_out** ports is stored in the **acc_out** registers. In addition, the **o_irq** register is set high for one cycle.

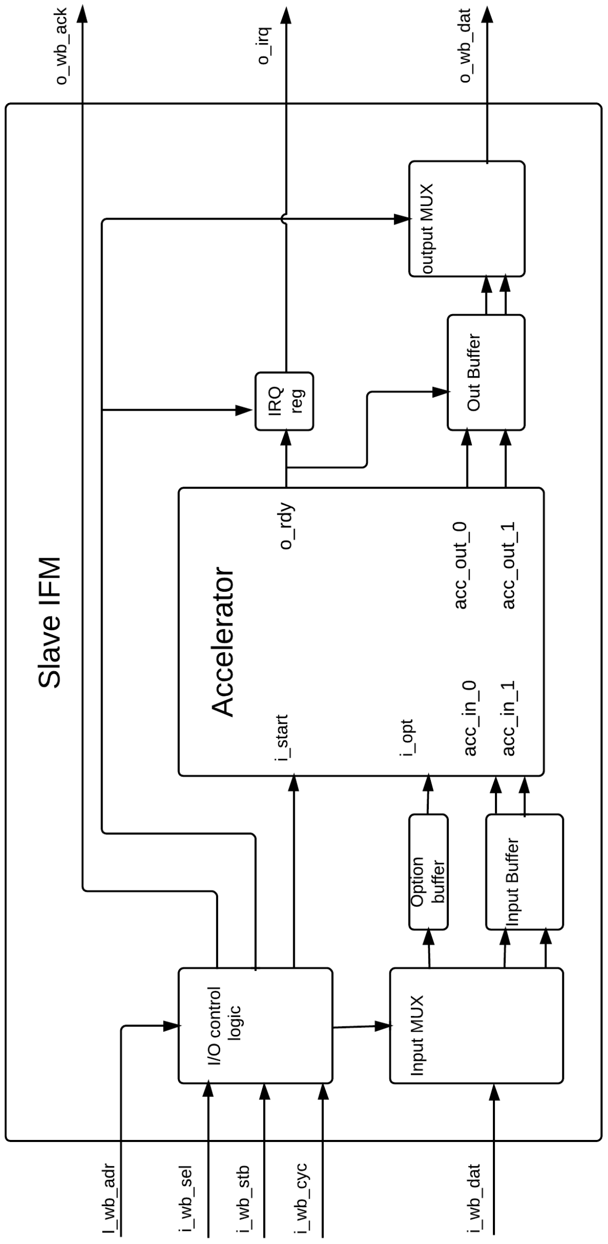


Figure 5.5: Schematic of the slave IFM interfacing a 2 input 2 output accelerator. Note that clock and reset signals are omitted.

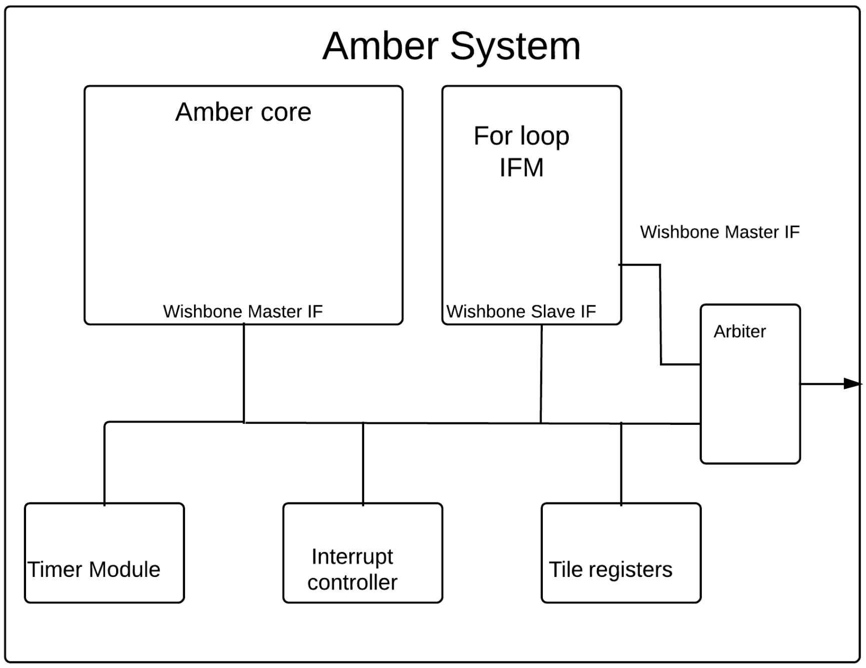


Figure 5.6: Amber System with the Master IFM

5.6 For loop Interface Module

The For loop IFM is a very specific IFM meant to perform the independent execution of a for loop containing a given accelerator. This solution moves away from the passive and heavily generalised interfaces provided by the Coprocessor and Slave IFMs, and takes on the task of providing a module capable of executing a for loop with minimal input required from the parent CPU. Due to the specific nature of a for loop, the For loop IFM requires specific interaction procedure of accelerators. This sets specific requirements to accelerator timing, described in Section 8.3.2, and limits the type of accelerators to those that can function as the action of a for loop. The For loop IFM implements both a Master and a Slave Wishbone interface, and is capable of direct interaction with the off tile memory. The Slave Wishbone controller allows control of the IFM by the Amber Core. Figure 5.6 shows the For loop IFM implemented in the Amber System module. As is apparent from Figure 5.6, the implementation of the For loop IFM in the Amber System module requires a Wishbone Master Arbiter to handle off tile communication.

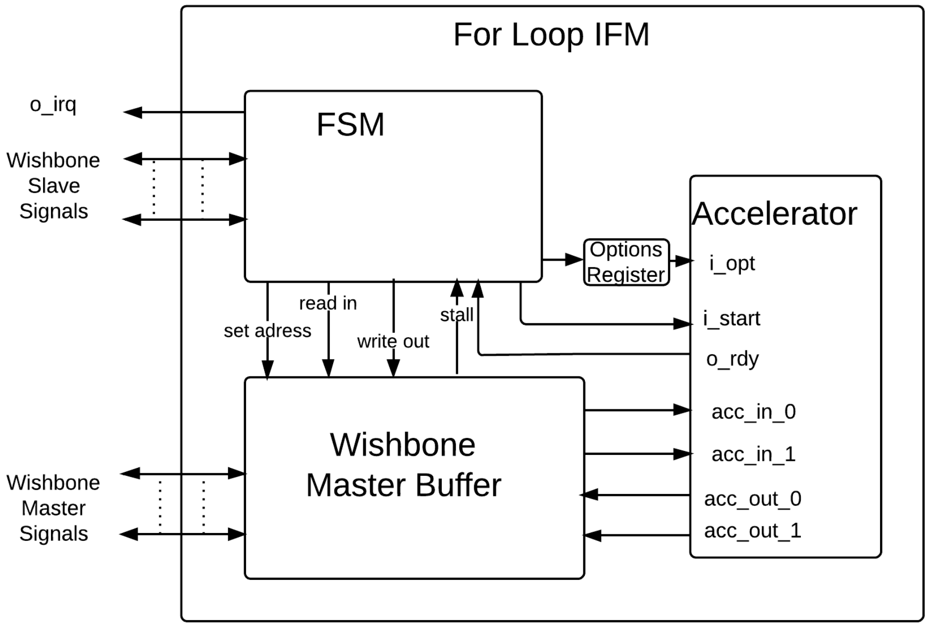


Figure 5.7: Schematic of the For Loop IFM interfacing a two input two output accelerator. Note that clock and reset signals are omitted.

5.6.1 Implementation

The For loop IFM consist of three major parts as seen in Figure 5.6: an accelerator interface, a Wishbone Master Buffer(WMB) and a FSM. The WMB reads and writes the memory and sorts the data to and from the accelerator inputs and outputs. The FSM controls the entire loop execution. It takes the input arguments from the CPU over the Wishbone Slave interface, controls the WMB and the accelerator, and sends an *interrupt* signal when the loop execution is done.

5.6.1.1 Interface

Figure 5.7 is the top level schematic of the For loop IFM. The ports are described in the list below.

i_clk and i_rst Standard system wide control signals, synchronous to the entire Amber Tile.

o_irq Interrupt signal connected to the Amber Tile Interrupt Controller.

Wishbone Slave Interface The ports marked **_wbs_**in the schematic is the Wishbone Slave Interface, used to pass control signals from the Amber Core. See Section 2.2 for details on the Wishbone interface

Wishbone Master Interface The ports marked **_wbm_**in the schematic are the Wishbone Master Interface, used to fetch input data and write output data to the off tile memory.

5.6.1.2 Finite State Machine

The state chart in Figure 5.8 gives the functionality of the For loop IFM's FSM. The list below gives a description of each state.

Idle This is the Idle state. The FSM moves to the next state with any activity on the Wishbone Slave interface

Recieve The For loop IFM recieves arguments over the Wishbone Slave interface in this state. When the **Options** argument is received, the FSM moves to the next state.

Adress Set The **input address** and **output address** arguments received in the previous state is written to the WBM

Fetch Input Data The WBM is signalled to fetch input data and place it on the accelerator interface. The FSM moves to the next state when the WBM's **stall** port is low

Calculation The accelerator is given a *start* signal. The FSM moves to the next state when a *ready* signal is recieved

Store Output Data The WBM is signalled to get store the output data from the accelerator interface and write it to memory. The next state is determined by subtracting 1 from the **Main_i** argument and checking for zero. If this was the last iteration, the next state is **Interrupt Set**. If not, the next state is **Fetch Input Data** The FSM moves to the next state when the WBM's **stall** port is low.

Interrupt Set Gives the Amber Tile's Interrupt Controller an *interrupt* signal. FSM moves to the next state immediately.

5.6.1.3 Wishbone Master Buffer

The Wishbone Master Buffer(WMB) is one of the main components of the For loop IFM, its function is to handle the necessary memory operations required to execute the for loop. A schematic of the WMB is shown in Figure 5.9. The WMB reads and writes the off tile memory and handles all data passing to and from the accelerator, controlled by the FSM. It is able to handle any number of I/O ports of a given accelerator and at the same time utilize the full capacity of the 128 bit Wishbone bus by arranging the data in four word transfers as far as is possible. The last transfers might not add up to four data words, but are of course executed to complete the calculation. This is done with the use of several buffers and First In First Out(FIFO) queues, and an adaptation by the Tile Generator script.

At the beginning of a for loop execution, the FSM passes the input and output address arguments to the WMB. During the execution, they are kept and updated in the WMB internally. The *stall* signal is used to halt the FSM execution when the WMB is busy and unable to comply with any control requests.

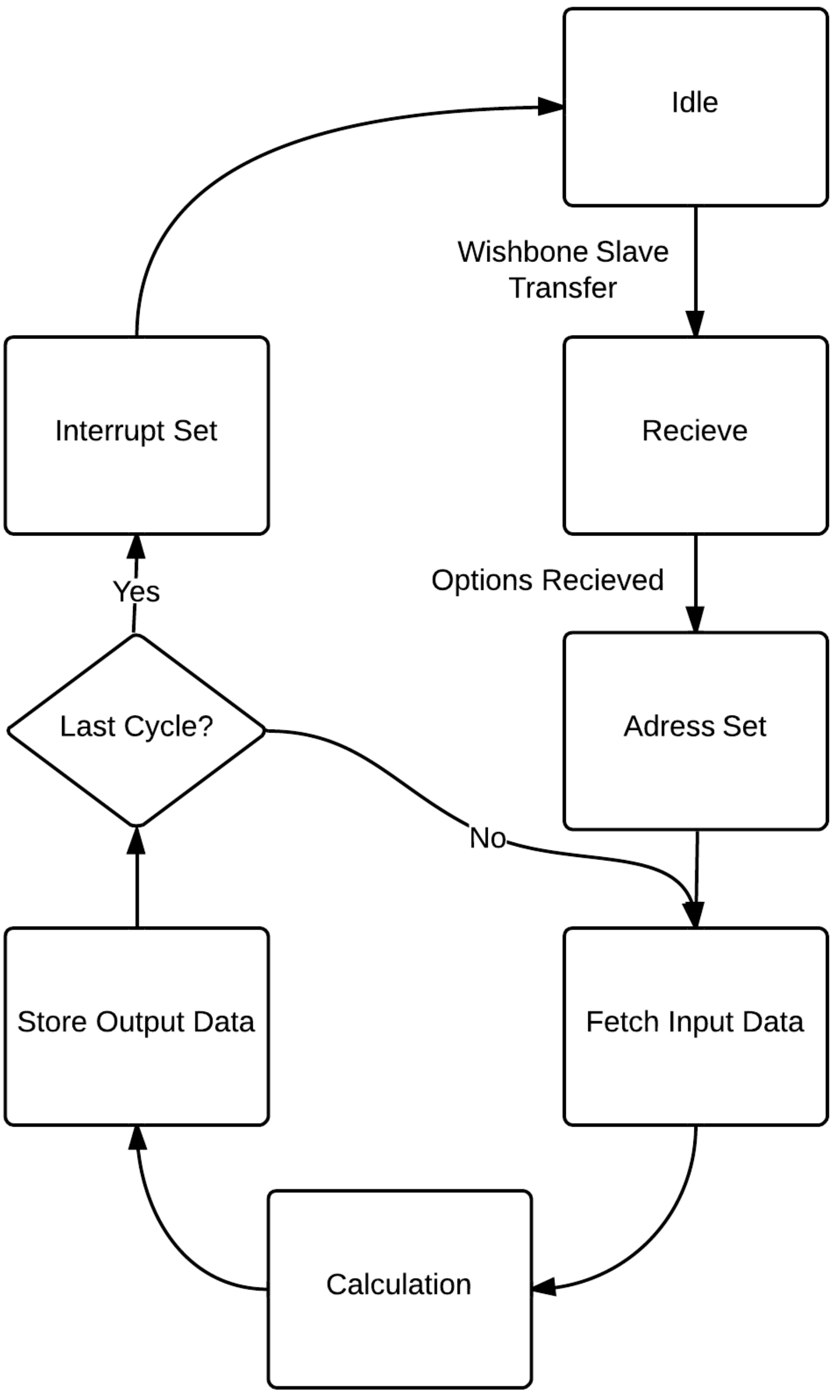


Figure 5.8: State chart for the For loop IFM's FSM control unit

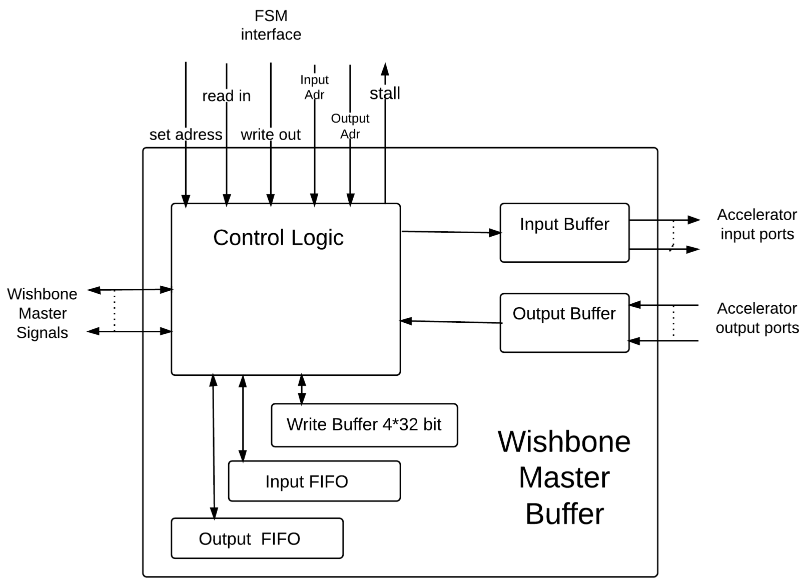


Figure 5.9: Schematic of the Wishbone Master Buffer

The input data to the accelerator is read sequentially in four word transfers from memory by incrementing the given input address. Afterwards it is placed directly into the **input FIFO** queue. When the *read in* signal is given by the FSM the correct number of data words are sent to the **Input Buffer** filling it with one word per input port on the accelerator.

The output data follows a similar path, only in the other direction. When the *write out* signal is given by the FSM, the **Output Buffer** is written, and read in to the **Output FIFO**. If the Output FIFO contains more than four words, a Wishbone write operation is initiated.

This module is the one most modified in the For loop IFM during script generation. The size of both the FIFO queues and I/O buffers are generated to adapt the WMB to the given accelerator and amount of data.

5.6.1.4 Wishbone Master Arbiter

The Wishbone Master Arbiter(WMA) is a simple two input one output round robin bus arbiter, placed in the Amber System level of a For loop IFM Tile. It reacts to the *cyc* signals from both Masters and assigns priority based on first come first serve. If the unprioritised Master sets its *cyc* signal high while the prioritised Master is transferring, it will be assigned priority as soon as the prioritised master is done. The fact that this arbiter reacts to the *cyc* signal allows for a multi transfer Wishbone cycle to pass unhindered. Figure 5.10

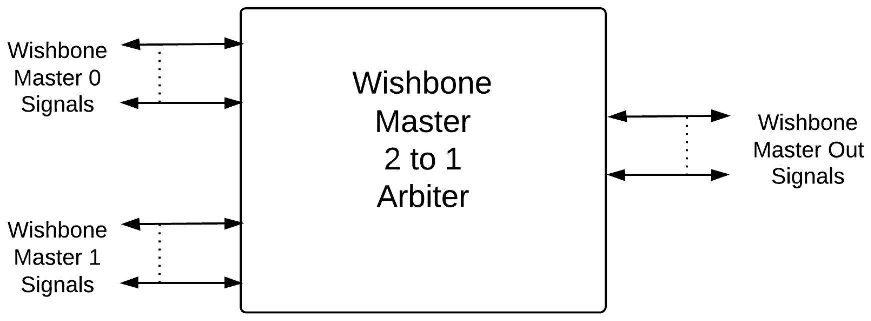

Wishbone Master 0 Signals
Wishbone Master 1 Signals
Wishbone Master 2 to 1 Arbiter
Wishbone Master Out Signals

Figure 5.10: Top level Wishbone Master Arbiter

shows the top level of the WMA.

Chapter 6

IFM and tile scripted generation

In this chapter I will elaborate on the functionality and set up of the Tile Generator, a IFM and tile generation script, as well as elaborate on the reasoning for using a high level scripting language to generate tile structures and verilog modules. Please note that a user guide for the tile generation script can be found in Section 8.1.

6.1 Overview of the Tile Generator

The Tile Generator is a Python script designed to generate a new tile based on the Amber Tile with a functioning IFM adapted to match a given accelerator. Figure 6.1 shows an overview of the script. The script takes some arguments from the user, loads up the necessary source and template files, and outputs a modified version of the Amber Tile with a new name and a functional IFM. The script operation is described in more detail in Sections 6.3.2 and 6.3.1.

The IFM system described in this thesis is meant to be as general as possible. In order to achieve this, a certain degree of flexibility, with regard to all the different accelerators that can be developed, is necessary. The general accelerator interface described in Section 4.3 offers this to a certain degree, but in order to cater to the differences in accelerator attributes, Section 5.1 describes the need for several IFM types. On top of this, the verilog HDL offers no easy way to generalise an unknown number of ports when writing a module interface. SHMAC is also a tile based architecture, so some way to easily differentiate between different tiles are necessary in order to keep the project easy to navigate and work with.

All of these issues complicate the IFM system and make it difficult for an accelerated system designer within the SHMAC project to create an accelerator tile with the correct IFM that has the correct module instantiation. In order to simplify these issues and keep the IFM system as general as possible, without the need for other designers and participants of the system to manipulate the IFM source files, I have created a script that generates a

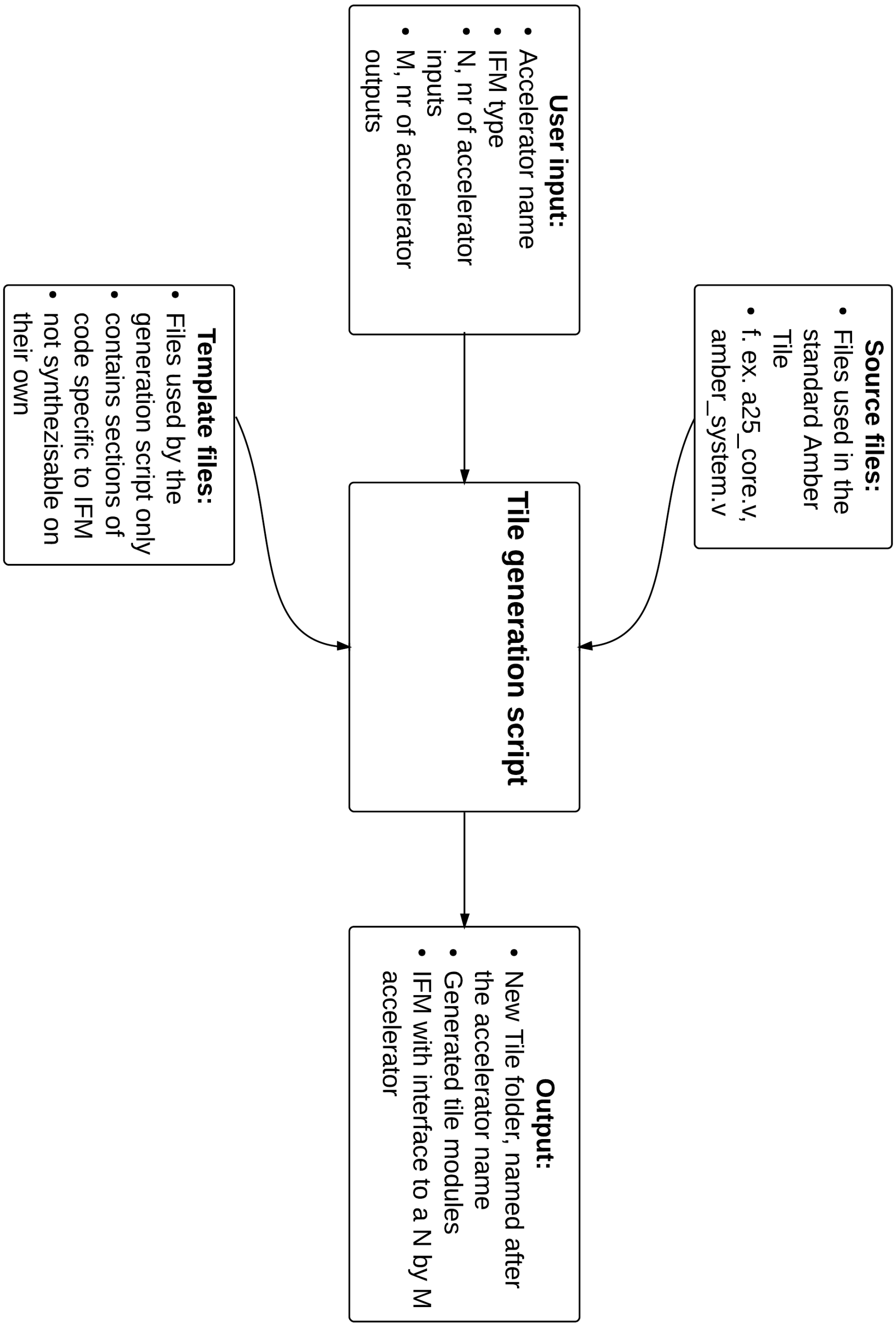

User input:
Accelerator name
IFM type
N, nr of accelerator inputs
M, nr of accelerator outputs
Template files:
Files used by the generation script only
contains sections of code specific to IFM
not synthezisable on their own
Source files:
Files used in the standard Amber Tile
f. ex. a25_core.v, amber_system.v
Tile generation script
Output:
New Tile folder, named after the accelerator name
Generated tile modules
IFM with interface to a N by M accelerator

Figure 6.1: Overview of the Tile Generator script

new tile type with a unique name, its own high level source files and an adapted IFM with the correct module instantiation for the given accelerator.

6.2 Use of source and template files

The Tile Generator script uses a mix of template files and source files to generate the new tile structures and IFMs. The source files are the files used in the Amber Tile and Amber core as part of the working SHMAC repository. These files are, as a consequence of the project's development and progress, subject to change. This, in turn, makes the Tile Generator script and the entire IFM system vulnerable to compatibility issues and conflicts with future changes. Figure 6.2 shows a visual representation of how the two files are used. Note that the *a25_coprocessor.v* file is one of the source files for the unmodified Amber Tile's Amber Core module.

One alternative to this approach that would solve the danger of conflicting changes to the source files is to use a template only solution. This would mean to only use a template file, and modify this through the script. This would guarantee a working generated tile, but the accelerated tiles would not benefit from any future changes. In effect, this would mean branching off the affected files at this point in the development. This could be mitigated through manually keeping the template files updated, but this would mean a substantial extra workload for any designer that would improve the Amber Tile in the future.

Evaluating these options against each other the current solution with the use of working source files were chosen. This means that any designer working on one of the affected files will need to take great care not to create any conflicts, but I consider this the best option. In order to mitigate the danger of designers unaware of the danger creating conflicts, all files that are used as source files have been given a warning in the top of the file, in the form of a comment. This warning lets the designer know that the file is used as a source file, in addition to a list of the template files and the script file that might cause a conflict. Listing 6.1 gives an example of one of these warnings.

Listing 6.1: excerpt from a25_coprocessor.v: *Danger of conflict* warning

```
//////////////////////
//////WARNING!!///////
//////////////////////
//
//This file is used as a source file for accelerator interface tile
    generation!
//It contains tags used for automatic navigation by the generator script.
//make sure any changes made to this file is compatible with the rest of
    the system
//Files that can cause compatibility issues are listed below
// /shmac/hardware/tileGenerator/tileGenerator.py
// /shmac/hardware/tileGenerator/templates/coproc_TEMP.v
// /shmac/hardware/units/amber/hw/vlog/amber25/a25_core.v
//
// Marton Teilgard mteilgard@gmail.com
//////////////////////
```

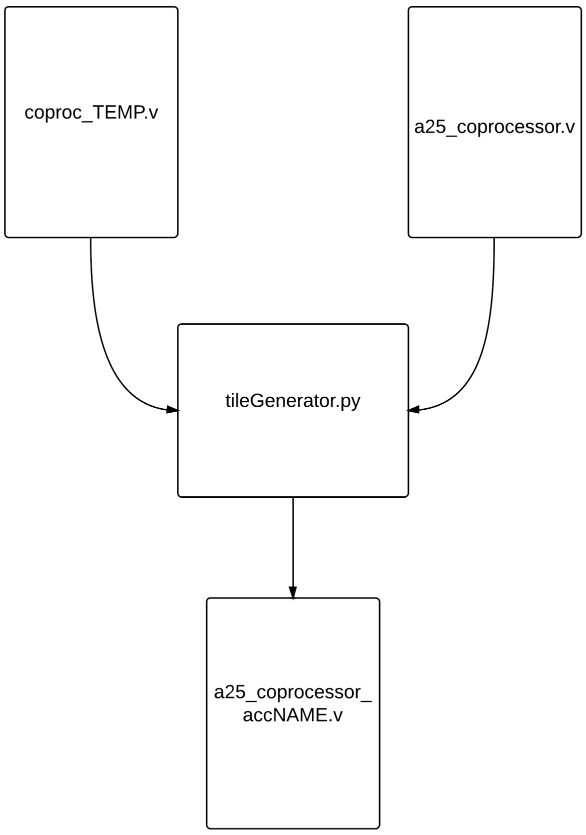


Figure 6.2: Script merging source file and template file in to the coprocessor IFM source file

6.3 Detailed Functionality

The script takes four arguments from the user, listed below. The script then generates the high level modules required to instantiate the new tile type, and the new IFM modules. Modified versions of some low level Amber modules is also created in some cases.

Accelerator Name Name of the accelerator module, used to name the new tile and its files, and to generate the accelerator instantiation in the IFM

IFM type Decides what type of IFM to generate

N and M Integers denoting the number of input ports and output ports on the accelerator, respectively

The high level source files are generated, as explained in the next subsections, by modifying existing source files by direct changes and sections from template files. The high level source files are updated with new module names and instantiations, with the exception of *Amber_system.v*, which is changed more extensively when modified for the For loop IFM. The IFM modules relies mostly on template files only, except from when generating the Coprocessor IFM, which creates a modified version of the *a25_coprocessor.v* source file from the Amber Core architecture. The IFM source files are generated by copying sections from template files and generating the structures necessary to fit the dimensions of the accelerator.

6.3.1 Generating the tile structure

In order to generate the tile structure a new folder in the */shmac/hardware/tiles/* directory has to be created. Furthermore the high level tile architecture source files have to be copied and modified from the source files of the Amber Tile module. The script first creates the new folder and names it *amber_tile_[IFMTYPE]_[ACCNAME]*, then it starts the copying and modifying of the high level tile modules from the Amber Tile, and creates the files *amber_tile_[ACCNAME].vhd*, *amber_wrapper_[ACCNAME].v* and *amber_system_[ACCNAME].v*. These files are created by the script by first opening a source file and a template file, and the output file. Then it sequentially runs through the source file line by line, copying it in to the output file, all the while checking each read line for a commented tag. These tags are unique, and are placed in the source files at the points where modifications and/or sections from a template file is required. An example of such a tag is shown in Listing 6.2. In this particular case, the script removes the tag, inserts the modified module name, skips through the source file to the **...DONE** tag, and finally resumes the copying of the source file in to the output file.

Listing 6.2: excerpt from amber_system.v: commented tag for the Tile Generator

```
`include "common_defs.v"
//ACCTAGSYSINST
module amber_system
//ACCTAGSYSINSTDONE
  #(
    parameter tile_x = 4'b0,
    parameter tile_y = 4'b0,
```

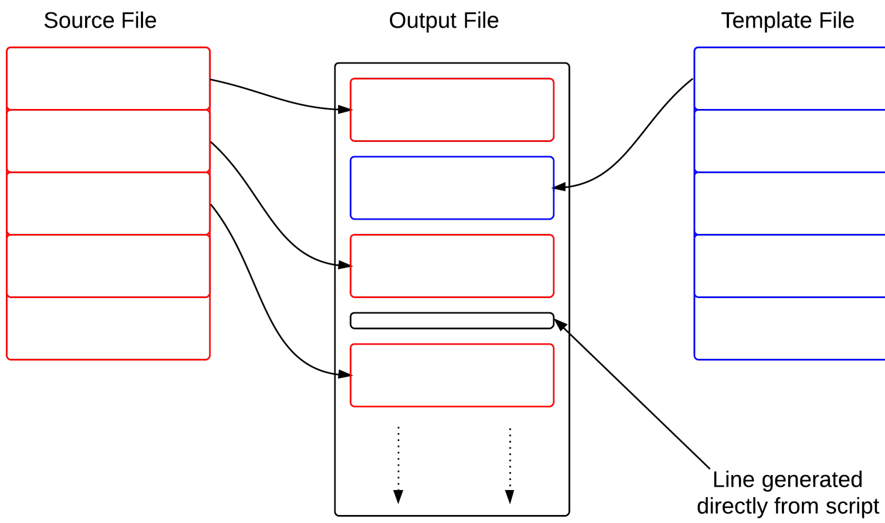


Figure 6.3: Detailed illustration of scripted file generation

```
    parameter cpu_id = 8'hff
    )
```

In other cases the scripts copies sections from the template files at such tags and in some cases the **...DONE** tag is omitted, as there is no need to skip lines in the source file. The template files are equipped with similar tags for the same reason. These are slightly different to make it easier to distinguish the different types of files. An example is given in Listing 6.3. Note that these tags also make the verilog template files uncompilable, adding a barrier for unintentional misuse of these files.

Listing 6.3: excerpt from slave_sys_TEMP.v: tag *#0* for the Tile Generator

```
assign int_sources = {26'b0,irq_acc , irq_timers , i_irq};

  #0
    .i_clk(i_clk),
    .i_rst(i_rst),
```

The script goes through a sequential program for each file, modifying them by alternating between lines from source files, lines from template files and lines written directly from the script. The script differentiates which files are created and how they are modified based on the IFM type selected. An illustration of how a modified tile source file is created is shown in Figure 6.3.

6.3.2 Generating the IFM

Large sections of the IFM files are generated with the same method as the tile structure files, by mixing parts from source files, template files and directly generated lines. However, in order to adapt the IFMs to the dimensions of the given accelerator, some parts of the modules are generated directly as a function of the number of input and output ports in the accelerator interface. This is where the user arguments **N** and **M** are used.

Listing 6.4: excerpt from tileGenerator.py: script generating the instantiation of an accelerator in the Slave IFM

```
  #2 found
  for i in range(inputsN):
    oFile.write("\t.acc_in_" + str(i) +" (acc_in["+ str(i) +"]),\n")
  for i in range(outputsM):
    oFile.write("\t.acc_out_" + str(i) +" (acc_out_" + str(i))
    if i == (outputsM - 1):
      oFile.write(")\n")
    else:
      oFile.write("),\n")
```

Listing 6.4 shows the part of the script where the instantiation of the accelerator is generated. The input and output ports are connected to the previously generated I/O registers. Listing 6.5 shows a generated instantiation of an accelerator named *acc_dummy_4_1* with four input ports and one output port.

Listing 6.5: excerpt from slave_acc_dummy_4_1.v: generated accelerator instantiation

```
  acc_dummy_4_1 accelerator(
  .i_clk (i_clk),
  .i_rst (i_rst),
  .i_start (acc_start),
  .i_opt (acc_opt),
  .o_rdy (acc_rdy),
  .acc_in_0 (acc_in[0]),
  .acc_in_1 (acc_in[1]),
  .acc_in_2 (acc_in[2]),
  .acc_in_3 (acc_in[3]),
  .acc_out_0 (acc_out_0)
);
```

Chapter 7

Verification and Overhead

This chapter will go in to how the IFMs and the Tile Generators functionality were verified, and what overhead that area associated with the different IFM solutions.

7.1 Verification method and tools

The Amber Tile structure is made up of several different components, some of which have been created for the SHMAC project, and some that have been modified from other work, like the Amber Core. The Amber Core is the most important component of the Amber Tile, as it is the CPU, and is therefore the most important component to test the IFMs with. This section will show how the different tools and methods were used to verify the functionality of the IFM system.

7.1.1 Amber Test Framework

The Amber Test Framework(ATF) detailed in 3.3 is able to run assembly files on the Amber Core. This system has been used to test the Coprocessor IFM's functionality, but the complexity of the framework, and the fact that it is designed mainly to test the Amber Core, made it impractical to use for testing of the Slave and For loop IFM. I was however able to document the Amber Core's Wishbone interactions, which made me able to emulate these in the Amber dummy module, detailed in Section 7.1.2.

7.1.2 Amber Core dummy module

The **a25_core_dummy**, or Amber dummy module, is a simple verilog module with the Amber Core's interface created to emulate the Amber Core's Wishbone interactions. The timing which this module is created to emulate was recorded with the use of the ATF. These interactions are shown in Figures 7.1 and 7.2.

The Amber dummy contains of a simple FSM. It is made up of two sequential state strings, one for testing the Master IFM and one for the Slave IFM, selected by the verilog

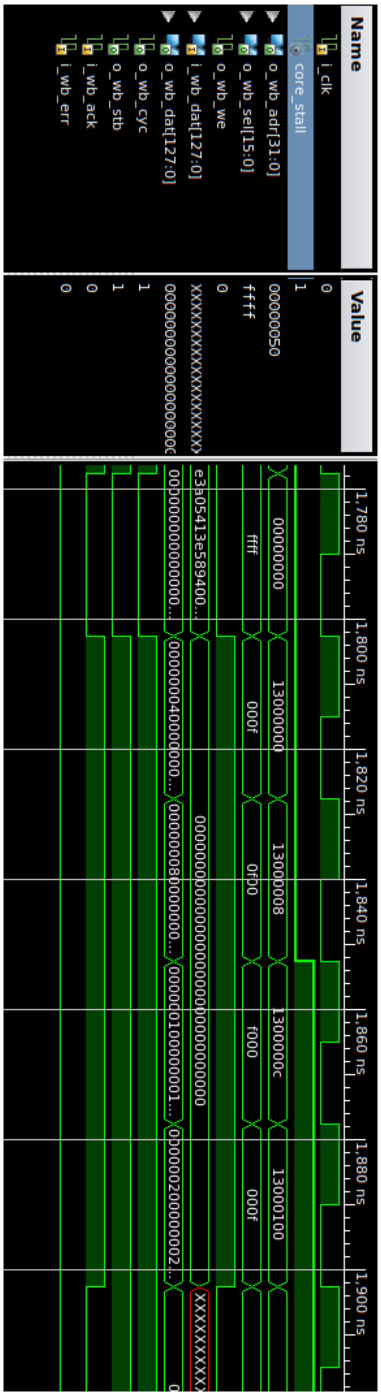


Figure 7.1: Waveform showing four consecutive Wishbone writes made by the Amber Core

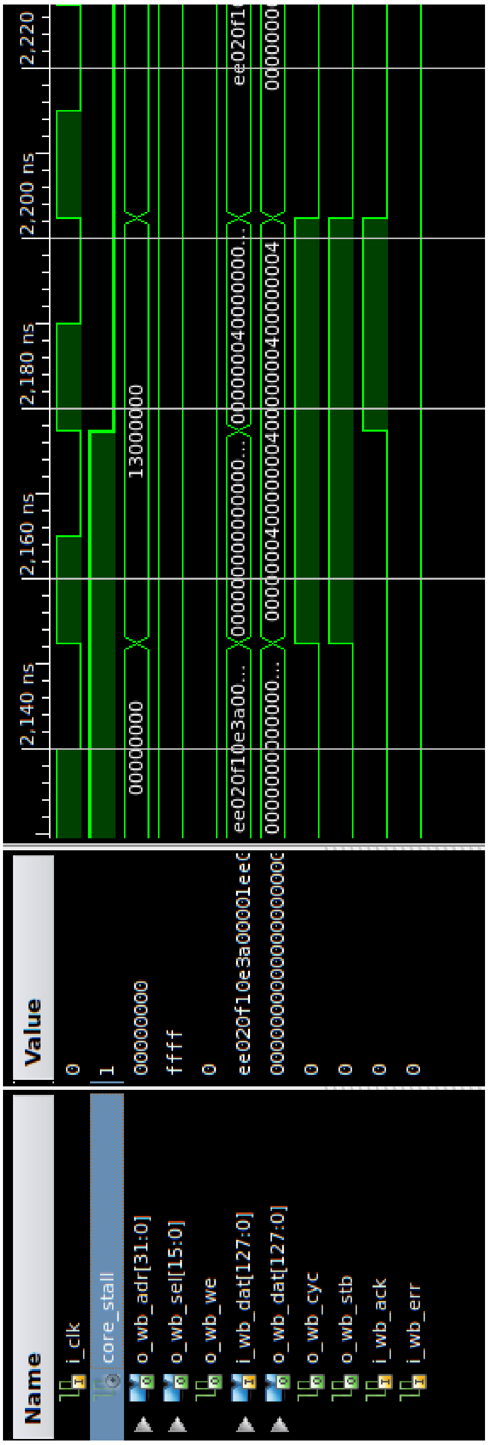


Figure 7.2: Waveform showing one Wishbone read made by the Amber Core

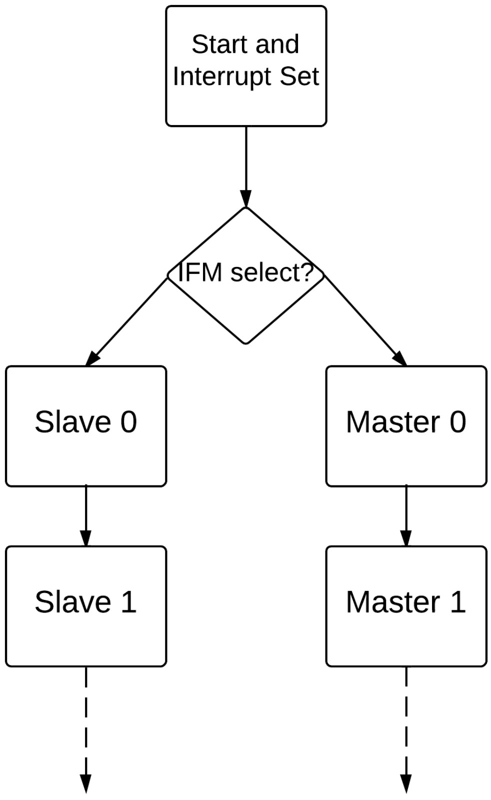


Figure 7.3: Amber dummy state chart

test bench via the first Wishbone read. A simple version of the state chart is shown in Figure 7.3 This is made in such a way that it is easy to manipulate by adding or modifying the states, easily creating new Wishbone interactions and test regimes. By using this module as an Amber Core, it is possible to easily test an entire generated Amber Tile with an IFM. For all intents and purposes, this FSM was the main test bench for the final verification of the Tile Generator and the Slave and For loop IFMs. The Amber Core dummy module can be found in Appendix .1.

7.1.3 Accelerator dummy modules

In order to test an accelerator interface method, you need an accelerator, preferably several. In order to be able to test as simply as possible, I opted to create several very basic accelerator like modules with different versions of the accelerator interface defined in Chapter 4. These are very basic modules that reacts to inputs and options and create easily verifiable output data. These modules make it very easy to verify that the IFMs are working, and have been used to test all the IFMs throughout development. The Accelerator dummy source file can be found in Appendix .1.

7.1.4 Verilog Testbenches

Verilog testbenches are used throughout the development of the IFM system, to test separate components and debug specific issues and to test the high level accelerated Amber Tiles. They are used with the Xilinx ISE and ISim plugin described in Section 3.3 to simulate, test and debug various verilog modules. One specific testbench needs to be mentioned, the **genSysTB**. It is used to test the generated Amber system modules, and is the highest level testbench used in verification. It emulates an off tile memory, and together with the Amber Core dummy module in Section 7.1.2 makes it possible to easily test both the Slave and For loop IFMs and the tile structures. The **genSysTB** can be found in Appendix .1.

7.1.5 Xilinx ISE synthesis

In addition to the above methods, the Tile Generator has been extensively tested by synthesising the entire generated tile structures. Tile types with several variations of each IFM have been synthesised to verify that the script generates modules that can be programmed to the Xilinx FPGA.

7.2 Coprocessor IFM verification

The Coprocessor IFM has been verified directly in the Amber Test Framework. Several Coprocessor IFMs have been generated with an Accelerator dummy module and implemented in the existing Amber Core structure of the SHMAC project. The Amber Test Framework was the initialized with an assembly program designed to verify the IFM. The assembly program passes data and options to the IFM, reads the **poll** register until it reads as high, reads the output data and verifies that it is correct. An example of the assembly programs can be found in Appendix .1. In addition the waveform files are submitted with the Thesis.

7.3 Slave IFM verification

The Slave IFMs functionality is verified with the use of the Amber Core dummy module in Section 7.1.2 and the **genSysTB** in Section 7.1.4. Several generated tiles have been tested, the one used as an example in this section implements a four input four output dummy accelerator. The testbench is simulated in ISim and the functionality is verified visually, by checking that the data moves correctly and that the control signals act accordingly. One example is shown in Figure 7.4, where a write of the **options** register initiated by the Amber Core dummy results in that both the options argument and the *start* signal is passed to the accelerator. The Amber Core test for this IFM system writes input data and options

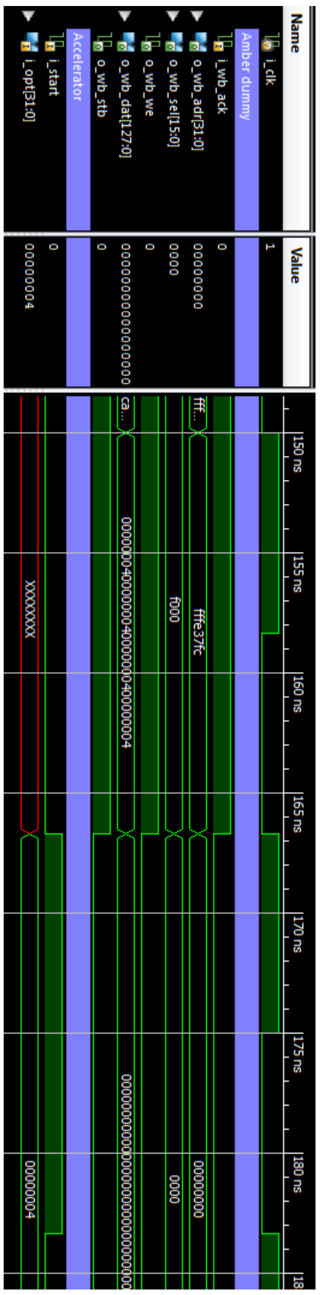


Figure 7.4: Waveform showing the *write options* action through the Slave IFM

to the IFM, waits for an *interrupt* signal, and reads the output data in both one and four word transmissions. Figure 7.5 shows how a *ready* signal from the accelerator translates to an *interrupt* signal to the Amber Core dummy, and a four word Wishbone read of the output data from the accelerator. Figure 7.6 shows how a single word read is translated to the correct placement on the Wishbone data bus by checking the *select* signal.

The waveform files are submitted with the Thesis.

7.4 For loop IFM verification

The For loop IFMs functionality is verified with the use of the Amber Core dummy module in Section 7.1.2 and the **genSysTB** in Section 7.1.4. Several generated tiles have been tested, this section includes examples from the verification of a system implementing a one input one output accelerator dummy, and a system implementing a six input six output accelerator dummy. The tests are controlled by an Amber Core dummy that writes the necessary arguments one word at a time, waits for the *interrupt* signal, and then writes all four arguments in one transfer. The **genSysTB** emulates a memory with two address spaces. The For loop IFM reads one of the address spaces and writes the other one. The tests are simulated in ISim and the functionality is verified visually, by checking that the data moves correctly and that the control signals act accordingly.

Figure 7.7 shows the For loop IFM writing four words to off tile memory, the data is output from a one output accelerator organised in four word transfers. Figure 7.8 shows the Wishbone Master Buffer(WMB) transferring six data words from the internal input FIFO to the accelerator's six input ports. The waveform files are submitted with the Thesis.

7.5 Time Overhead

This section details the overhead time associated with each IFM, excepting the execution time of the accelerator and any time used by the Amber Core to fetch data and instructions relevant to the IFM control. Note that this section does not take in to account parallelism, see Section 9.1.4 for details.

7.5.1 Coprocessor IFM

The Coprocessor IFM is an internal component of the Amber Core, and is controlled directly by the Amber Core pipeline. The Coprocessor IFM is a passive IFM and requires that the Amber Core passes all input and output data, in addition to the options argument. As is shown in Figure 7.9, every transfer from the Amber Core to the Coprocessor IFM takes two cycles. The Amber Core passes one 32 bit word per transfer, both in read and write. In addition to this, the Coprocessor IFM takes one cycle to update the **Poll** register after a *ready* signal from the accelerator, shown in Figure 7.10.

Equation 7.1 shows the overhead time of one calculation with the Coprocessor IFM

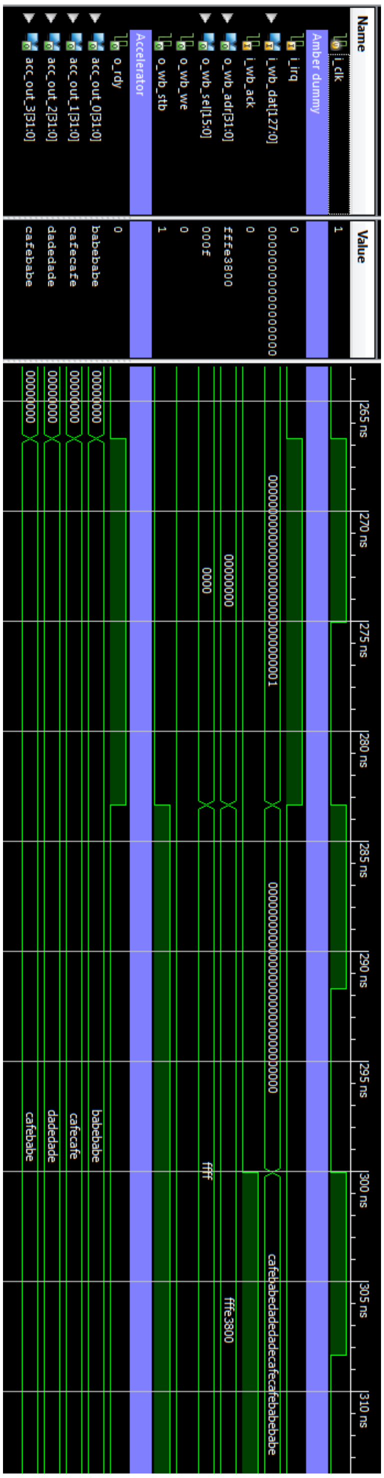


Figure 7.5: Waveform showing an interrupt sequence and a four word read through the Slave IFM

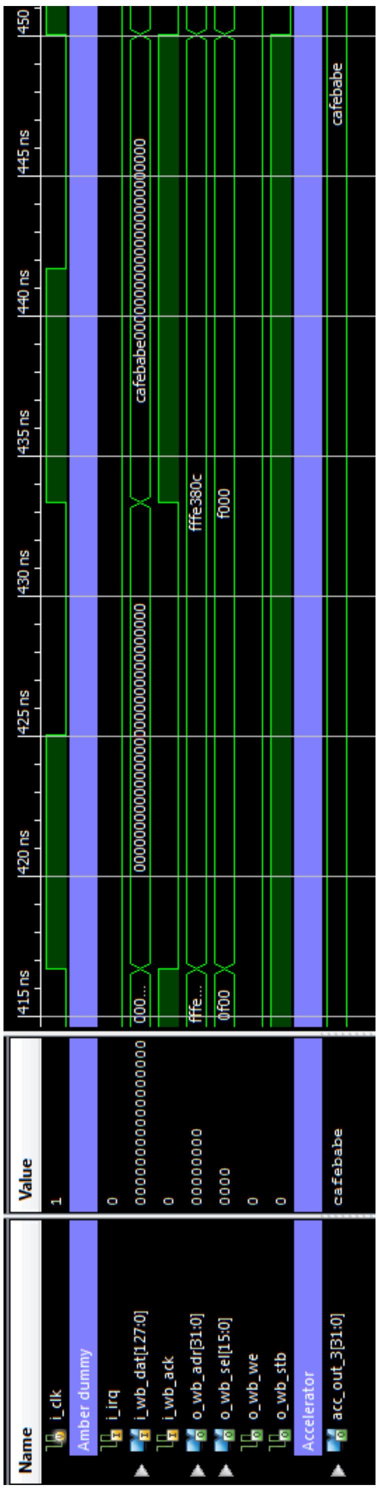


Figure 7.6: Waveform showing a one word read through the Slave IFM

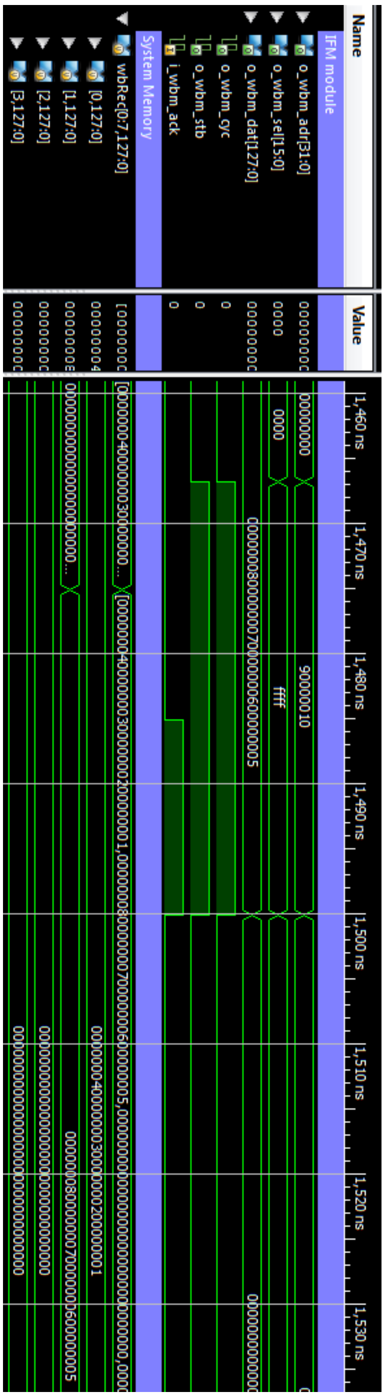
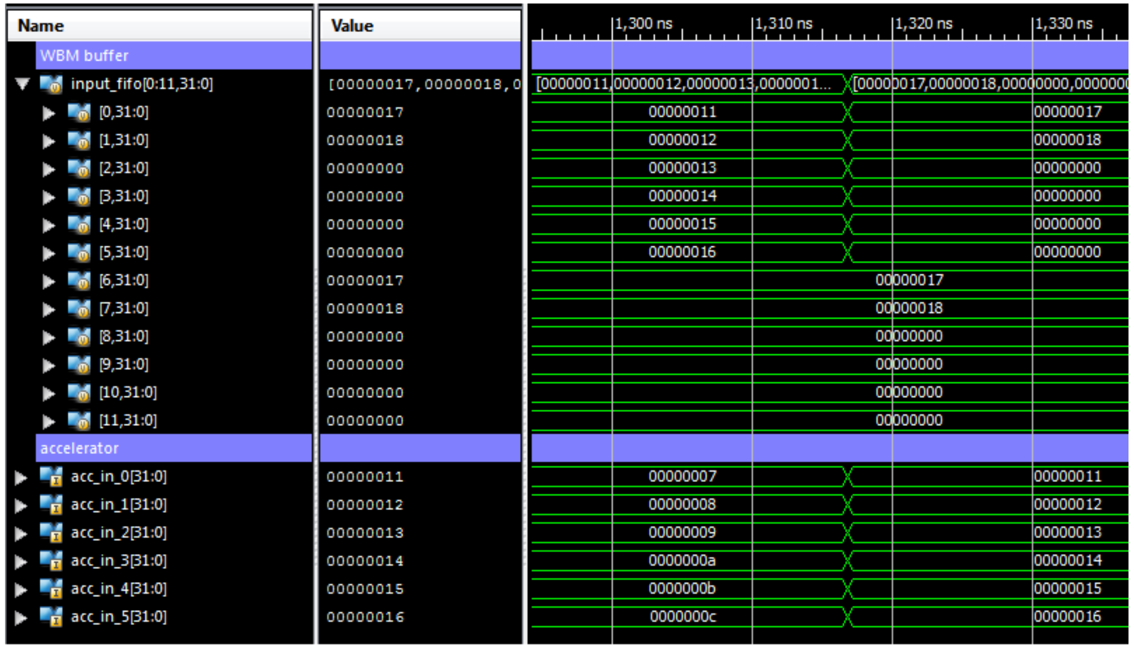


Figure 7.7: Waveform showing the For loop IFM writing system memory



Name	Value
WBM buffer	
input_fifo[0:11,31:0]	[00000017,00000018,0
[0,31:0]	00000017
[1,31:0]	00000018
[2,31:0]	00000000
[3,31:0]	00000000
[4,31:0]	00000000
[5,31:0]	00000000
[6,31:0]	00000017
[7,31:0]	00000018
[8,31:0]	00000000
[9,31:0]	00000000
[10,31:0]	00000000
[11,31:0]	00000000
accelerator	
acc_in_0[31:0]	00000011
acc_in_1[31:0]	00000012
acc_in_2[31:0]	00000013
acc_in_3[31:0]	00000014
acc_in_4[31:0]	00000015
acc_in_5[31:0]	00000016

Figure 7.8: Waveform showing input data transferred by the WMB to a six input accelerator

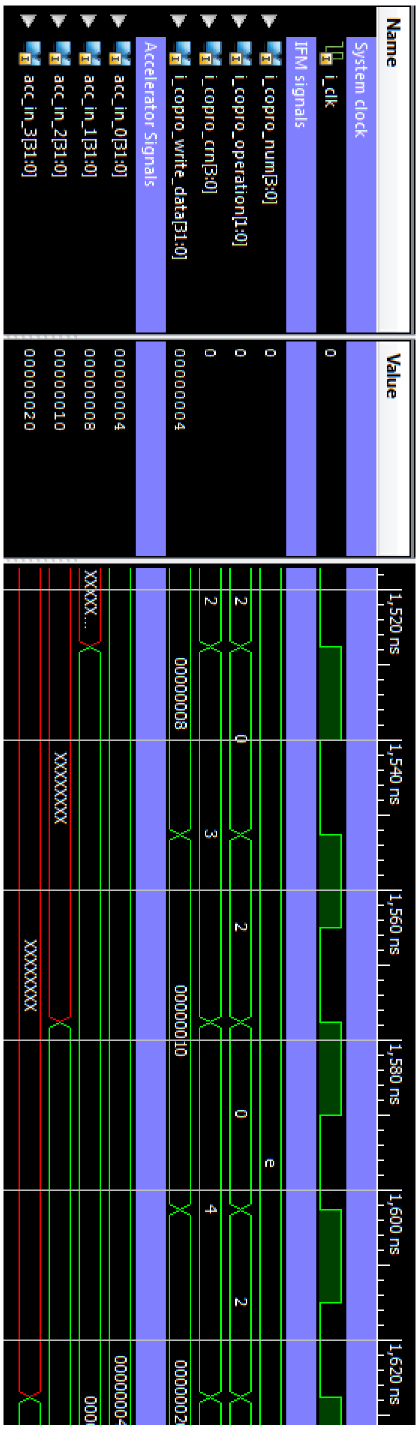


Figure 7.9: Waveform showing input data written to the accelerator through the Coprocessor IFM

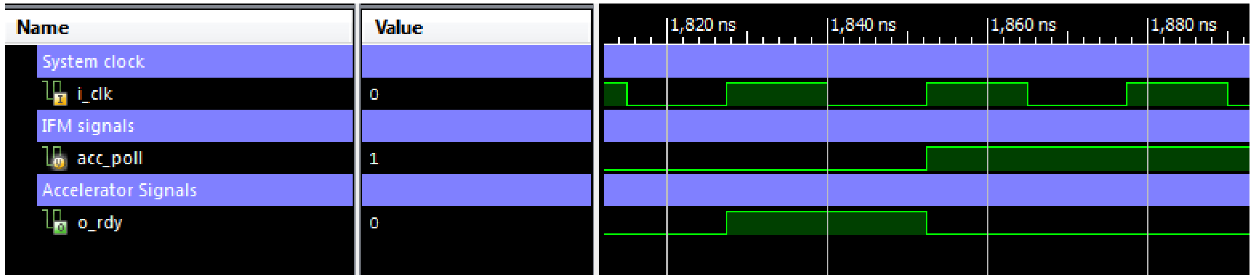


Figure 7.10: Waveform showing poll delay in the Coprocessor IFM

in clock cycles. *I* and *O*,represents the number of input and output ports in the accelerator interface, respectively. The three extra cycles represents the two cycles needed to pass the *options* argument and the one cycle delay needed to update the **poll** register.

$$T_o = 3 + 2I + 2O \tag{7.1}$$

7.5.2 Slave IFM

The Slave IFM is connected to the Amber Core through the Wishbone bus with a slave interface. As this is a passive IFM, the Amber Core has to write all input data and options to the Slave IFM, and read all output data back again. The Amber Core writes one word per Wishbone transfer, and uses one clock cycle per transfer, as seen in Figure 7.1. The Amber Core can read up to four words per Wishbone transfer, as shown in Figure 7.5. In addition to this, Figures 7.5 and 7.4 show that the IFM passes both the *start* signal and the *interrupt* signal without delay. This adds up to total time, in clock cycles per calculation, given in Equation 7.2, where *I* is the number of input ports and *O* is the number of output ports on the accelerator interfaced by the IFM. The extra cycle represents the *options* write.

$$T_o = 1 + I + \left\lceil \frac{O}{4} \right\rceil \tag{7.2}$$

7.5.3 For loop IFM

The For loop IFM is connected through the Amber Core through the Wishbone bus with a slave interface. This connection is used to pass four different control arguments from the Amber Core. As with the Slave IFM and as shown in Figure 7.1, the write interaction from the Amber Core uses one clock cycle and writes one word. When these arguments are passed, the for loop execution starts, then there is no need for any further communication with the Amber Core until the *interrupt* signal is sent when the for loop execution is completed.

During the for loop execution, there is some overhead time consumed by the IFM. Figure 7.11 shows that the minimum time between a *ready* signal from the accelerator until a new *start* signal is given, is five.

In addition to this, the For loop IFM reads and writes data to memory in batches of four. Equation 7.3 gives the overhead time, in clock cycles per execution. The variable *it* is the number of iterations the for loop runs, T_m is memory access time. *I* is the number of input ports and *O* is the number of output ports on the accelerator interfaced by the IFM.

$$T_o = 4 + 5it + it * T_m * \left\lceil \frac{I}{4} \right\rceil + it * T_m * \left\lceil \frac{O}{4} \right\rceil \tag{7.3}$$

7.6 Area Overhead

The area overhead of the IFMs are related to the number of ports on the accelerator, as they are generated to fit. In other words, the area size of the interface is directly dependent on

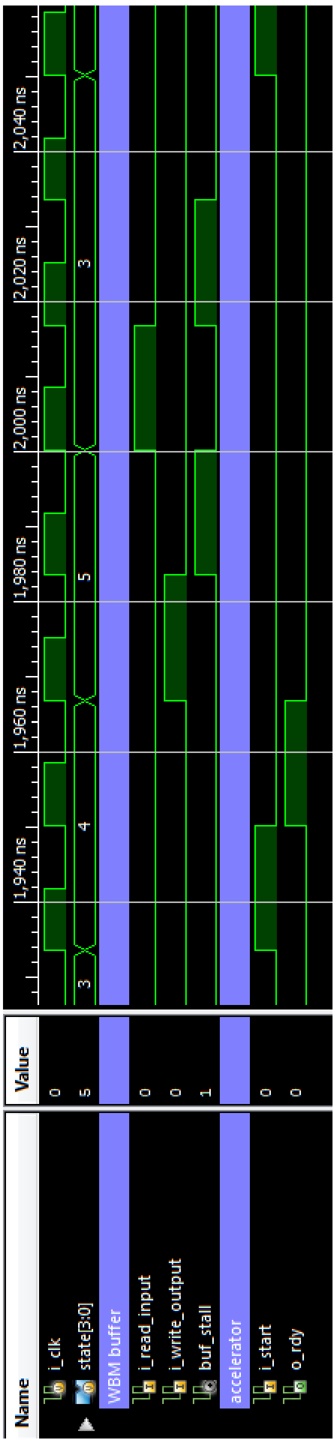


Figure 7.11: Waveform showing minimum delay in the For loop IFM

the accelerator it is created to support, and are therefore difficult to predict. In an attempt to solve this I have assumed that there is a linear relation between the number of ports and the size of the interface. Equation 7.4 shows the assumed relation.

$$A_{tot} = K + NI + MO \tag{7.4}$$

In Equation 7.4 *K* represents a base area for the IFM, *I* represents the size increase to the IFM of one input port and *O* represents the size increase to the IFM of one output port. *N* and *M* represents the umber of accelerator input and output ports, respectively. The values of *K*, *I* and *O* needs to be calculated for each IFM. In order to do so, the Equations 7.5, 7.6 and 7.7 where set up and solved for *K*, *I* and *O* in Matlab[25].

$$A = K + 4I + 4O \tag{7.5}$$

$$B = K + 4I + 1O \tag{7.6}$$

$$C = K + 1I + 4O \tag{7.7}$$

The solutions are shown in Equations 7.8, 7.9 and 7.10.

$$K = \frac{4B + 4C - 5A}{3} \tag{7.8}$$

$$I = \frac{A - C}{3} \tag{7.9}$$

$$O = \frac{A - B}{3} \tag{7.10}$$

The values of *K*, *I* and *O* can be calculated by synthesising IFMs with the structure of Equations 7.5, 7.6 and 7.7 to obtain values for *A*, *B* and *C*. This was done for all three IFMs, and the resulting constants, usable for area prediction, are shown in Tables 7.1, 7.2 and 7.3.

Table 7.1: Constants for area prediction for the Coprocessor IFM

Constant for size calc	Nr of slice registers	Nr of slice luts
base size	204,00	116,33
size of one input	32,00	-10,33
size of one output	32,00	27,00

Table 7.2: Constants for area prediction for the Slave IFM

Constant for size calc	Nr of slice registers	Nr of slice luts
base size	162,00	87,67
size of one input	32,00	22,33
size of one output	32,00	31,00

These numbers have been calculated by synthesizing modified versions of the IFM modules with XIlinx ISE[20]. They have been modified by removing the accelerator from the module and reclassifying the accelerator interface as an I/O interface. Full tables with the raw data are included in Appendix .3.

Table 7.3: Constants for area prediction for the For loop IFM

Constant for size calc	**Nr of slice registers**	**Nr of slice luts**
base size	1043,67	983,33
size of one input	32,00	-4,00
size of one output	32,33	548,67

Chapter 8

Using the IFM system

This chapter is meant to function as a user guide for any accelerator designers and programmers interested in working with the IFM system. The chapter is split into three sections. The first one will summarize the different IFM types to help select which one to use for a given accelerator and give a user guide for the tile generation script. The next section is a programmers manual that will explain how to program the controlling Amber core for the different IFMs, at assembly level. The last section of this chapter will define a general interface for an accelerator, explain the requirements for an accelerator to function with the different IFMs, and explain the different communication interactions between an accelerator and the IFMs.

The general attributes of the different IFMs are shown in Table 8.1. However, it is highly recommended to check the detailed descriptions in Chapter 5 before deciding on an IFM type.

IFM module	**Coprocessor**	**Slave**	**For loop**
IO ports supported	15 input 16 output	511 input 512 output	N\A, theoretically unlimited
Ready handling	pollable ready indication	tile interrupt signal	tile interrupt signal
Options	32 bits	32 bits	32 bits
Start signal	synchronous with options write	synchronous with options write	one per iteration controlled by IFM
Programming interface	mrc and mcr assembly instructions	memory mapped	memory mapped
Data fetch/store	CPU	CPU	IFM

Table 8.1: Summary of the IFM general attributes

8.1 Tile Generator Usage

The Tile Generator is a Python script that requires four input options to function. The options and their valid values are listed in Table 8.2.

Table 8.2: Options and their input values for the Tile Generator

Option	Description	valid entry
-t	IFM type	one of the following strings: coproc, slave, floop
-m	accelerator module name	string
-i	number of input ports	integer
-o	number of output ports	integer

In order to generate a working accelerated tile, all these options must be set and have the correct values. Note that the script does not check if the input values are valid, or that it has one of each option, so the user must take care to get this right. An example of the correct use of the Tile Generator is given in Listing 8.1.

Listing 8.1: Usage example of the Tile Generator

```
python.exe tileGenerator.py -t floop -m acc_dummy_5_5 -i 5 -o 5
```

The example call creates a tile structure named *amber_tile_floopIFM_acc_dummy_5_5* with a For loop IFM interfacing the accelerator *acc_dummy_5_5* with five input and five output ports. Note that the input to the *-m* option is the accelerator **module** name, not the name of the verilog file containing the accelerator. The accelerator file should be placed in the */shmac/hardware/units/accelerator/* folder, and will need to be included in the SHMAC project separately. See Section 9.2.3 for details on project integration.

8.2 Programmers manual

The programmers manual is a description of how a programmer will work with the different IFMs. I will go in to the different methods to work with the IFMs and detail how they can and should be used through the Amber core. The manual is split in to three parts, each detailing the programming methods and mapping of one IFM.

8.2.1 Programming the Coprocessor IFM

The list below summarizes the attributes relevant to programming of the Coprocessor IFM.

- Uses ARM coprocessor instructions
- Uses the coprocessor **14** tag
- Amber core writes control and data directly
- Passive command passing

- Start signal given to accelerator when writing **options** register
- Completion handling by setting **poll** register

register	**reg nr**	**ARM instruction**
Input 0 to 14	0x0 0xE	mcr
Output 0 to 14	0x0 0xE	mrc
Options	0xF	mcr
poll	0xF	mrc

Table 8.3: Adress list for Coprocessor IFM

The Coprocessor IFM is the most core-near of the IFMs and is controlled by using the coprocessor register transfer instructions, mcr and mrc, found in the ARM ISA[13]. The standard use case is pretty straight forward, but the IFM's passive control nature will allow for some variations. See Section 5.4 for details on the module implementation.

The Coprocessor IFM has very little control on the interactions between the programmer and the accelerator. The programmer can decide to write or read any register at any time, and the corresponding values and signals will be passed on to the accelerator without any check of context. For example, if an input register is written during a calculation, the input port of the accelerator will be updated without any regard to preceding *start* signals or the state of the accelerator's *ready* signal. This means that the IFM does not limit the possibility of several consecutive *start* signals from the programmer or several consecutive *ready* signals from the accelerator.

There is however one feature on the Coprocessor IFM that is reliant on a more traditional way of passing signals, the poll functionality. The IFM has a **poll** register, that can be used to check if the previously initiated calculation has completed. When a *start* signal is given, in other words when the **options** register is written, the value of the **poll** register is set to 0. It will remain 0 until the first following *ready* signal from the accelerator. Any consecutive *ready* signals does not change the value, only a new *start* signal will change the value back to 0. This allows for a traditional use of an accelerator, shown in the example code in listing 8.3.

There are only a few commands a programmer can give the accelerator through the IFM. These commands and the responding actions are listed below.

Write input Writing an input port on the accelerator is done with the mcr instruction, with the corresponding coprocessor register number set.

Write options This is done by using the mcr instruction with the coprocessor register number 15. This action does three things, it sets the options given on the **i_options** port of the accelerator, gives a one cycle high start signal on the **i_start** port of the accelerator, and sets the value of the **poll** register to 0.

Read outputs Reading an accelerator output is done with the mrc instruction with the corresponding coprocessor number set. This will return to the Amber core the value in the corresponding output buffer in the IFM. See section 8.3.1 for details on how the buffers are updated.

Read poll Reading the poll signal is accomplished by using the mrc instruction with the coprocessor register number 15. This returns a value of 1 if there has been a *ready* signal from the accelerator after the previous *start* signal. If there has been a *start* signal and no following *ready* signal, this will return a value of 0.

Listing 8.2: ARM coprocessor register transfer instructions with examples of use

```
1 mcr p#, opcode, Rs, crn, crm
2 mrc p#, opcode, Rd, crn, crm
3
4 @examples for use with coprocessor IFM
5 mcr 14, 0, r0, cr1, cr5
6 mrc 14, 0, r1, cr0, cr5
```

As we see in listing 8.2, the mcr and mrc take several arguments, not all of which is relevant to the Coprocessor IFM. The *opcode* and *crm* arguments are ignored by the IFM, so the value given are irrelevant. The other arguments are described in Table 8.4.

Table 8.4: Arguments used for the mrc/mcr instructions

Argument	Description
p#	Coprocessor nr
Rd	Destination register
Rs	Source register
crn	Coprocessor register nr

Lets look at the first example, line 5 in listing 8.2. This instructions writes the contents of the Amber core's *r0* to coprocessor register 1(*cr1*) in coprocessor *14*. The next example, at line 6, reads coprocessor register 0(*cr0*) in coprocessor *14* to the Amber core's *r0*. Finally, in listing 8.3, a small program that runs one calculation on an accelerator in the Coprocessor IFM is given.

Listing 8.3: Simple assembly program to run a 2 input 2 output accelerator in the coprocessor IFM

```
    mcr 14, 0, r0, cr1, cr5    @ write acc_in 0
    mcr 14, 0, r1, cr2, cr5    @ write acc_in 1
    mcr 14, 0, r2, cr15, cr5    @ write options to start

    @check if ready
  ldr      r0, =0x00000001
pollpoint:
    mcr 14, 0, r1, cr15, cr5    @ read poll port
  cmp r0, r1
  bne   pollpoint

  mrc 14, 0, r0, cr0, cr5     @read acc_out 0
    mrc 14, 0, r1, cr1, cr5     @read acc_out 1
```

The program writes the two input registers and then the **options** register, triggering a start signal to the accelerator. It then waits for the accelerator to give a ready signal by checking the **poll** register of the IFM. When the poll goes high, the program continues on to reading the two output registers from the Coprocessor IFM.

8.2.2 Programming the Slave IFM

The list below summarizes the attributes relevant to programming of the Slave IFM.

- Memory mapped programming interface
- Uses the wishbone bus system through a slave interface
- Amber core writes control and data directly
- Passive command passing
- Start signal given to accelerator when writing **options** register
- Completion handling by giving an *interrupt* signal to the on tile Interrupt controller

unit	**Absolute address [31:0]**	**Local byte address**	**Local word address**
Slave IFM	0xFFFE 3000	[15:0] 0x3000	[15:2] 0xC00
Input 0 to 511	0xFFFE 3000 0xFFFE 37F8	[11:0] 0x000 0x7F8	[11:2] 0x000 0x1FE
Output 0 to 512	0xFFFE 3800 0xFFFE 3FFC	[11:0] 0x800 0xFFC	[11:2] 0x200 0x3FF
Options	0xFFFE 37FC	[11:0] 0x7FC	[11:2] 0x1FF

Table 8.5: Address mapping for Slave IFM, bit indication given from absolute address

The Slave IFM is a memory mapped IFM connected to the Amber System's 128 bit wishbone bus with a slave bus controller, and is capable of handling four word bus transfers for both reading and writing. Please note that at the time of writing the Amber core does not support more than one word writes. See section 5.5 for details on the module implementation.

The Slave IFM incorporates the same passive level of control as the Coprocessor IFM. The Slave IFM passes signals both ways without context, meaning that the timing of input writes, *start* signals, *ready* signals and output buffer writes is completely in the hands of the programmer and the accelerator designer.

The interrupt feature of the Slave IFM is implemented with a port connected to the Interrupt Controller on the Amber Tile, and is controlled without any interference by the IFM. when a *ready* signal is given by the accelerator, a one cycle *interrupt* signal is given to the Interrupt Controller. This functionality does not require any preceding *start* signals or register writes.

As with the Coprocessor IFM, the commands that a programmer is able to give to the IFM is limited. These commands and the responding actions are listed below.

Write input Writing an input port on the accelerator is completed by writing a word to the corresponding memory address, given in table 8.5, by the Amber core.

Write options This is done by writing the memory address given in table 8.5. This action does two things, it sets the options given on the **i_options** port of the accelerator and gives a one cycle high start signal on the **i_start** port of the accelerator.

Read outputs Reading an accelerator output is done by reading the memory address in table 8.5. This will return to the Amber core the value in the corresponding output buffer in the IFM. See section 8.3.1 for details on how the buffers are updated.

8.2.3 Programming the For loop IFM

The list below summarizes the attributes relevant to programming of the For loop IFM.

- Memory mapped programming interface
- Runs a self contained for loop execution, able to run in parallel with the Amber Core
- Fetches input data and writes output data to off tile memory with minimal control from the Amber Core
- Uses the wishbone bus system, with both slave and master controllers
- Amber core writes control data only
- Strict command passing, actively controls the accelerator
- Starts loop execution when writing the **options** register
- Completion handling by giving an *interrupt* signal to the on tile Interrupt controller

unit	Absolute address [31:0]	Local byte address	Local word address
Master IFM	0xFFFE 3000	[15:0] 0x3000	[15:2] 0x000
Options	0xFFFE 3000	[15:0] 0x3000	[11:2] 0x000
Main i	0xFFFE 3004	[15:0] 0x3004	[11:2] 0x001
input address	0xFFFE 3008	[15:0] 0x3008	[11:2] 0x002
Output address	0xFFFE 300C	[15:0] 0x300C	[11:2] 0x003

Table 8.6: Address mapping for For loop IFM

The For loop IFM differs a great deal from the other IFMs. It is a For loop accelerator interface capable of running parallel to the Amber core. This feature creates the necessity of a stricter handling of control signals, this means that the For loop IFM does not pass instructions between the programmer and accelerator passively. The tile design is modified with a round robin arbiter to share the outgoing memory interface between the IFM and the Amber Core. See Section 5.6 for details on the module implementation.

The For loop IFM is controlled by a Finite State Machine(FSM) that handles the control variables from the programmer, the execution of the for loop and the flow of variables and control signals to the accelerator. This has some major implications for the programming model. First and foremost, the IFM does only accept input control words when in the appropriate state. This is after a *reset* signal, before the first *start* signal or after a loop execution is completed, in other words when a given *start* signal has been answered with an *interrupt* signal from the IFM.

The programming control interface holds some similarities to the other IFMs, but is clearly distinct. It takes four control variables to operate, filling the **options**, **Main_i**, **input_address** and **output_address** control registers. The IFM's Wishbone interface accepts any number of words written, up to and including the bus width of four words. Please note that at the time of writing the Amber core is only capable of writing one 32 bit word on the bus. These are written by addressing the memory according to table 8.6. It is important to remember that writing the **options** control register also initiates the loop execution. Make sure that the other three control registers are written before or at the same time as the **options** register, as the IFM will not accept any new input until the execution is done. All control registers are 32 bit registers, and are described below.

options, tile address 0x3000 This is the options register, passed directly to the accelerator. Writing this control register also initiates the for loop execution, so it should always be written after or at the same time as the other control registers.

Main_i, tile address 0x3004 This is the iterator variable of the for loop. The IFM will iterate for as many cycles as indicated by the value written to this register.

Input_address, tile address 0x3008 This is the memory address of which the IFM will fetch the input data for the accelerator. The IFM assumes that the first input data value, going to the **input_0** port of the accelerator on the first iteration of the first loop is at this address, and that the consecutive input data words are in the consecutive addresses. In order to comply with the off tile memory system, this address need to be four word aligned. This means that to avoid errors due to address offset, this address need to comply with the form **0xXXXX XXX0**.

Output_address, tile address 0x300C This is the memory address of which the IFM will write the output data from the accelerator. The IFM will write the first output data word from the first for loop iteration to this address, and the following data words to the consecutive addresses. This value is also subject to the limitations of the off tile memory system, and will need to comply with the form **0xXXXX XXX0**.

8.3 Communication between accelerator and IFM

This section gives a quick look at the communication cycles between the different IFMs and a given accelerator. For details on the design and architecture of the accelerator interface required to work with the IFMs, see Chapter 4.

8.3.1 Communication cycles with the Coprocessor and Slave IFMs

General interaction

Figure 8.1 shows the standard communication cycle between the basic IFMs, the Slave and the Coprocessor, and interface of an accelerator with four inputs and one output. The first action that happens is that the *acc_in_* signals are set by the IFM. When the options array, *i_opt*, is set, the **i_start** signal is set high for one cycle. Next, the accelerator sets *o_rdy* high for one cycle. When this happens, the IFMs *acc_out* buffer is written on the rising edge of the *o_rdy* signal. The *o_rdy* signal also triggers the ready handling of the IFM, see section 8.3.1 for details.

Special cases and differences between the Coprocessor and Slave IFMs

The basic IFMs does not actively limit the interactions between the controlling Amber system, in other words the programmer, and the accelerator. Any interaction initiated by either the Amber system or the accelerator are treated separately by the IFM. I will now elaborate on what this means for the accelerator communication cycles.

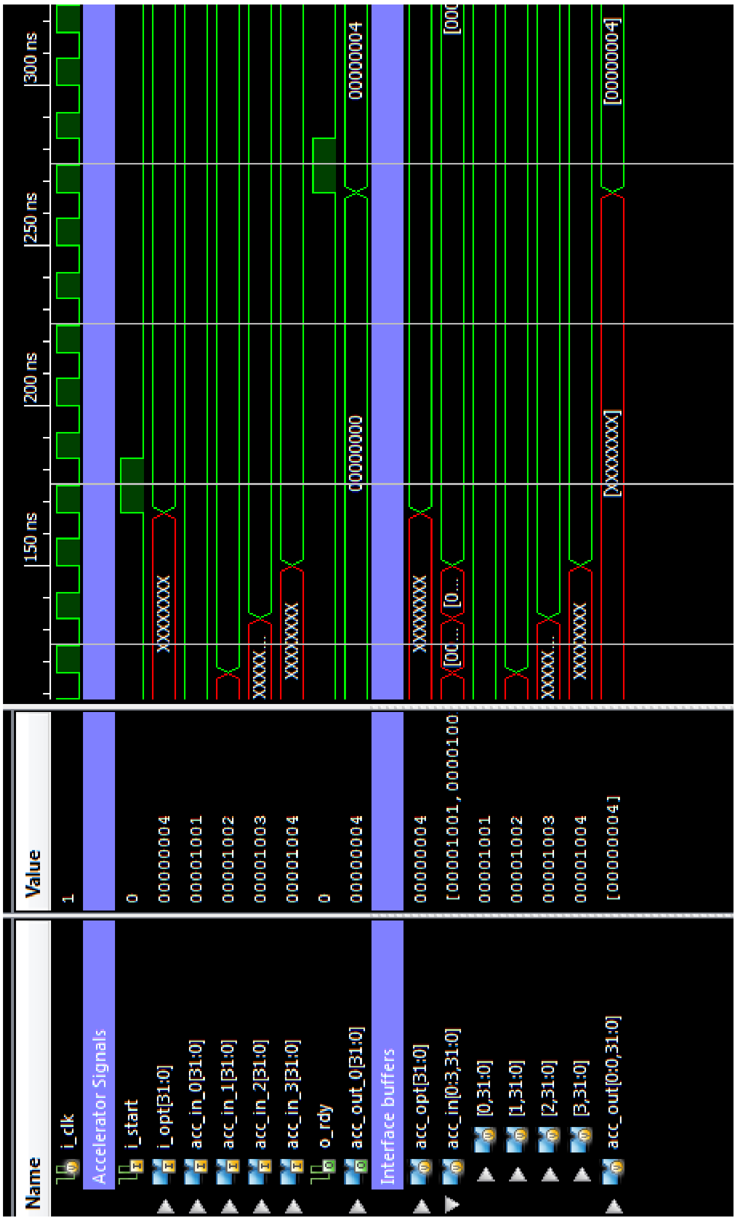
Figure 8.2 shows that the *options* signal, with the synchronously set **i_start** signal, can change several times before the accelerator gives the *o_rdy*. With the basic IFMs these interactions are solely controlled by the parent system, and must be carefully timed by the programmer and accelerator designer. Note that the IFM responds normally to the high state of the *o_rdy* signal, by setting the *acc_out* buffer on the following rising clock edge.

Similar to the IFM's handling of several option writes, the IFM responds to every ready signal, indicated by *o_rdy*, regardless of preceding start signal, as shown in figure 8.3.

So far in this section the two basic IFMs have been equal in their interface to the accelerator. Now we shall look at the differences. Figure 8.4 shows four modules. From the top we have **Accelerator 1**, who is connected to the module **IFM_coproc**, a coprocessor IFM. **Accelerator 2** is interacting with the **IFM_slave** module. Note that there is no connection between the IFMs or the accelerators, they are shown in the same figure purely for comparison purposes.

When the *acc_start* signal of **Accelerator 1** goes high, an internal signal in **IFM_coproc**, called *acc_poll* goes low. This is a register than can be polled by the parent system to check if a calculation is completed. When the next high ready signal is set, this register is set high. The *acc_poll* register can only be set low by a new start signal, and will not be changed by any consecutive ready signals from the accelerator. The writing of the **IFM_coproc**'s *acc_out* buffer register is not affected by this, and will update as previously described.

The lower half of figure 8.4 shows the response of the **IFM_slave** module to the *o_rdy*



Name	Value
i_clk	1
Accelerator Signals	
i_start	0
i_opt[31:0]	00000004
acc_in_0[31:0]	00001001
acc_in_1[31:0]	00001002
acc_in_2[31:0]	00001003
acc_in_3[31:0]	00001004
o_rdy	0
acc_out_0[31:0]	00000004
Interface buffers	
acc_opt[31:0]	00000004
acc_in[0:3,31:0]	[00001001,0000100
[0,31:0]	00001001
[1,31:0]	00001002
[2,31:0]	00001003
[3,31:0]	00001004
acc_out[0:0,31:0]	[00000004]

Figure 8.1: A IO cycle between an accelerator and the Slave IFM during one complete calculation

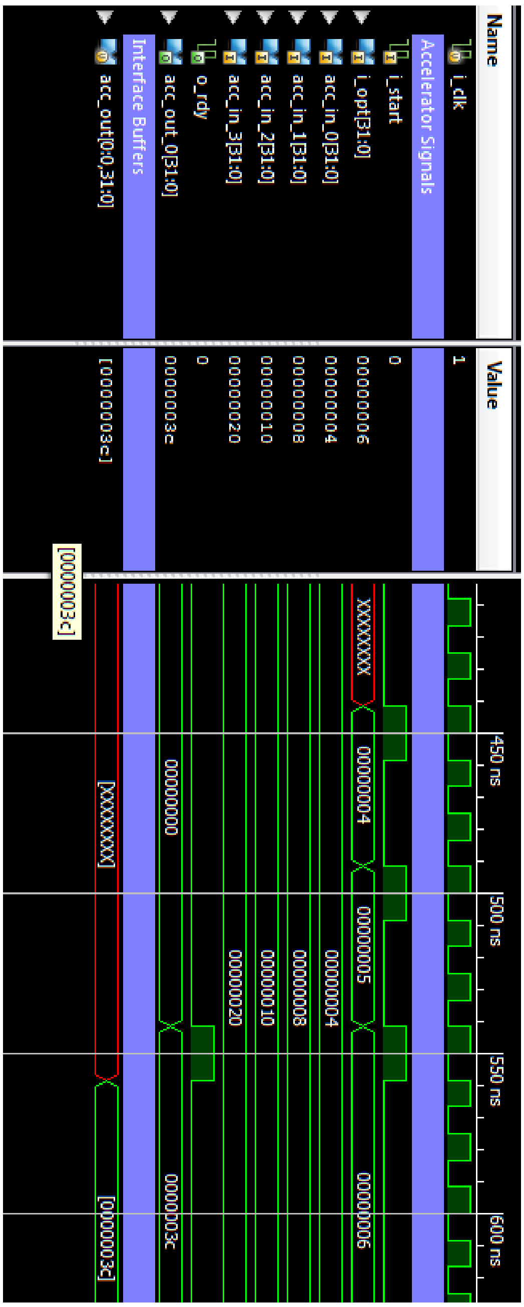


Figure 8.2: Several options writes unrelated to the responding ready signal from the accelerator

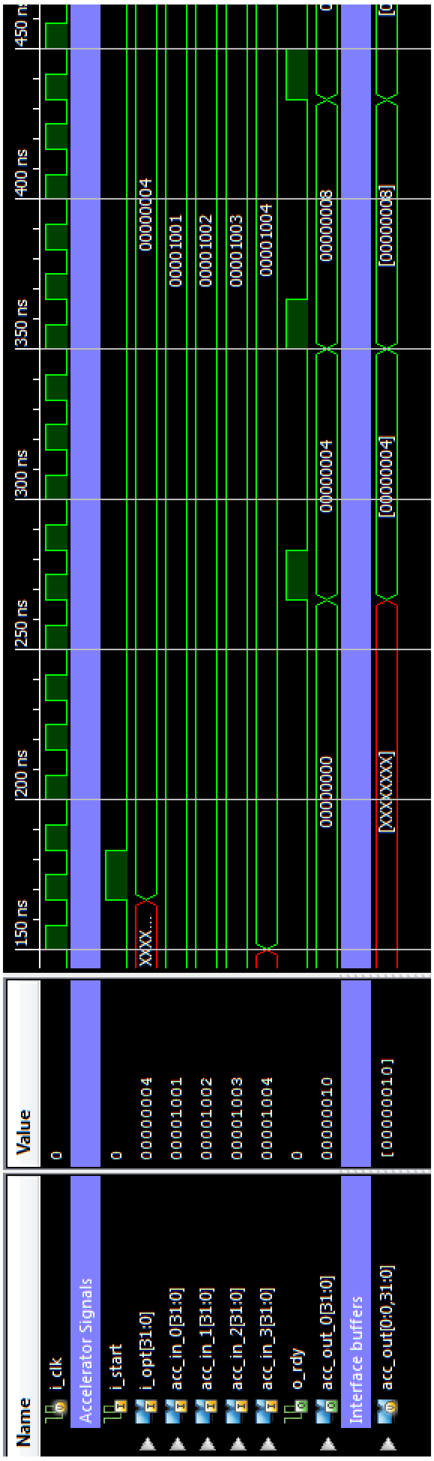


Figure 8.3: Several ready signals from the accelerator triggers writing of the IFM buffers

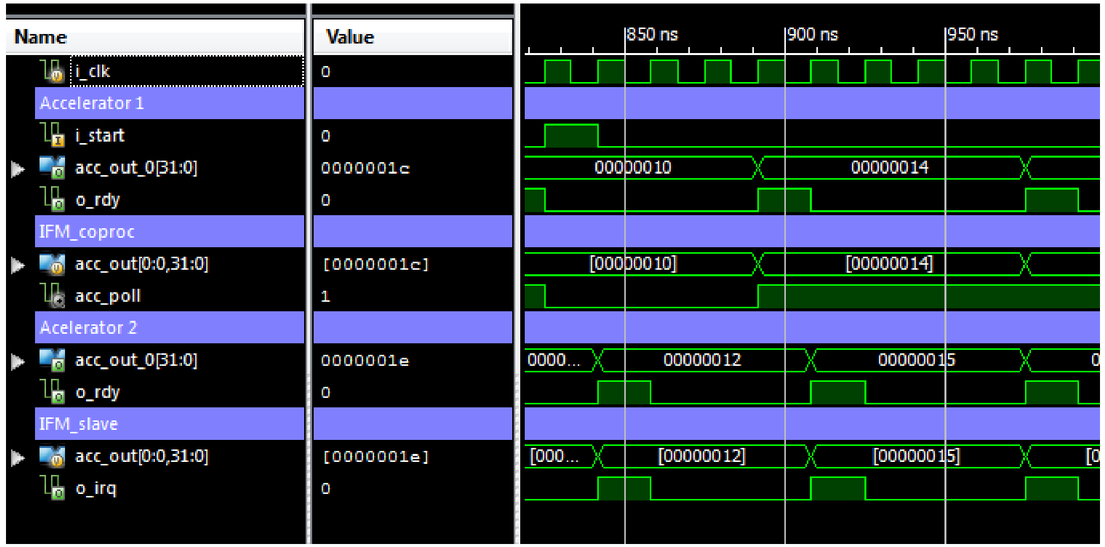


Figure 8.4: A waveform showing the differences between the response to a ready signal from the Coprocessor and the Slave IFMs

signal of **Accelerator 2**. The Slave IFM has a port named **o_rdy**, that is connected to the interrupt controller in the Amber Tile(3.2). Figure 8.4 shows how the *o_rdy* triggers an interrupt signal from the **IFM_slave** module, in addition to the *acc_out* buffer write. This signal resets after one clock cycle, and will therefore be set on any consecutive ready signals.

8.3.2 Communication cycles with the For loop IFM

The For loop IFM differs from the Coprocessor IFM and the Slave IFM in the way it communicates with the accelerator. The operation of the for loop structure and the unpredictable interaction with the memory through the wishbone bus sets some extra demands and constraints on the accelerator interface. Figure 8.5 shows the interface timing of accelerator execution by the For Loop IFM. The accelerator used for this simulation is a two input, two output accelerator.

The cycle starts with a one cycle start signal on the accelerators *i_start* signal. The input signals, *acc_in_n*, is held from the rising flank of the start signal until the next start signal. The For loop IFM then waits for the accelerators *o_rdy* signal to go high for one cycle. It is important to note here that the *acc_out_m* is not read on the falling flank of the ready signal. Due to the unpredictable nature of the Wishbone communications, the IFM might stall at this point for a number of cycles. It is therefore important that the accelerator holds the values on the output ports until the next start signal, to avoid loss of data.

It is important to note that there are no special cases for the accelerator interface of the For loop IFM. The communication timing is strict. Failure to comply with the timing can result in loss of data and a situation where the IFM stalls in an infinite loop.

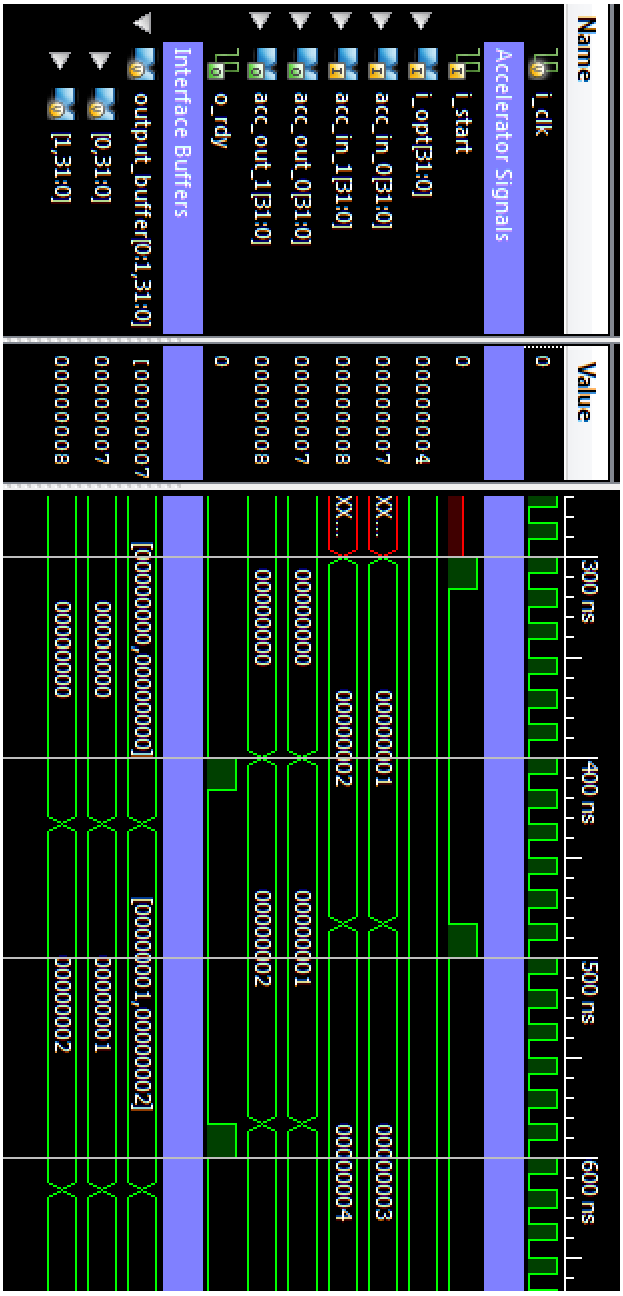


Figure 8.5: Accelerator interactions with the For loop IFM

Chapter 9

Discussion and Future work

9.1 Discussion

9.1.1 General Accelerator Interface

When defining a generalisation, it is almost impossible to create something that works perfectly for every conceivable application of that generalisation. This is also true for the General Accelerator Interface proposed in Chapter 4. The interface is designed to work with as many different accelerators as possible. One effect of this is that there are several accelerators for which this interface is not optimal. One example of this is the fact that the interface is clocked, which for an asynchronous accelerator is unnecessary, and therefore an unused port. The interface, as well as the IFMs, does however support asynchronous accelerators, remaining useful for this type of accelerator.

Another feature of the interface that might not be optimal is the 32 bit **options** port. Many accelerators do not require any options, and others require only a few. On the other hand, this feature enables accelerators to expand its functionality greatly, and is a requirement for many accelerators.

With the previous points in consideration, the General Accelerator Interface remains a highly functional and versatile interface. All accelerators mentioned in Section 4.1 and in [19] can easily be fitted in to this interface, with only small modifications and excess area.

9.1.2 IFM Design

The IFM modules were designed first with functionality in mind, and then optimized. They need to function with accelerators of unknown size, which made the functionality very important. There are no known issues with the Coprocessor or Slave IFM that are unoptimized. However, the For loop IFM has some known issues making it ineffective in some areas.

The Wishbone Master Buffer(WMB) is designed to handle memory reads and writes in four word transmissions for any number of accelerator ports. Furthermore, the nature of a for loop and the FSM requires that the WMB is able to present and store data to and from the accelerator at specific times in the execution process to ensure data validity. In order to ensure this functionality in the amount of time available for design, the WMB has been designed in a safe but slow way. The read and write tasks are not handled separately, and the buffer to FIFO process is not optimized. This, in combination with the FSMs flow, is the main reason for the five cycle overhead time found in Section 7.5.3. Suggestions for improvements can be found in Section 9.2.1.

9.1.3 IFM verification

The verification of the IFMs have been completed with the methods mentioned in Section 7.1. The Coprocessor IFM has been verified by simulating in the Amber Test Framework(ATF), and the Slave and For loop IFMs have been verified with a testbench and a module simulating the behaviour of an Amber Core as recorded with the ATF. While the Coprocessor IFMs verification is clearly safer, with fewer sources for error, than the Slave and For loop IFM, both verification methods suffer from the fact that they are simulations. The functionality of the IFMs are not completely verified before they are tested in the SHMAC FPGA platform, but the SHMAC project is not at a level where it can offer this yet.

This means that I can not say that there will not be any issues with the design and functionality when the generated tiles and the IFMs are implemented in to the SHMAC architecture, but I am confident that the basic functionality has been verified to the extent possible with the available resources.

9.1.4 IFM Overhead

Time

In Section 7.5 the time used as overhead with the use of the different IFMs is given in Equations 7.1, 7.2 and 7.3. It is however important to take in to consideration the level of parallelism offered by the different IFMs when discussing time overhead.

The Coprocessor IFM offers practically no parallelism in the conventional use case due to its use of the poll method for completion notification 5.3. This means that the Coprocessor IFM needs to continuously check if the accelerator is finished, which takes up the entire potential of the Amber Core.

The Slave IFM offers a higher level off parallelism than the Coprocessor due to its use of the interrupt method for completion notification. This means that as soon as the Amber Core has started the calculation it can complete other tasks as the IFM will notify with an *interrupt* signal when the accelerator completes its calculation. It does however still require that the Amber Core reads output and writes input data, which takes the amount of time stated in Equation 7.2.

The For loop IFM, which is very independent and uses the interrupt method, offers the greatest level of parallelism. When the Amber Core has passed the four arguments required to start the calculation, all data passing is done by the IFM. This means that the Amber Core can execute any other task while the For loop executes. It is however important to take in to consideration that while the For loop IFM runs, the off tile memory interface is shared. This will impact the execution time of the Amber Core, especially if the accelerator interfaced by the IFM has a short execution time.

Area

The calculations we have for area overhead in Section7.6 are compromised in several ways. We are interested in the size of the IFM, not the accelerator. However, in HDL synthesis it is non trivial to separate these without compromising the area overhead data. It is necessary to construct the test module in such a way that no important overhead is removed by optimization and the accelerator is as small as possible, or possible to subtract from the design after the fact. The only solution found giving reasonable data was to reclassify the internal accelerator interface as an I/O interface on the IFM module. This does create extra overhead related to I/O logic, but it was the best option available.

Area calculations are split into two mayor parts for FPGAs, *number of registers* and *number of LUTs*. The calculations in Section 7.6 show that for number of registers there are a clear linear relationship between number of ports of the implemented accelerator and the number of registers. Every 32 bit port adds 32 registers, in addition to a constant base size for each IFM. With the LUT calculations the relationship is not as clear. For example, the size of a Coprocessor IFM, in LUTs, are diminishing with every extra input port, according to Table 7.1. This is clearly wrong, and shows that the relationship between the number of accelerator ports and the number of LUTs are not strictly linear. However, comparisons made with these numbers and synthesis results from other Coprocessor IFMs show that the factors give a usable approximation for IFMs with small numbers of accelerator ports.

The results of area predictions are, as mentioned, compromised. The numbers can be used for rough area predictions, with more precise results for registers than LUTs, but the best way to get a solid number is to implement an accelerator with the TileGenerator, synthesize the tile and get overhead data for the entire tile. This will give the Xilinx synthesizer full use of it's optimizations and give the most precise results.

Energy

One of the major focus area of the SHMAC project is energy efficiency. Unfortunately the current state of the SHMAC project[12] does not include an easy way to measure and calculate energy usage at module level and due to time constraints I have not been able to circumvent this. However, one can say in general terms that the use of overhead energy is closely related to overhead in time and area. In other words, a short overhead time and a small overhead area translates to less overhead energy.

9.1.5 TileGenerator

The TileGenerator is able to generate synthezisable tiles from four arguments, creating a new tile structure interfacing a given accelerator. The functionality of the script is tested and verified and the generated modules are used in the tests performed in order to verify the designed IFM modules, described in Chapter 7.

However, the script has some minor issues, some of which are described as targets for further development in Section 9.2.2. In addition to this there is the known issue of checking and recognising erroneous arguments. The script does not do this at this point and requires the user to only input the correct number of valid arguments to function properly.

9.2 Future Work

In this project, there are several features and optimizations that have been left out due to time constraints. This section describes the most prominent unoptimized issues and the most important expansions that are possible to make in future work.

9.2.1 Improving the IFMs

The IFMs as they stand today offer proven functionality, but there are possible improvements and optimizations known today. In order to change the generated IFMs, the template files in */shmac/hardware/tileGenerator/templates/* will need to be changed. I recommend generating an IFM, modifying it in Xilinx ISE, and update the template files with verified modifications only.

Expanding poll and interrupt functionality The IFMs today are equipped with one type of completion notification method each. The Coprocessor IFM uses the poll method and the Slave and For loop IFMs use the interrupt method. It is however viable to expand on this and implement both for all IFMs. I suggest to combine it with the suggestion in Section 9.2.2 to give the script an option to choose between the two methods.

Expanding the address space of the Coprocessor IFM In Section 5.4 it is explained that the Coprocessor IFM uses the four bit *crn* signal to address the internal registers, allowing 16 addresses in total. There are however several unused signals in this interface, including *crm*. If this signal was added to the addressing method, the total number of addresses would be 256.

Optimizing the For loop IFM The For loop IFM is a complex and independent IFM, but at the moment it is slow. Most of the five cycles(Section 7.5.3) spent in overhead between an accelerator's computations are spent waiting for the Wishbone Master Buffer(WMB). I would suggest focusing optimizing efforts on this module. Examples of improvements would be to separate the read and write functionality into two separate modules, and to try to increase the combinatorial fraction of the control logic in this module. In addition, the size of the FIFO queues in this module is generated after the ”at least big enough” philosophy, and would benefit from a more optimized calculation of size based on the number of accelerator ports.

9.2.2 Expanding the TileGenerator script

The Tile Generator as it stands today supplies a basic IFM tile generation functionality. However, there are a lot that can be added in functionality. A list of suggested expansions is listed below.

Improving structure the script would definitely benefit from some work with the structure. Placing often used code in to functions and making the script easier to understand and read would help any future developers intent on expanding and improving the TileGenerator

Option for poll/interrupt can be implemented to give the accelerator designer the option between a poll or an interrupt(or both) completion notification from the IFM, given the suggestion for future work in Section 9.2.1

Several accelerators per tile it could be preferable to have several accelerators per tile, and the script could be expanded to include such possibilities

Option for the *Options* port can be implemented as a way of optimizing area for accelerators that does not require an options input. This does however require extensive modification of the IFM templates, and will require large amounts of work.

Any new feature the script is an easy to use and easy to modify Python script, and could easily be adapted to include any future customization wanted in the tile architecture, whether related to accelerator development or not, the TileGenerator is an efficient way of adapting tiles.

9.2.3 System integration of generated tiles

The TileGenerator as it stands today creates a modified tile compatible with the SHMAC platform, but it does not integrate it for compilation and synthesis. In order to do this, a few files in the *shmac* repository needs to be changed, listed below.

/shmac/hardware/scripts/toplevel.py needs to be updated with a *def* section for the new tile

/shmac/hardware/scripts/create.py has to be updated with a letter assignment for the new tile, for use with the */shmac/hardware/shmac/setup.txt* file

/shmac/hardware/shmac.prj all generated files will need to be added in this file

Chapter 10

Conclusion

The main goal of this thesis has been to design and implement a general and efficient method for interfacing a wide variety of accelerators. I have created a system that can offer three different Interface Modules(IFM) for programming and interfacing a given accelerator. Furthermore I have created a scripted system for generating a modified Amber Tile that implements the given IFM and is compatible with the SHMAC platform.

In order to create a general technique for accelerator integration in the form of IFMs a General Accelerator Interface has been defined. This interface is modelled on several known accelerator traits from both literature and accelerators currently being developed for the SHMAC platform and provides a minimal interface that can be used by a wide variety of accelerators.

The IFMs have been designed to provide several levels of integration and control of an accelerator by the Amber Core structure. Ranging from the minimal structural changes and high control demands of the Coprocessor IFM to the highly independent but structurally large For loop IFM, the proposed IFMs are able to meet the requirements and demands of a wide range of diverse applications.

The Tile Generator Python script wraps the IFMs and the complexity of the different architectures into an easy to use system. This system offers a user the possibility, with a minimum of effort and design time, to generate an accelerated Amber Tile. This accelerated Amber Tile implements a given accelerator with the chosen IFM, ready for integration into the SHMAC platform.

The General Accelerator Interface and the IFMs, together with the Tile Generator provides a simple and functional method for interfacing and programming a wide variety of accelerators without too much of the compromise to efficiency normally associated with heavily generalized interface solutions.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics 38, 8*, April 1965.

[2] J. L. Hennesy and D. A. Patterson, *Computer Architecture, A Quantative Approach, Fifth Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[3] M. B. Taylor, "Is dark silicon useful?," *DAC*, 2012.

[4] R. H. D. et al., "Design of ion-implanted mosfet's with very small physical dimensions," *Solid State Circuits, IEEE journal of 9.5*, 1974.

[5] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Proceedings of the 38th annual international symposium on Computer Architecture*, 2011.

[6] Y. Zhang, L. Peng, X. Fu, and Y. Hu, "Lighting the dark silicon by exploiting heterogeneity on future processors," *DAC*, 2013.

[7] W. Wolf, *Computers as components, priciples of embedded computing system design*. Elsevier Inc, 2008.

[8] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," *ASPLOS*, 2010.

[9] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," *HPCA*, 2011.

[10] P. M. S. jr, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell, "Hippai: High performance portable accelerator interface," *High Performance Computing (HiPC), 2009 International Convention on*, 2009.

[11] L. T. Rusten and G. I. Sortland, "Implementing a heterogeneous multi-core prototype in an fpga," Master's thesis, Norwegian Univerity of Science and Technology, 2012.

[12] EECS, “Single-isa heterogeneous many-core computer project plan,” 2014.

[13] “Arm7di data sheet.” `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0027d/index.html`, December 1994.

[14] “Opencores, amber core datasheet.” `http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf`, February 2014.

[15] “Opencores.” `http://opencores.org/`, February 2014.

[16] OpenCores, *Wishbone System-on-Chip(SoC) INterconnection Architecture for Portabe IP Cores*. OpenCores, 2010.

[17] “Combining wishbone interfrace signals, application note.” `http://opencores.org/opencores,wishbone`, April 2001.

[18] B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, “Quantifying acceleration: Power/performance trade-offs of application kernels in hardware,” *ISLPED*, 2013.

[19] M. L. Teilgrd and N. Sunniva Nergaard Berg. IET, “Evaluation of basic block accelerators for use on the shmac platform,” 2013.

[20] “Xilinx ise.” `http://www.xilinx.com/products/design-tools/ise-design-suite/`, June 2014.

[21] “Opencores, amber project user guide.” `http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-user-guide.pdf`, May 2013.

[22] A. L. Indegrd, “Configurable floating-point unit for the shmac platform,” Master’s thesis, Norwegian Univerity of Science and Technology, 2014.

[23] N. Einar Johan Tran Smen. IDI, “A cellular automata accellerator for shmac,” 2014.

[24] S. N. Berg, “Implementation of epileptic seizure prediction algorithm on the shmac platform,” Master’s thesis, Norwegian Univerity of Science and Technology, 2014.

[25] MathWorks, “Solve, matlab.” ”`http://www.mathworks.se/help/symbolic/solve.html`”, June 2014.

.1 Appendix A: verification files

Listing 1: acc_coproc.S: Assembly file verifying the functionality of a 4 input 1 output coprocessor IFM

```
#include "amber_registers.h"

  .section .text
  .globl  main
main:
        @ ------------------------
        @ Straight forward test of ADD
        @ ADD 93, 0.07
        @ ------------------------
        ldr     r0, =0x00000004  @ acc_in 0
        ldr     r1, =0x00000008  @ acc_in 1
        ldr     r2, =0x00000010  @ acc_in 2
        ldr     r3, =0x00000020  @ acc_in 3
    ldr     r4, =0x00000004  @ options

  mcr 14, 0, r0, cr1, cr5    @ write acc_in 0
  mcr 14, 0, r1, cr2, cr5    @ write acc_in 1
  mcr 14, 0, r2, cr3, cr5    @ write acc_in 2
  mcr 14, 0, r3, cr4, cr5    @ write acc_in 3
  mcr 14, 0, r4, cr0, cr5    @ write options


    @ check for ready

  ldr     r0, =0x00000001
pollpoint:
    mrc 14, 0, r1, cr0, cr5    @ read poll port
  cmp r0, r1
  bne   pollpoint

  ldr r0, =0x0000003C
  mcr 14, 0, r1, cr1, cr5    @read results port

        @ Check that the output is correct
        cmp     r0, r1
        bne     testfail

    @passed
    b testpass


@ ------------------------------------------------
@ ------------------------------------------------

testfail:
        ldr     r11, AdrTestStatus
        str     r10, [r11]
        b       testfail

testpass:
        ldr     r11, AdrTestStatus
        mov     r10, #17
```

```
         str      r10, [r11]
         b        testpass



/* Write 17 to this address to generate a Test Passed message */
AdrTestStatus:              .word ADR_AMBER_TEST_STATUS
AdrTestBase   :             .word 0x001fffc0

/* sum of numbers 0 to 2047 inclusive */
MagicNumber1024   :          .word  523776
MagicNumber2048   :          .word 2096128

/*
    ========================================================================
     */
/*
    ========================================================================
     */
```

Listing 2: acc_wb.S: Assembly file used to record the Amber Cores wishbone timing

```
************************************************************/

#include "amber_registers.h"

   .section .text
   .globl  main
main:
     ldr       r0, =0x00000004   @ acc_in 0
         ldr       r1, =0x00000008   @ acc_in 1
         ldr       r2, =0x00000010   @ acc_in 2
         ldr       r3, =0x00000020   @ acc_in 3
     ldr       r4, =0x00000004   @ options

     ldr    r5, =0xfffe0001  @adr in 0
     ldr    r6, =0xfffe0002  @adr in 0
     ldr    r7, =0xfffe0003  @adr in 0
     ldr    r8, =0xfffe0004  @adr in 0
     ldr    r9, =0xfffe0000  @adr options

     str    r0, r5, #0
     str    r1, r6, #0
     str    r2, r7, #0
     str    r3, r8, #0
     str    r4, r9, #0

     ldr       r0, =0x00000000   @ acc_in 0
         ldr       r1, =0x00000000   @ acc_in 1
         ldr       r2, =0x00000000   @ acc_in 2
         ldr       r3, =0x00000000   @ acc_in 3
     ldr       r4, =0x00000000   @ options

     ldr    r0, r5, #0
     ldr    r1, r6, #0
     ldr    r2, r7, #0
     ldr    r3, r8, #0
```

```
    ldr   r4, r9, #0

        b       testpass

testfail:
        ldr     r11, AdrTestStatus
        str     r10, [r11]
        b       testfail

testpass:
        ldr     r11, AdrTestStatus
        mov     r10, #17
        str     r10, [r11]
        b       testpass


/* Write 17 to this address to generate a Test Passed message */
AdrTestStatus:  .word  ADR_AMBER_TEST_STATUS
AdrHiBootBase:  .word  ADR_HIBOOT_BASE

Data1:          .word  0x3
                .word  0x4
                .word  0x5
                .word  0x6
                .word  0x7
Data2:          .word  0x44332211
Data3:          .word  0x12345678

/*
    ==========================================================================
     */
/*
    ==========================================================================
     */
```

Listing 3: genSysTB.v

```
`timescale 1ns / 1ps

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

// Company:
// Engineer:
//
// Create Date:    15:09:07 05/16/2014
// Design Name:
// Module Name:    D:/AmberWrapper/genSysTB.v
// Project Name:   AmberWrapper
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module:
    amber_system_floop_acc_dummy_4_4
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
`define CYC 16.667 //ca 60 MHZ
`define MEMSIZE 8
module genSysTB;

  // Inputs
  reg i_clk;
  reg i_rst;
  reg i_irq;
  reg i_system_rdy;
  reg [127:0] i_wb_dat;
  reg i_wb_ack;
  reg i_wb_err;

  // Outputs
  wire [31:0] o_wb_adr;
  wire [15:0] o_wb_sel;
  wire o_wb_we;
  wire [127:0] o_wb_dat;
  wire o_wb_cyc;
  wire o_wb_stb;
  wire o_ld_excl;

  reg [127:0] wbMem[0:`MEMSIZE-1];
  reg [127:0] wbRec[0:`MEMSIZE-1];
  `include "wbMemTB.v"
  integer i;

  // Instantiate the Unit Under Test (UUT)
  amber_system_floop_acc_dummy_6_6 uut (
    .i_clk(i_clk),
    .i_rst(i_rst),
    .i_irq(i_irq),
    .i_system_rdy(i_system_rdy),
    .o_wb_adr(o_wb_adr),
    .o_wb_sel(o_wb_sel),
    .o_wb_we(o_wb_we),
    .i_wb_dat(i_wb_dat),
    .o_wb_dat(o_wb_dat),
    .o_wb_cyc(o_wb_cyc),
    .o_wb_stb(o_wb_stb),
    .i_wb_ack(i_wb_ack),
    .i_wb_err(i_wb_err),
    .o_ld_excl(o_ld_excl)
  );
  always
  begin
    //#16.667 clk = ~clk;
    #(0.5*`CYC) i_clk = ~i_clk;
  end

  //wb slave response
  always @(posedge o_wb_stb)begin
    #(0.5* `CYC);
    if(!o_wb_we)
    begin
      if(o_wb_adr[7:4]< `MEMSIZE && !(o_wb_adr[31:28] == 2))
      begin
        i_wb_dat = wbMem[o_wb_adr[7:4]];
        #(0.6* `CYC) i_wb_ack = 1;
      end
      else if(o_wb_adr == 32'h20000000)
      begin
        i_wb_dat = 128'h2;// slave(1) / master(2) select
        #(0.6* `CYC) i_wb_ack = 1;
      end
      else
      begin
        i_wb_dat = 128'h0;
        #(0.6* `CYC) i_wb_ack = 1;
      end
    end
    else
    begin
      if(o_wb_adr[7:4]< `MEMSIZE)
        wbRec[o_wb_adr[7:4]]  = o_wb_dat;
      #(0.6* `CYC) i_wb_ack = 1;
    end
  end
  always @(negedge o_wb_stb)begin
    i_wb_ack = 0;
  end

  initial begin
    // Initialize Inputs
    i_clk = 1;
    i_rst = 1;
    i_irq = 0;
    i_system_rdy = 0;
    i_wb_dat = 0;
    i_wb_ack = 0;
    i_wb_err = 0;

    // Wait 100 ns for global reset to finish
    #(2* `CYC) i_rst = 0;


  end

endmodule
```

Listing 4: wbMemTB.v

```
//end/*
initial
begin
wbMem[0] = 128'h00000004000000030000000200000001;
wbMem[1] = 128'h00000008000000070000000600000005;
wbMem[2] = 128'h0000000c0000000b0000000a00000009;
```

```
wbMem[3] = 128'h000000100000000f0000000e0000000d;
wbMem[4] = 128'h00000014000000130000001200000011;
wbMem[5] = 128'h00000018000000170000001600000015;
wbMem[6] = 128'h0000001c0000001b0000001a00000019;
wbMem[7] = 128'h000000200000001f0000001e0000001d;

wbRec[0] = 128'h00000000000000000000000000000000;
wbRec[1] = 128'h00000000000000000000000000000000;
wbRec[2] = 128'h00000000000000000000000000000000;
wbRec[3] = 128'h00000000000000000000000000000000;
wbRec[4] = 128'h00000000000000000000000000000000;
wbRec[5] = 128'h00000000000000000000000000000000;
wbRec[6] = 128'h00000000000000000000000000000000;
wbRec[7] = 128'h00000000000000000000000000000000;
end
```

Listing 5: a25_core_dummy.v

```
`define ACCADR   32'hfffe3000
`define INITADR  32'h20000000
`define MEMADR   32'h00000000
`define FLOPTADR   32'hfffe3000
`define FLMAADR   32'hfffe3004
`define FLINADR   32'hfffe3008
`define FLOUTADR   32'hfffe300C

`define SLOPTADR   32'hfffe37fc
`define SLINADR   32'hfffe3000
`define SLOUTADR   32'hfffe3800
`define SLPOLLADR     32'hfffe3ffc



`define FLOPT1   32'h00000002
`define FLMA1       32'h00000002
`define FLIN1       32'h80000000
`define FLOUT1   32'h90000000

`define FLALL    128'h90000040800000400000000100000002

`define SLOPT   32'h00000004
`define SLIN1   32'hbabebabe
`define SLIN2   32'hcafecafe
`define SLIN3   32'hdadedade
`define SLIN4   32'hcafebabe
`define SLIN5   32'h00001005
`define SLIN6   32'h00001006
`define SLIN7   32'h00001007
`define SLIN8   32'h00001008

`define IRQ    128'h00000020000000200000002000000020
module a25_core_dummy
(
 input              i_clk,

 input              i_rst,

 input           i_irq , // Interrupt request, active high
 input            i_firq , // Fast Interrupt request, active high

 input           i_system_rdy , // Amber is stalled when this is low

// Wishbone Master I/F
 output reg [31:0]  o_wb_adr ,
 output reg[15:0]  o_wb_sel ,
 output reg        o_wb_we ,
 input [127:0]  i_wb_dat ,
 output reg[127:0] o_wb_dat ,
 output reg        o_wb_cyc ,
 output reg        o_wb_stb ,
 input           i_wb_ack ,
 input           i_wb_err ,

// interface for exclusive op
 output          o_ld_excl
);


reg [31:0] state;


//statelist
  localparam  START = 32'h0,
          TSEL = 32'h1,
          M1 = 32'h2,
          M2 = 32'h3,
          M3 = 32'h4,
          M4 = 32'h5,
          M5 = 32'h6,
          M6 = 32'h7,

          IRQSET = 32'h2000,
          S1 = 32'h1001,
          S2 = 32'h1002,
          S3 = 32'h1003,
          S4 = 32'h1004,
          S5 = 32'h1005,
          S6 = 32'h1006,
          S7 = 32'h1007,
          S8 = 32'h1008,
          S9 = 32'h1009,
          Sa = 32'h100a;



always @(posedge i_clk)
begin
  if(i_rst)
  begin
    state = START;
    o_wb_adr <= 'h0;
    o_wb_sel <= 'h0;
    o_wb_we <= 'h0;
    o_wb_cyc <= 'h0;
    o_wb_stb <= 'h0;
    o_wb_dat <= 'h0;
  end
  else
  begin
    case(state)
      START:
      begin
        o_wb_adr <= 32'hfffe2008;
        o_wb_sel <= 16'h0f00;
        o_wb_we  <= 1;
        o_wb_cyc  <= 1;
        o_wb_stb  <= 1;
        o_wb_dat <= `IRQ;
        state = IRQSET;

      end
      IRQSET:
      begin
        if(i_wb_ack)
        begin
          o_wb_adr <= `INITADR;
          o_wb_sel <= 16'h000f;
          o_wb_we  <= 0;
          o_wb_cyc  <= 1;
          o_wb_stb  <= 1;
          state = TSEL;

        end
      end
      TSEL:
      begin
        if(i_wb_ack)
        begin
          if(i_wb_dat[1])
          begin
            o_wb_adr <= `FLMAADR;
            o_wb_sel <= 16'h00f0;
            o_wb_we  <= 1;
            o_wb_cyc  <= 1;
            o_wb_stb  <= 1;
            o_wb_dat <= {`FLMA1, `FLMA1, `FLMA1, `FLMA1};
            state = M1;
          end
          else if(i_wb_dat[3:0] == 4'h1)
          begin
            o_wb_adr <= `SLINADR + 'h0;
            o_wb_sel <= 'h000f;
            o_wb_we <= 'h1;
            o_wb_cyc <= 'h1;
            o_wb_stb <= 'h1;
            o_wb_dat <= {`SLIN1, `SLIN1, `SLIN1, `SLIN1};
            state = S1;
          end
          else
          begin
            o_wb_adr <= 'h0;
```

```
                o_wb_sel <= 'h0;
                o_wb_we <= 'h0;
                o_wb_cyc <= 'h0;
                o_wb_stb <= 'h0;
                state = START;
              end
            end
          end
          M1:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `FLINADR;
              o_wb_sel <= 16'h0f00;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`FLIN1, `FLIN1, `FLIN1, `FLIN1};
              state = M2;
            end
          end
          M2:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `FLOUTADR;
              o_wb_sel <= 16'hf000;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`FLOUT1, `FLOUT1, `FLOUT1, `FLOUT1};
              state = M3;
            end
          end
          M3:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `FLOPTADR;
              o_wb_sel <= 16'h000f;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`FLOPT1, `FLOPT1, `FLOPT1, `FLOPT1};
              state = M4;
            end
          end
          M4:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= 0;
              o_wb_sel <= 0;
              o_wb_we  <= 0;
              o_wb_cyc  <= 0;
              o_wb_stb  <= 0;
              o_wb_dat <= 0;
            end

            if(i_irq)
            begin // Set all at once
              o_wb_adr <= `FLOPTADR;
              o_wb_sel <= 16'hffff;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= `FLALL;
              state = M5;
            end

          end
          M5:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= 0;
              o_wb_sel <= 0;
              o_wb_we  <= 0;
              o_wb_cyc  <= 0;
              o_wb_stb  <= 0;
              o_wb_dat <= 0;
            end
            if(i_irq)
              state = M6;
          end
          S1:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `SLINADR + 'h4;
              o_wb_sel <= 16'h00f0;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`SLIN2, `SLIN2, `SLIN2, `SLIN2};
              state = S2;
            end
          end
          S2:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `SLINADR + 'h8;
              o_wb_sel <= 16'h0f00;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`SLIN3, `SLIN3, `SLIN3, `SLIN3};
              state = S3;
            end
          end
          S3:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `SLINADR + 'hC;
              o_wb_sel <= 16'hf000;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`SLIN4, `SLIN4, `SLIN4, `SLIN4};
              state = S4;
            end
          end
          S4:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `SLOPTADR;
              o_wb_sel <= 16'hf000;
              o_wb_we  <= 1;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              o_wb_dat <= {`SLOPT, `SLOPT, `SLOPT, `SLOPT};
              state = S5;
            end
          end
          S5:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= 0;
              o_wb_sel <= 0;
              o_wb_we  <= 0;
              o_wb_cyc  <= 0;
              o_wb_stb  <= 0;
              o_wb_dat <= 0;
            end
            if(i_irq)
            begin//read all
              o_wb_adr <= `SLOUTADR;
              o_wb_sel <= 16'hffff;
              o_wb_we  <= 0;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              state = S6;
            end
          end
          S6:
          begin
            if(i_wb_ack)
            begin
              o_wb_adr <= `SLOUTADR;
              o_wb_sel <= 16'h000f;
              o_wb_we  <= 0;
              o_wb_cyc  <= 1;
              o_wb_stb  <= 1;
              state = S7;
            end
          end
          S7:
```

```
      begin
        if(i_wb_ack)
        begin
          o_wb_adr <= `SLOUTADR + 'h4;
          o_wb_sel <= 16'h00f0;
          o_wb_we  <= 0;
          o_wb_cyc  <= 1;
          o_wb_stb  <= 1;
          state = S8;
        end
      end
      S8:
      begin
        if(i_wb_ack)
        begin
          o_wb_adr <= `SLOUTADR + 'h8;
          o_wb_sel <= 16'h0f00;
          o_wb_we  <= 0;
          o_wb_cyc  <= 1;
          o_wb_stb  <= 1;
          state = S9;
        end
      end
      S9:
      begin
        if(i_wb_ack)
        begin
          o_wb_adr <= `SLOUTADR + 'hC;
          o_wb_sel <= 16'hf000;
          o_wb_we  <= 0;
          o_wb_cyc  <= 1;
          o_wb_stb  <= 1;
          state = Sa;
        end
      end
      Sa:
      begin
        if(i_wb_ack)
        begin
          o_wb_adr <= 0;
          o_wb_sel <= 0;
          o_wb_we  <= 0;
          o_wb_cyc  <= 0;
          o_wb_stb  <= 0;
          o_wb_dat <= 0;
        end
      end
    endcase
  end
end




endmodule
```

Listing 6: accelerator_dummy.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:04:21 04/15/2014
// Design Name:
// Module Name:    acc_dummy
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module acc_dummy_6_6(
   input      i_clk,
   input      i_rst,
   input      i_start,
   input [31:0]   i_opt,
   output reg     o_rdy,
    input [31:0] acc_in_0,
    input [31:0] acc_in_1,
    input [31:0] acc_in_2,
   input [31:0] acc_in_3,
   input [31:0] acc_in_4,
   input [31:0] acc_in_5,
    output reg [31:0] acc_out_0,
   output reg [31:0] acc_out_1,
   output reg [31:0] acc_out_2,
   output reg [31:0] acc_out_3,
   output reg [31:0] acc_out_4,
   output reg [31:0] acc_out_5
    );

  reg [31:0] counter;
  reg [31:0] input_buffer [0:5];
  reg start_state;
  always @(posedge i_clk or posedge i_rst)
  begin
    if(i_rst)
    begin
      counter = 'h0;

      acc_out_0 = 'h0;
      acc_out_1 = 'h0;
      acc_out_2 = 'h0;
      acc_out_3 = 'h0;
      acc_out_4 = 'h0;
      acc_out_5 = 'h0;
      o_rdy = 'h0;
      start_state = 'h0;
    end
    else if(start_state) //TODO: warning, possible bug, start state more
        important then start_i
    begin
      if (counter > 0)
        begin
          counter = counter - 1;
          o_rdy = 'h0;
        end
      else
      begin
        acc_out_0 = input_buffer[0];
        acc_out_1 = input_buffer[1];
        acc_out_2 = input_buffer[2];
        acc_out_3 = input_buffer[3];
        acc_out_4 = input_buffer[4];
        acc_out_5 = input_buffer[5];
        counter = 0;
        o_rdy = 1;
        start_state = 0;
      end
    end
    else if(i_start)
    begin
      counter = i_opt;
      input_buffer[0] = acc_in_0;
      input_buffer[1] = acc_in_1;
      input_buffer[2] = acc_in_2;
      input_buffer[3] = acc_in_3;
      input_buffer[4] = acc_in_4;
      input_buffer[5] = acc_in_5;
      o_rdy = 0;
      start_state = 1;
    end
    else
      o_rdy = 0;
  end

endmodule

module acc_dummy_5_3(
   input        i_clk,
   input        i_rst,
   input        i_start,
   input [31:0]    i_opt,
   output reg      o_rdy,
    input [31:0] acc_in_0,
    input [31:0] acc_in_1,
    input [31:0] acc_in_2,
   input [31:0] acc_in_3,
   input [31:0] acc_in_4,
    output reg [31:0] acc_out_0,
   output reg [31:0] acc_out_1,
   output reg [31:0] acc_out_2
    );

  reg [31:0] counter;
  reg [31:0] input_buffer [0:4];
  reg start_state;
  always @(posedge i_clk or posedge i_rst)
  begin
    if(i_rst)
    begin
      counter = 'h0;

      acc_out_0 = 'h0;
      acc_out_1 = 'h0;
      acc_out_2 = 'h0;
      o_rdy = 'h0;
      start_state = 'h0;
    end
    else if(start_state) //TODO: warning, possible bug, start state more
        important then start_i
    begin
      if (counter > 0)
        begin
          counter = counter - 1;
          o_rdy = 'h0;
        end
      else
      begin
        acc_out_0 = input_buffer[0] + input_buffer[3];
        acc_out_1 = input_buffer[1] + input_buffer[4];
        acc_out_2 = input_buffer[2] + input_buffer[1];
        counter = 0;
        o_rdy = 1;
        start_state = 0;
      end
    end
    else if(i_start)
    begin
      counter = i_opt;
      input_buffer[0] = acc_in_0;
      input_buffer[1] = acc_in_1;
      input_buffer[2] = acc_in_2;
      input_buffer[3] = acc_in_3;
      input_buffer[4] = acc_in_4;
      o_rdy = 0;
      start_state = 1;
    end
    else
      o_rdy = 0;
  end

endmodule

module acc_dummyV2_6_6(
   input       i_clk,
   input       i_rst,
   input       i_start,
   input [31:0]    i_opt,
   output reg      o_rdy,
    input [31:0] acc_in_0 ,
    input [31:0] acc_in_1 ,
    input [31:0] acc_in_2 ,
   input [31:0] acc_in_3 ,
   input [31:0] acc_in_4 ,
   input [31:0] acc_in_5 ,
    output reg [31:0] acc_out_0 ,
   output reg [31:0] acc_out_1 ,
   output reg [31:0] acc_out_2 ,
   output reg [31:0] acc_out_3 ,
   output reg [31:0] acc_out_4 ,
   output reg [31:0] acc_out_5
    );

  reg [31:0] counter;
  reg [31:0] input_buffer [0:5];
  reg start_state;
  always @(posedge i_clk or posedge i_rst)
  begin
    if(i_rst)
    begin
      counter = 'h0;

      acc_out_0 = 'h0;
      acc_out_1 = 'h0;
      acc_out_2 = 'h0;
      acc_out_3 = 'h0;
      acc_out_4 = 'h0;
      acc_out_5 = 'h0;
      o_rdy = 'h0;
      start_state = 'h0;
    end
    else if(start_state) //TODO: warning, possible bug, start state more
        important then start_i
    begin
      if (counter > 0)
        begin
          counter = counter - 1;
          o_rdy = 'h0;
        end
      else
      begin
        acc_out_0 = input_buffer[0];
        acc_out_1 = input_buffer[1];
        acc_out_2 = input_buffer[2];
        acc_out_3 = input_buffer[3];
        acc_out_4 = input_buffer[4];
        acc_out_5 = input_buffer[5];
        counter = 0;
        o_rdy = 1;
        start_state = 0;
      end
    end
    else if(i_start)
    begin
      counter = i_opt;
      input_buffer[0] = acc_in_0 + acc_in_1;
      input_buffer[1] = acc_in_1 + acc_in_2;
      input_buffer[2] = acc_in_2 - acc_in_3;
      input_buffer[3] = acc_in_3 + acc_in_4;
      input_buffer[4] = acc_in_4 - acc_in_5;
      input_buffer[5] = acc_in_5 + acc_in_0;
      o_rdy = 0;
      start_state = 1;
    end
    else
      o_rdy = 0;
  end

endmodule
module acc_dummy_4_4(
   input       i_clk,
   input       i_rst,
   input       i_start,
   input [31:0]    i_opt,
   output reg      o_rdy,

  input [31:0] acc_in_0,
    input [31:0] acc_in_1,
    input [31:0] acc_in_2,
   input [31:0] acc_in_3,

   output [31:0] acc_out_0,
   output [31:0] acc_out_1,
   output [31:0] acc_out_2,
   output [31:0] acc_out_3
   );
   assign acc_out_0 = acc_in_0 & i_opt;
   assign acc_out_1 = acc_in_1 & i_opt;
   assign acc_out_2 = acc_in_2 & i_opt;
   assign acc_out_3 = acc_in_3 & i_opt;

   always @(posedge i_clk)
    o_rdy = !i_start;
endmodule


module acc_dummyV2_4_4(
   input       i_clk,
   input       i_rst,
   input       i_start,
   input [31:0]    i_opt,
   output reg      o_rdy,
    input [31:0] acc_in_0,
    input [31:0] acc_in_1,
    input [31:0] acc_in_2,
   input [31:0] acc_in_3,

   output reg [31:0] acc_out_0,
   output reg [31:0] acc_out_1,
   output reg [31:0] acc_out_2,
   output reg [31:0] acc_out_3

   );

  reg [31:0] counter;
  reg [31:0] input_buffer [0:5];
  reg start_state;
  always @(posedge i_clk or posedge i_rst)
  begin
    if(i_rst)
    begin
      counter = 'h0;

      acc_out_0 = 'h0;
      acc_out_1 = 'h0;
      acc_out_2 = 'h0;
      acc_out_3 = 'h0;
      o_rdy = 'h0;
      start_state = 'h0;
    end
    else if(start_state) //TODO: warning, possible bug, start state more
        important then start_i
    begin
      if (counter > 0)
        begin
          counter = counter - 1;
          o_rdy = 'h0;
        end
      else
      begin
        acc_out_0 = input_buffer[0] + input_buffer[1];
        acc_out_1 = input_buffer[1] + input_buffer[2];
        acc_out_2 = input_buffer[2] - input_buffer[1];
        acc_out_3 = input_buffer[3] - input_buffer[0];
        counter = 0;
        o_rdy = 1;
        start_state = 0;
      end
    end
    else if(i_start)
    begin
      counter = i_opt;
      input_buffer[0] = acc_in_0;
      input_buffer[1] = acc_in_1;
      input_buffer[2] = acc_in_2;
      input_buffer[3] = acc_in_3;
      o_rdy = 0;
      start_state = 1;
    end
    else
      o_rdy = 0;
  end

endmodule

module acc_dummy_1_4(
   input       i_clk,
   input       i_rst,
   input       i_start,
   input [31:0]   i_opt,
  output reg     o_rdy,
   input [31:0] acc_in_0,

  output reg [31:0] acc_out_0,
  output reg [31:0] acc_out_1,
  output reg [31:0] acc_out_2,
  output reg [31:0] acc_out_3

  );

reg [31:0] counter;
reg [31:0] input_buffer [0:5];
reg start_state;
always @(posedge i_clk or posedge i_rst)
begin
  if(i_rst)
  begin
    counter = 'h0;

    acc_out_0 = 'h0;
    acc_out_1 = 'h0;
    acc_out_2 = 'h0;
    acc_out_3 = 'h0;
    o_rdy = 'h0;
    start_state = 'h0;
  end
  else if(start_state) //TODO: warning, possible bug, start state more
      important then start_i
  begin
    if (counter > 0)
      begin
        counter = counter - 1;
        o_rdy = 'h0;
      end
    else
    begin
      acc_out_0 = input_buffer[0];
      acc_out_1 = input_buffer[1];
      acc_out_2 = input_buffer[2];
      acc_out_3 = input_buffer[3];
      counter = 0;
      o_rdy = 1;
      start_state = 0;
    end
  end
  else if(i_start)
  begin
    counter = i_opt;
    input_buffer[0] = acc_in_0 * 2;
    input_buffer[1] = acc_in_0 + 8;
    input_buffer[2] = acc_in_0 - 4;
    input_buffer[3] = acc_in_0;
    o_rdy = 0;
    start_state = 1;
  end
  else
    o_rdy = 0;
  end

endmodule


module acc_dummyV3_4_1(
   input      i_clk ,
   input      i_rst ,
   input      i_start ,
   input [31:0]   i_opt ,
   output reg     o_rdy ,
    input [31:0] acc_in_0 ,
    input [31:0] acc_in_1 ,
    input [31:0] acc_in_2 ,
   input [31:0] acc_in_3 ,
    output reg [31:0] acc_out_0
    );

  reg [31:0] counter;
  reg [31:0] input_buffer [0:3];
  reg start_state;
  always @(posedge i_clk or posedge i_rst)
  begin
    if(i_rst)
    begin
      counter = 'h0;

      acc_out_0 <= 'h0;
      o_rdy <= 'h0;
      start_state = 'h0;
    end
    else if(start_state)
    begin
      if (counter > 0)
        begin
          counter = counter - 1;
          o_rdy <= 'h0;
        end
      else
      begin
        acc_out_0 <= input_buffer[0] + input_buffer[1] + input_buffer[2] +
             input_buffer[3];
        counter = i_opt;
        o_rdy <= 1;
      end
    end
    else if(i_start)
    begin
      counter = i_opt;
      input_buffer[0] = acc_in_0;
      input_buffer[1] = acc_in_1;
      input_buffer[2] = acc_in_2;
      input_buffer[3] = acc_in_3;
      o_rdy <= 0;
      start_state = 1;
    end
    else
```

```
        o_rdy <= 0;
    end

endmodule



module acc_dummyV2_4_1(
      input       i_clk ,
      input       i_rst ,
      input       i_start ,
      input [31:0]    i_opt ,
      output reg      o_rdy ,
        input [31:0] acc_in_0 ,
        input [31:0] acc_in_1 ,
        input [31:0] acc_in_2 ,
      input [31:0] acc_in_3 ,
        output reg [31:0] acc_out_0
        );

    reg [31:0] counter;
    reg [31:0] counter2;
    reg [31:0] input_buffer [0:3];
    reg start_state;
    always @(posedge i_clk or posedge i_rst)
    begin
      if(i_rst)
      begin
        counter = 'h0;
        counter2 = 'h0;
        acc_out_0 = 'h0;
        o_rdy <= 'h0;
        start_state = 'h0;
      end
      else if(start_state)
      begin
        if (counter < i_opt )
          begin
            counter = counter +1;
            counter2 = counter2 +1;
            o_rdy <= 'h0;
          end
        else
        begin
          acc_out_0 = counter2;
          counter = 0;
          o_rdy <= 1;
        end
      end
      else if(i_start)
      begin
        counter = 0;
        counter2 = 'h0;
        input_buffer[0] = acc_in_0;
        input_buffer[1] = acc_in_1;
        input_buffer[2] = acc_in_2;
        input_buffer[3] = acc_in_3;
        o_rdy <= 0;
        start_state = 1;
      end
      else
        o_rdy <= 0;
    end

endmodule

module acc_dummy_2_2(
      input       i_clk,
      input       i_rst,
      input       i_start,
      input [31:0]    i_opt,
      output reg      o_rdy,
       input [31:0] acc_in_0,
       input [31:0] acc_in_1,
       output reg [31:0] acc_out_0,
      output reg [31:0] acc_out_1
      );

    reg [31:0] counter;
    reg [31:0] input_buffer [0:5];
    reg start_state;
    always @(posedge i_clk or posedge i_rst)
    begin
      if(i_rst)
      begin
        counter = 'h0;

        acc_out_0 = 'h0;
        acc_out_1 = 'h0;
        o_rdy = 'h0;
        start_state = 'h0;
      end
      else if(start_state) //TODO: warning, possible bug, start state more
          important then start_i
      begin
        if (counter > 0)
          begin
            counter = counter - 1;
            o_rdy = 'h0;
          end
        else
        begin
          acc_out_0 = input_buffer[0];
          acc_out_1 = input_buffer[1];
          counter = 0;
          o_rdy = 1;
          start_state = 0;
        end
      end
      else if(i_start)
      begin
        counter = i_opt;
        input_buffer[0] = acc_in_0;
        input_buffer[1] = acc_in_1;
        o_rdy = 0;
        start_state = 1;
      end
      else
        o_rdy = 0;
    end

endmodule

module acc_dummy_1_1(
      input        i_clk,
      input      i_rst,
      input      i_start,
      input [31:0]  i_opt,
      output reg      o_rdy,
      input [31:0] acc_in_0,
      output reg [31:0] acc_out_0
      );

      always @(posedge i_clk)
      begin
        if(i_rst)
        begin
          acc_out_0 = 'h0;
          o_rdy = 'h0;
        end
        else
        begin
          if(o_rdy)
            o_rdy = 0;
          if(i_start)
          begin
            acc_out_0 = acc_in_0;
            o_rdy = 1;
          end
        end
      end

endmodule

module acc_dummy_3_1(
      input        i_clk,
      input      i_rst,
      input      i_start,
      input [31:0]  i_opt,
      output reg      o_rdy,
      input [31:0] acc_in_0,
      input [31:0] acc_in_1,
      input [31:0] acc_in_2,
      output reg [31:0] acc_out_0
      );

      always @(posedge i_clk)
      begin
        if(i_rst)
        begin
          acc_out_0 = 'h0;
```

```
          o_rdy = 'h0;
        end
        else
        begin
          if(o_rdy)
            o_rdy = 0;
          if(i_start)
          begin
            acc_out_0 = acc_in_0;
            o_rdy = 1;
          end
        end
      end

endmodule
```

.2 Appendix B: Tile Generator and Template Files

Listing 7: tileGenerator.py

```
import sys, getopt, os


def main(argv):
  ifType = ''
  moduleName = ''
  inputsN = 0
  outputsM = 0
  amber25Path = "../units/amber/hw/vlog/amber25/"
  amberTilePath = "../tiles/amber_tile/"
  TilesPath = "../tiles/"
  outputPath = "../tiles/amber_tile_"


  try:
    opts, args = getopt.getopt(argv,'ht:m:i:o:')
  except getopt.GetoptError:
    print("ERROR, lacking options, Script exited.")
    print('tileGenerator.py -t <interfacetype(coproc, slave, floop)> -m <
        Module name> -i <nr of input ports> -o <nr of output ports>')
    sys.exit(2)

  if len(opts) < 4:
    print("ERROR, lacking options, Script exited.")
    print('tileGenerator.py -t <interfacetype(coproc, slave, floop)> -m <
        Module name> -i <nr of input ports> -o <nr of output ports>')
    sys.exit(2)

  for opt, arg in opts:
    if opt == '-h':
      print('tileGenerator.py -t <interfacetype(coproc, slave, floop)> -m
          <Module name> -i <nr of input ports> -o <nr of output ports>')
      sys.exit()
    elif opt =="-t":
      ifType = arg
    elif opt =="-m":
      moduleName = arg
    elif opt == "-i":
      inputsN = int(arg)
    elif opt == "-o":
      outputsM = int(arg)
  print( 'Interface type is "', ifType, '"')
  print( 'Accelerator module is "', moduleName, '"')
  print( 'number of input ports is:', inputsN)
  print( 'number of output ports is:', outputsM)
  #creating tile

  ####################################################
  ###        coproc IF generate          ###
  ####################################################

  if ifType == "coproc":
    outputPath = outputPath + "coprocIFM_" + moduleName + "/"
    if not os.path.exists(outputPath):
      os.makedirs(outputPath)

    if not os.path.exists(outputPath + "/a25_placeholder/"):
      os.makedirs(outputPath + "/a25_placeholder/")

    #creating tile files
    #tileregs.v
    sourceFile = open(amberTilePath + "tile_regs.v", "r")
    oFile = open(outputPath + "tile_regs.v", "w")

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_wrapper.v
    sourceFile = open(amberTilePath + "amber_wrapper.v", "r")
    oFile = open(outputPath + "amber_wrapper_" + moduleName +".v", "w")

    #find //ACCTAGMOD
    for line in sourceFile:
      if "//ACCTAGMOD" in line:
        break
      oFile.write(line)

    oFile.write("module amber_wrapper_" + moduleName + "\n")

    #find //ACCTAGMODDONE
    for line in sourceFile:
      if "//ACCTAGMODDONE" in line:
        break

    #find //ACCTAGSYS
    for line in sourceFile:
      if "//ACCTAGSYS" in line:
        break
      oFile.write(line)

    oFile.write("\tamber_system_" + moduleName + "\n")

    #find //ACCTAGSYSDONE
    for line in sourceFile:
      if "//ACCTAGSYSDONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_tile.vhd
    sourceFile = open(amberTilePath + "amber_tile.vhd", "r")
    oFile = open(outputPath + "amber_tile_" + moduleName +".vhd", "w")

    #find --ACCTAGENT
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break
      oFile.write(line)

    oFile.write("entity amber_tile_" + moduleName + " is\n")

    #find --ACCTAGENTDONE
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break

    #find --ACCTAGENTEND
    for line in sourceFile:
      if "--ACCTAGENTEND" in line:
        break
      oFile.write(line)

    oFile.write("end amber_tile_" + moduleName + ";\n\n")
    oFile.write("architecture rtl of amber_tile_" + moduleName + " is\n\n
        ")

    #find --ACCTAGENTENDDONE
    for line in sourceFile:
      if "--ACCTAGENTENDDONE" in line:
        break

    #find --ACCTAGWRA
    for line in sourceFile:
      if "--ACCTAGWRA" in line:
        break
      oFile.write(line)

    oFile.write("amber_u: amber_wrapper_" + moduleName + "\n")

    #find --ACCTAGWRADONE
    for line in sourceFile:
      if "--ACCTAGWRADONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_system.v
    sourceFile = open(amberTilePath + "amber_system.v", "r")
    oFile = open(outputPath + "amber_system_" + moduleName +".v", "w")

    #find //ACCTAGSYSINST
    for line in sourceFile:
      if "//ACCTAGSYSINST" in line:
        break
      oFile.write(line)

    oFile.write("module amber_system_" + moduleName + "\n")

    #find //ACCTAGSYSINSTDONE
    for line in sourceFile:
      if "//ACCTAGSYSINSTDONE" in line:
        break

    #find //ACCTAGCORE
    for line in sourceFile:
      if "//ACCTAGCORE" in line:
        break
      oFile.write(line)

    oFile.write("a25_core_" + moduleName + "\n")

    #find //ACCTAGCOREDONE
    for line in sourceFile:
      if "//ACCTAGCOREDONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)
    #creating coprocessor file

    sourceFile = open(amber25Path + "a25_coprocessor.v", "r")
    templateFile = open("templates/coproc_TEMP.v", "r")
    oFile = open(outputPath + "a25_placeholder/a25_coprocessor_" +
        moduleName +".v", "w")
    wireOutName = "wire [31:0] acc_out_"


    #find //ACCTAGDEF
    for line in sourceFile:
      if "//ACCTAGDEF" in line:
        break
      oFile.write(line)

    #ACCTAGDEF found
    oFile.write("// Script generated defines \n")
    oFile.write("`define ACC_INPUTS\t\t" + str(inputsN) + "\n")
    oFile.write("`define ACC_OUTPUTS\t\t" + str(outputsM) + "\n\n")

    oFile.write("module a25_coprocessor_" + moduleName + "\n")
    #skip until next tag, no write
    #find //ACCTAGMOD
    for line in sourceFile:
      if "//ACCTAGMOD" in line:
        break
    #find //ACCTAGINST
    for line in sourceFile:
      if "//ACCTAGINST" in line:
        break
      oFile.write(line)

    #ACCTAGINST found
    #Moving to template file
    #find #0
    for line in templateFile:
      if "#0" in line:
        break
      oFile.write(line)

    # #0 found
    for i in range(outputsM):
      oFile.write("\t"+ wireOutName + str(i) +";\n")

    #find #1
    for line in templateFile:
      if "#1" in line:
        break
      oFile.write(line)

    # #1 found
    oFile.write("\t" + moduleName + " accelerator(\n")

    #find #2
    for line in templateFile:
      if "#2" in line:
        break
      oFile.write(line)

    # #2 found
    for i in range(inputsN):
      oFile.write("\t.acc_in_" + str(i) +" (acc_in["+ str(i) +"]),\n")
    for i in range(outputsM):
      oFile.write("\t.acc_out_" + str(i) +" (acc_out_" + str(i))
      if i == (outputsM - 1):
        oFile.write(")\n")
      else:
        oFile.write("),\n")

    #find #3
    for line in templateFile:
      if "#3" in line:
        break
      oFile.write(line)

    # #3 found, end of section
    #moving back to sourcefile
    #find //ACCTAGREGW
    for line in sourceFile:
      if "//ACCTAGREGW" in line:
        break
      oFile.write(line)

    #ACCTAGREGW found
    #Moving to template file
    #find #4
    for line in templateFile:
      if "#4" in line:
        break
      oFile.write(line)

    # #4 found
    for i in range(outputsM):
      oFile.write("\tacc_out[" + str(i) +"] <= acc_out_" + str(i) + ";\n")

    for line in templateFile:
      oFile.write(line)

    sourceFile.close()
    templateFile.close()
    oFile.close()

    # Generate compatible amber_core

    sourceFile = open(amber25Path + "a25_core.v", "r")
    oFile = open(outputPath + "a25_placeholder/a25_core_" + moduleName +".
        v", "w")

    #find //ACCTAGCOPROC
    for line in sourceFile:
      if "//ACCTAGCOPROC" in line:
        break
      oFile.write(line)

    oFile.write("a25_coprocessor_" + moduleName +" u_coprocessor ( \n")

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    print("Done!")
    print("Your verilog files can be found in the output folder")

  ####################################################
  ###        slave IF generate        ###
  ####################################################

  elif ifType == "slave":
    outputPath = outputPath + "slaveIFM_" + moduleName + "/"
    if not os.path.exists(outputPath):
      os.makedirs(outputPath)

    if not os.path.exists(outputPath + "/a25_placeholder/"):
      os.makedirs(outputPath + "/a25_placeholder/")

    #creating tile files
    #tileregs.v
    sourceFile = open(amberTilePath + "tile_regs.v", "r")
    oFile = open(outputPath + "tile_regs.v", "w")

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()


    #amber_wrapper.v
    sourceFile = open(amberTilePath + "amber_wrapper.v", "r")
    oFile = open(outputPath + "amber_wrapper_" + moduleName +".v", "w")

    #find //ACCTAGMOD
    for line in sourceFile:
      if "//ACCTAGMOD" in line:
        break
      oFile.write(line)

    oFile.write("module amber_wrapper_" + moduleName + "\n")

    #find //ACCTAGMODDONE
    for line in sourceFile:
      if "//ACCTAGMODDONE" in line:
        break

    #find //ACCTAGSYS
    for line in sourceFile:
      if "//ACCTAGSYS" in line:
        break
      oFile.write(line)

    oFile.write("\tamber_system_" + moduleName + "\n")

    #find //ACCTAGSYSDONE
    for line in sourceFile:
      if "//ACCTAGSYSDONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_tile.vhd
    sourceFile = open(amberTilePath + "amber_tile.vhd", "r")
    oFile = open(outputPath + "amber_tile_" + moduleName +".vhd", "w")

    #find --ACCTAGENT
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break
      oFile.write(line)

    oFile.write("entity amber_tile_" + moduleName + " is\n")

    #find --ACCTAGENTDONE
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break

    #find --ACCTAGENTEND
    for line in sourceFile:
      if "--ACCTAGENTEND" in line:
        break
        oFile.write(line)

    oFile.write("end amber_tile_" + moduleName + ";\n\n")
    oFile.write("architecture rtl of amber_tile_" + moduleName + " is\n\n
        ")

    #find --ACCTAGENTENDDONE
    for line in sourceFile:
        if "--ACCTAGENTENDDONE" in line:
            break

    #find --ACCTAGWRA
    for line in sourceFile:
        if "--ACCTAGWRA" in line:
            break
        oFile.write(line)

    oFile.write("amber_u: amber_wrapper_" + moduleName + "\n")

    #find --ACCTAGWRADONE
    for line in sourceFile:
        if "--ACCTAGWRADONE" in line:
            break

    for line in sourceFile:
        oFile.write(line)

    sourceFile.close()
    oFile.close()


    ### Generate slave IFM

    sourceFile = open(amberTilePath + "amber_system.v", "r")
    templateFile = open("templates/slave_TEMP.v", "r")
    oFile = open(outputPath + "slave_" + moduleName +".v", "w")
    wireOutName = "wire [31:0] acc_out_"



    #find #0
    for line in templateFile:
        if "#0" in line:
            break
        oFile.write(line)

    #0 found
    oFile.write ("//Script generated defines \n")
    oFile.write("`define ACC_INPUTS\t\t" + str(inputsN) + "\n")
    oFile.write("`define ACC_OUTPUTS\t\t" + str(outputsM) + "\n\n")

    oFile.write("module IFM_slave_" + moduleName + "\n")

    #find #1
    for line in templateFile:
        if "#1" in line:
            break
        oFile.write(line)

    #1 found
    for i in range(outputsM):
        oFile.write("\t"+ wireOutName + str(i) +";\n")
    oFile.write("\n")
    oFile.write("\t" + moduleName + " accelerator(\n")

    #find #2
    for line in templateFile:
        if "#2" in line:
            break
        oFile.write(line)

    #2 found
    for i in range(inputsN):
        oFile.write("\t.acc_in_" + str(i) +" (acc_in["+ str(i) +"]),\n")
    for i in range(outputsM):
        oFile.write("\t.acc_out_" + str(i) +" (acc_out_" + str(i))
        if i == (outputsM - 1):
            oFile.write(")\n")
        else:
            oFile.write("),\n")

    #find #3
    for line in templateFile:
        if "#3" in line:
            break
        oFile.write(line)

    #3 found
    for i in range(outputsM):
        oFile.write("\t\tacc_out[" + str(i) +"] <= acc_out_" + str(i) + ";\n
            ")

    for line in templateFile:
        oFile.write(line)

    oFile.close()
    templateFile.close()
    ### slave IFM generation completed

    #generate new amber_system file
    templateFile = open("templates/slave_sys_TEMP.v", "r")
    oFile = open(outputPath + "amber_system_slave_" + moduleName +".v", "w
        ")

    #find //ACCTAGSYSINST
    for line in sourceFile:
        if "//ACCTAGSYSINST" in line:
            break
        oFile.write(line)

    #find //ACCTAGSYSINSTDONE
    for line in sourceFile:
        if "//ACCTAGSYSINSTDONE" in line:
            break
```

```

  oFile.write("module amber_system_slave_" + moduleName + "\n")

  #find //ACCTAGINST
  for line in sourceFile:
    if "//ACCTAGINST" in line:
      break
    oFile.write(line)

  #move past lines not to be written
  #find //ACCTAGINSTDONE
  for line in sourceFile:
    if "//ACCTAGINSTDONE" in line:
      break

  #move to template file
  #find #0
  for line in templateFile:
    if "#0" in line:
      break
    oFile.write(line)

  #0 found
  oFile.write("\t" +"IFM_slave_"+ moduleName + " u_acc_if_slave (\n")

  #find #1
  for line in templateFile:
    if "#1" in line:
      break
    oFile.write(line)

  #1 found
  #move to source file
  #find //ACCTAGSTBRST
  for line in sourceFile:
    if "//ACCTAGSTBRST" in line:
      break
    oFile.write(line)

  oFile.write("       wb_stb_acc = 0;\n")

  #find //ACCTAGSTB
  for line in sourceFile:
    if "//ACCTAGSTB" in line:
      break
    oFile.write(line)

  for line in templateFile:
    oFile.write(line)

  for line in sourceFile:
    oFile.write(line)

  sourceFile.close()
  templateFile.close()
  oFile.close()

```

```
    print("Done!")
    print("Your verilog files can be found in the output folder")
  ####################################################
  ###       master IF generate          ###
  ####################################################

  elif ifType == "floop":
    outputPath = outputPath + "floopIFM_" + moduleName + "/"
    if not os.path.exists(outputPath):
      os.makedirs(outputPath)

    if not os.path.exists(outputPath + "/a25_placeholder/"):
      os.makedirs(outputPath + "/a25_placeholder/")

    #creating tile files
    #tileregs.v
    sourceFile = open(amberTilePath + "tile_regs.v", "r")
    oFile = open(outputPath + "tile_regs.v", "w")

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_wrapper.v
    sourceFile = open(amberTilePath + "amber_wrapper.v", "r")
    oFile = open(outputPath + "amber_wrapper_" + moduleName +".v", "w")

    #find //ACCTAGMOD
    for line in sourceFile:
      if "//ACCTAGMOD" in line:
        break
      oFile.write(line)

    oFile.write("module amber_wrapper_" + moduleName + "\n")

    #find //ACCTAGMODDONE
    for line in sourceFile:
      if "//ACCTAGMODDONE" in line:
        break

    #find //ACCTAGSYS
    for line in sourceFile:
      if "//ACCTAGSYS" in line:
        break
      oFile.write(line)

    oFile.write("\tamber_system_" + moduleName + "\n")

    #find //ACCTAGSYSDONE
    for line in sourceFile:
      if "//ACCTAGSYSDONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    #amber_tile.vhd
    sourceFile = open(amberTilePath + "amber_tile.vhd", "r")
    oFile = open(outputPath + "amber_tile_" + moduleName +".vhd", "w")

    #find --ACCTAGENT
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break
      oFile.write(line)

    oFile.write("entity amber_tile_" + moduleName + " is\n")

    #find --ACCTAGENTDONE
    for line in sourceFile:
      if "--ACCTAGENT" in line:
        break

    #find --ACCTAGENTEND
    for line in sourceFile:
      if "--ACCTAGENTEND" in line:
        break
      oFile.write(line)

    oFile.write("end amber_tile_" + moduleName + ";\n\n")
    oFile.write("architecture rtl of amber_tile_" + moduleName + " is\n\n
       ")

    #find --ACCTAGENTENDDONE
    for line in sourceFile:
      if "--ACCTAGENTENDDONE" in line:
        break

    #find --ACCTAGWRA
    for line in sourceFile:
      if "--ACCTAGWRA" in line:
        break
      oFile.write(line)

    oFile.write("amber_u: amber_wrapper_" + moduleName + "\n")

    #find --ACCTAGWRADONE
    for line in sourceFile:
      if "--ACCTAGWRADONE" in line:
        break

    for line in sourceFile:
      oFile.write(line)

    sourceFile.close()
    oFile.close()

    ### Generate For loop IFM

    templateFile = open("templates/floop_TEMP.v", "r")
    oFile = open(outputPath+"floop_" + moduleName +".v", "w")
    wireName = "\twire [31:0] buf_acc_"



    oFile.write("module IFM_floop_" + moduleName + "\n")

    #find #0
    for line in templateFile:
      if "#0" in line:
        break
      oFile.write(line)

    #0 found
    oFile.write("// script generated output wires\n")
    for i in range(inputsN):
      oFile.write(wireName +"in_" +str(i)+";\n")
    for i in range(outputsM):
      oFile.write(wireName +"out_"+str(i)+";\n")

    oFile.write("\n")
    oFile.write("\twbm_buffer_" + moduleName + " wbm_buffer(\n")

    #find #1
    for line in templateFile:
      if "#1" in line:
        break
      oFile.write(line)
  #1

    for i in range(inputsN):
      oFile.write("\t\t.acc_in_" + str(i) +" (buf_acc_in_"+ str(i) +"),\n")
    for i in range(outputsM):
      oFile.write("\t\t.acc_out_" + str(i) +" (buf_acc_out_" + str(i))
      if i == (outputsM - 1):
        oFile.write(")\n")
      else:
        oFile.write("),\n")

    oFile.write("\t);\n")
    oFile.write("\n")
    oFile.write("\t" + moduleName + " accelerator(\n")

    #find #2
    for line in templateFile:
      if "#2" in line:
        break
      oFile.write(line)

    #2
    for i in range(inputsN):
      oFile.write("\t\t.acc_in_" + str(i) +" (buf_acc_in_"+ str(i) +"),\n")
    for i in range(outputsM):
      oFile.write("\t\t.acc_out_" + str(i) +" (buf_acc_out_" + str(i))
        if i == (outputsM - 1):
          oFile.write(")\n")
        else:
          oFile.write("),\n")

    for line in templateFile:
      oFile.write(line)

    ###Floop IFM finished!

    oFile.close()
    templateFile.close()

    ###generating wb buffer

    oFile = open(outputPath +"wb_buffer_" + moduleName +".v", "w")
    templateFile = open("templates/wb_buffer_TEMP.v", "r")

    oFile.write("// Script generated defines \n")
    oFile.write("`define ACC_INPUTS\t\t" + str(inputsN) + "\n")
    oFile.write("`define ACC_OUTPUTS\t\t" + str(outputsM) + "\n\n")

    if(inputsN > 4):
      oFile.write("`define OUTQUEUESIZE\t\t" + str(inputsN*2) + "\n")
    else:
      oFile.write("`define OUTQUEUESIZE\t\t" + str(8) + "\n")

    if(outputsM > 4):
      oFile.write("`define INQUEUESIZE\t\t" + str(outputsM*2) + "\n\n")
    else:
      oFile.write("`define INQUEUESIZE\t\t" + str(8) + "\n\n")

    oFile.write("module wbm_buffer_" + moduleName + "\n")

    #find #0
    for line in templateFile:
      if "#0" in line:
        break
      oFile.write(line)
    #0
    for i in range(inputsN):
      oFile.write("\toutput reg\t[31:0]\tacc_in_" + str(i) +",\n")
    for i in range(outputsM):
      oFile.write("\tinput\t\t[31:0]\tacc_out_" + str(i))
      if i == (outputsM - 1):
        oFile.write("\n")
      else:
        oFile.write(",\n")

    #find #1
    for line in templateFile:
      if "#1" in line:
        break
      oFile.write(line)

    #1
    for i in range(inputsN):
            oFile.write("\t\t\t\t\tacc_in_" + str(i) +" <= input_fifo[" + str(i)
                + "];\n")

        #find #2
        for line in templateFile:
            if "#2" in line:
                break
            oFile.write(line)
        #2
        for i in range(outputsM):
            oFile.write("\t\t\t\t\toutput_buffer[" + str(i) + "] = acc_out_" +
                str(i) + ";\n")

        for line in templateFile:
            oFile.write(line)

        ###wb buffer completed!

        oFile.close()
        templateFile.close()

        ###generating amber_system
        oFile = open(outputPath +"amber_system_Floop_" + moduleName +".v", "w
            ")
        templateFile = open("templates/Floop_sys_TEMP.v", "r")



        #find #0
        for line in templateFile:
            if "#0" in line:
                break
            oFile.write(line)

        oFile.write("module amber_system_floop_" + moduleName + "\n")

        #find #1
        for line in templateFile:
            if "#1" in line:
                break
            oFile.write(line)

        oFile.write("\tIFM_floop_" + moduleName + " IFM_floop\n")
        for line in templateFile:
            oFile.write(line)

        ###Floop IFM finished!
        oFile.close()
        templateFile.close()
        print("Done!")
        print("Your verilog files can be found in the output folder")
    ##Unknown ifType
    else:
        print("ifType not recognized")


main(sys.argv[1:])
```

Listing 8: coproc_TEMP.v

```

 //start of generated section from template file
  reg [31:0]  acc_in  [0:(`ACC_INPUTS - 1)];
  reg [31:0]  acc_out[0:(`ACC_OUTPUTS - 1)];
  reg [31:0]  acc_opt;
  wire    acc_poll;


 #0


  reg acc_start;
  wire acc_rdy;
  integer i;

  assign acc_poll = ((!acc_start) && (acc_rdy || acc_poll));

#1
  .i_clk (i_clk),
  .i_rst (i_rst),
  .i_start (acc_start),
  .i_opt (acc_opt),
  .o_rdy (acc_rdy),
#2
 );
 //end of generated section

 #3

 ////start of generated section from template file

  always @(posedge acc_rdy)
 begin
#4
 end

always @ ( posedge i_clk )
  if(i_rst)
  begin
    cache_control <= 3'b000;
    cacheable_area <= 32'h0;
    updateable_area <= 32'h0;
    disruptive_area <= 32'h0;
    acc_start <= 'h0;
  end
  else
  begin
    if(acc_start)
      acc_start <= 0;
    if ( !i_core_stall )
    begin
      if ( i_copro_operation == 2'd2 )
        if (i_copro_num == 'd14) //TAG_ACC
        begin
          if(i_copro_crn == 4'hf)
          begin
```

```
            acc_opt <= i_copro_write_data[31:0];
            acc_start <= 1;
          end
          if (i_copro_crn <= (`ACC_INPUTS))
            acc_in[i_copro_crn] <= i_copro_write_data[31:0];
        end
        else
          case ( i_copro_crn )
              4'd2: cache_control   <= i_copro_write_data[2:0];
              4'd3: cacheable_area  <= i_copro_write_data[31:0];
              4'd4: updateable_area <= i_copro_write_data[31:0];
              4'd5: disruptive_area <= i_copro_write_data[31:0];
          endcase
    end
  end

// Flush the cache
assign copro15_reg1_write = !i_core_stall && i_copro_operation == 2'd2 &&
    i_copro_crn == 4'd1 && i_copro_num == 'd15;

 // ---------------------------
 // Register Reads
 // ---------------------------
always @ ( posedge i_clk )
     if ( !i_core_stall )
    if (i_copro_num == 'd14)//TAG_ACC
    begin
      if(i_copro_crn == 4'hf)
        o_copro_read_data <= {31'b0,acc_poll};
      else
        o_copro_read_data <= acc_out[i_copro_crn];
    end
    else
         case ( i_copro_crn )
              4'd0:    o_copro_read_data <= 32'h4156_0300;
              4'd2:    o_copro_read_data <= {29'd0, cache_control};
              4'd3:    o_copro_read_data <= cacheable_area;
              4'd4:    o_copro_read_data <= updateable_area;
              4'd5:    o_copro_read_data <= disruptive_area;
              4'd6:    o_copro_read_data <= {24'd0, fault_status };
              4'd7:    o_copro_read_data <= fault_address;
              default: o_copro_read_data <= 32'd0;
         endcase
endmodule
```

Listing 9: Floop_sys_TEMP.v

```
//------------------------------------------------------------------------

// Title          : Amber system
// Project        : SHMAC
//------------------------------------------------------------------------

// File           : amber_system_<acc module>.v
// Author         : Asbjorn Djupdal <djupdal@idi.ntnu.no>, edited by
    Marton Teilgard
//           for module generation
// Created       : 06.09.2013
// Last modified : ??.06.2014
//------------------------------------------------------------------------
// Description : Wishbone based system containing CPU, timer,
// interrupt controller and tile register
//
//------------------------------------------------------------------------
// Copyright (c) 2013 by ARM/CARD
//------------------------------------------------------------------------
// Modification history :
// ??.07.2013 : created
//------------------------------------------------------------------------

`include "common_defs.v"

#0
  #(
    parameter tile_x = 4'b0,
    parameter tile_y = 4'b0,
    parameter cpu_id = 8'hff
  )
   (
    input wire          i_clk,
    input wire          i_rst,

    input wire          i_irq,

    input wire          i_system_rdy,

    output wire [31:0]  o_wb_adr,
    output wire [15:0]  o_wb_sel,
    output wire         o_wb_we,
    input wire [127:0]  i_wb_dat,
    output wire [127:0] o_wb_dat,
    output wire         o_wb_cyc,
    output wire          o_wb_stb,
    input wire          i_wb_ack,
    input wire          i_wb_err,
    output wire         o_ld_excl

    );

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


   wire                 irq;
   wire                 firq;

   wire [31:0]       in0_o_wbm_adr;
   wire [15:0]       in0_o_wbm_sel;
   wire          in0_o_wbm_we;
   wire [127:0]      in0_o_wbm_dat;
   wire            in0_o_wbm_cyc;
   reg             in0_o_wbm_stb;
   wire [127:0]        in0_i_wbm_dat;


   wire [31:0]         acc_o_wbm_adr;
   wire [15:0]         acc_o_wbm_sel;
   wire            acc_o_wbm_we;
   wire [127:0]        acc_o_wbm_dat;
   wire            acc_o_wbm_cyc;
   wire            acc_o_wbm_stb;
   wire [127:0]        acc_i_wbm_dat;


   reg [127:0]             wb_dat_r_cpu;
   wire [127:0]            wb_dat_r_tileregs;
   wire [127:0]            wb_dat_r_timer;
   wire [127:0]            wb_dat_r_irq;
   wire [127:0]            wb_dat_r_acc;

   reg                     wb_ack_cpu;
   wire                    wb_ack_tileregs;
   wire                    wb_ack_timer;
   wire                    wb_ack_irq;
   wire                    wb_ack_acc;

   reg                     wb_err_cpu;
   wire                    wb_err_tileregs;
   wire                    wb_err_timer;
   wire                    wb_err_irq;
   wire                    wb_err_acc;

   wire                    wb_stb_cpu;
   reg                     wb_stb_tileregs;
   reg                     wb_stb_timer;
   reg                     wb_stb_irq;
   reg                     wb_stb_acc;

   wire [2:0]              irq_timers;
   wire            irq_acc;
   wire [31:1]             int_sources;

   wire [31:0]             tile_base = `TILE_BASE;
   wire [15:0]             tilereg = `TILE_REGS;
   wire [15:0]             timer_mod = `TIMER;
   wire [15:0]             int_ctrl = `INT_CTRL;
   wire [15:0]         acc = `ACC;

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

   a25_core u_amber
     (
       .i_clk              (i_clk),
       .i_rst              (i_rst),

       .i_irq              (irq),
       .i_firq          (firq),

       .i_system_rdy    (i_system_rdy),

       .o_wb_adr        (in0_o_wbm_adr),
       .o_wb_sel        (in0_o_wbm_sel),
       .o_wb_we         (in0_o_wbm_we),
       .i_wb_dat        (wb_dat_r_cpu),
       .o_wb_dat        (in0_o_wbm_dat),
       .o_wb_cyc        (in0_o_wbm_cyc),
       .o_wb_stb        (wb_stb_cpu),
       .i_wb_ack        (wb_ack_cpu),
       .i_wb_err        (wb_err_cpu),
       .o_ld_excl       (o_ld_excl)
       );

   tile_regs
     #(
       .WB_DWIDTH (128),
       .WB_SWIDTH (16),
       .tile_x (tile_x),
       .tile_y (tile_y),
       .cpu_id (cpu_id)
       )
   u_tile_regs
     (
       .i_clk (i_clk),
       .i_rst (i_rst),

       .i_wb_adr (in0_o_wbm_adr),
       .i_wb_sel (in0_o_wbm_sel),
       .i_wb_we (in0_o_wbm_we),
       .o_wb_dat (wb_dat_r_tileregs),
       .i_wb_dat (in0_o_wbm_dat),
       .i_wb_cyc (in0_o_wbm_cyc),
       .i_wb_stb (wb_stb_tileregs),
       .o_wb_ack (wb_ack_tileregs),
       .o_wb_err (wb_err_tileregs)
       );

   timer_module
     #(
       .WB_DWIDTH (128),
       .WB_SWIDTH (16)
       )
   u_timer
     (
       .i_clk (i_clk),
       .i_rst (i_rst),

       .i_wb_adr (in0_o_wbm_adr),
       .i_wb_sel (in0_o_wbm_sel),
       .i_wb_we (in0_o_wbm_we),
       .o_wb_dat (wb_dat_r_timer),
       .i_wb_dat (in0_o_wbm_dat),
       .i_wb_cyc (in0_o_wbm_cyc),
       .i_wb_stb (wb_stb_timer),
       .o_wb_ack (wb_ack_timer),
       .o_wb_err (wb_err_timer),
       .o_timer_int (irq_timers)
       );

   interrupt_controller
     #(
       .WB_DWIDTH (128),
       .WB_SWIDTH (16)
       )
   u_irq_ctrl
     (
       .i_clk (i_clk),
       .i_rst (i_rst),

       .i_wb_adr (in0_o_wbm_adr),
       .i_wb_sel (in0_o_wbm_sel),
       .i_wb_we (in0_o_wbm_we),
       .o_wb_dat (wb_dat_r_irq),
       .i_wb_dat (in0_o_wbm_dat),
       .i_wb_cyc (in0_o_wbm_cyc),
       .i_wb_stb (wb_stb_irq),
       .o_wb_ack (wb_ack_irq),
       .o_wb_err (wb_err_irq),

       .o_irq (irq),
       .o_firq (firq),

       .i_int_sources (int_sources)
       );

  #1
  (
    .i_clk(i_clk),
    .i_rst(i_rst),
    .o_irq(irq_acc),
    .o_wbm_adr(acc_o_wbm_adr),
    .o_wbm_sel(acc_o_wbm_sel),
    .o_wbm_we(acc_o_wbm_we),
    .i_wbm_dat(acc_i_wbm_dat),
    .o_wbm_dat(acc_o_wbm_dat),
    .o_wbm_cyc(acc_o_wbm_cyc),
    .o_wbm_stb(acc_o_wbm_stb),
    .i_wbm_ack(acc_i_wbm_ack),
    .i_wbm_err(acc_i_wbm_err),
    .i_wbs_adr (in0_o_wbm_adr),
       .i_wbs_sel (in0_o_wbm_sel),
       .i_wbs_we (in0_o_wbm_we),
       .o_wbs_dat (wb_dat_r_acc),
       .i_wbs_dat (in0_o_wbm_dat),
       .i_wbs_cyc (in0_o_wbm_cyc),
       .i_wbs_stb (wb_stb_acc),
       .o_wbs_ack (wb_ack_acc),
       .o_wbs_err (wb_err_acc)
  );

  wb_arbiter u_arbiter (
    .i_rst(i_rst),
    .o_wbm_adr(o_wb_adr),
    .o_wbm_sel(o_wb_sel),
    .o_wbm_we(o_wb_we),
    .i_wbm_dat(i_wb_dat),
    .o_wbm_dat(o_wb_dat),
    .o_wbm_cyc(o_wb_cyc),
    .o_wbm_stb(o_wb_stb),
    .i_wbm_ack(i_wb_ack),
    .i_wbm_err(i_wb_err),
    .in0_o_wbm_adr(in0_o_wbm_adr),
    .in0_o_wbm_sel(in0_o_wbm_sel),
    .in0_o_wbm_we(in0_o_wbm_we),
    .in0_i_wbm_dat(in0_i_wbm_dat),
    .in0_o_wbm_dat(in0_o_wbm_dat),
    .in0_o_wbm_cyc(in0_o_wbm_cyc),
    .in0_o_wbm_stb(in0_o_wbm_stb),
    .in0_i_wbm_ack(in0_i_wbm_ack),
    .in0_i_wbm_err(in0_i_wbm_err),
    .in1_o_wbm_adr(acc_o_wbm_adr),
    .in1_o_wbm_sel(acc_o_wbm_sel),
    .in1_o_wbm_we(acc_o_wbm_we),
    .in1_i_wbm_dat(acc_i_wbm_dat),
    .in1_o_wbm_dat(acc_o_wbm_dat),
    .in1_o_wbm_cyc(acc_o_wbm_cyc),
    .in1_o_wbm_stb(acc_o_wbm_stb),
    .in1_i_wbm_ack(acc_i_wbm_ack),
    .in1_i_wbm_err(acc_i_wbm_err)
 );

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


   assign int_sources = {26'b0,irq_acc, irq_timers, i_irq};

   always @* begin
      // default is router
      wb_dat_r_cpu = in0_i_wbm_dat;
      wb_ack_cpu = in0_i_wbm_ack;
      wb_err_cpu = in0_i_wbm_err;
      in0_o_wbm_stb = wb_stb_cpu;
      wb_stb_tileregs = 0;
      wb_stb_timer = 0;
      wb_stb_irq = 0;
    wb_stb_acc = 0;

      // override default for local wishbone addresses
      if(in0_o_wbm_adr[31:16] == tile_base[31:16]) begin
         case(in0_o_wbm_adr[15:12])
           tilereg[15:12]: begin
              wb_dat_r_cpu = wb_dat_r_tileregs;
              wb_ack_cpu = wb_ack_tileregs;
              wb_err_cpu = wb_err_tileregs;
              in0_o_wbm_stb = 0;
              wb_stb_tileregs = wb_stb_cpu;
           end

```

```
            timer_mod[15:12]: begin
               wb_dat_r_cpu = wb_dat_r_timer;
               wb_ack_cpu = wb_ack_timer;
               wb_err_cpu = wb_err_timer;
               in0_o_wbm_stb = 0;
               wb_stb_timer = wb_stb_cpu;
            end

            int_ctrl[15:12]: begin
               wb_dat_r_cpu = wb_dat_r_irq;
               wb_ack_cpu = wb_ack_irq;
               wb_err_cpu = wb_err_irq;
               in0_o_wbm_stb = 0;
               wb_stb_irq = wb_stb_cpu;
            end

         acc[15:12]: begin
               wb_dat_r_cpu = wb_dat_r_acc;
               wb_ack_cpu = wb_ack_acc;
               wb_err_cpu = wb_err_acc;
               in0_o_wbm_stb = 0;
               wb_stb_acc = wb_stb_cpu;
            end
          endcase
      end

   end

endmodule
```

Listing 10: floop_TEMP.v

```
  (
    input                 i_clk,
    input                 i_rst,
    output reg          o_irq,
    // Wishbone Master I/F
    output     [31:0]     o_wbm_adr,
    output     [15:0]        o_wbm_sel,
    output              o_wbm_we,
    input      [127:0]       i_wbm_dat,
    output     [127:0]    o_wbm_dat,
    output              o_wbm_cyc,
    output                o_wbm_stb,
    input                 i_wbm_ack,
    input                 i_wbm_err,
    //wishbone Slave IF
    input      [31:0]        i_wbs_adr,
    input      [15:0]     i_wbs_sel,
    input                    i_wbs_we,
    output reg   [127:0]   o_wbs_dat,
    input      [127:0]    i_wbs_dat,
    input                    i_wbs_cyc,
    input                       i_wbs_stb,
    output reg                  o_wbs_ack,
    output reg               o_wbs_err
  );

  //statelist
  localparam  IDLE = 4'h0,
        REC = 4'h1,
        ADRS = 4'h2,
        FETCH = 4'h3,
        CALC = 4'h4,
        STORE = 4'h5,
        IRQ = 4'h6;

  //control registers
  reg [31:0]    options;
  reg [31:0]    main_i;
  reg [31:0]    in_adr;
  reg [31:0]    out_adr;
  reg [3:0]   state;
  reg        reset_buffer;

  // accelerator control
  reg acc_start;
  wire acc_rdy;
  wire [31:0] acc_opt;

  //wishbone master buffer IF
  reg [31:0] buf_input_adr;
  reg [31:0] buf_output_adr;
  reg buf_set_adr;
  reg buf_read_input;
  reg buf_write_output;
  reg buf_last_read;
  wire buf_stall;
  wire buf_rst;

  #0
    .i_clk(i_clk),
    .i_rst(buf_rst),
    .i_input_adr(in_adr),
    .i_output_adr(out_adr),
    .i_set_adr(buf_set_adr),
    .i_read_input(buf_read_input),
    .i_write_output(buf_write_output),
    .i_last_read(buf_last_read),
    .o_stall(buf_stall),
    .o_wbm_adr(o_wbm_adr),
    .o_wbm_sel(o_wbm_sel),
    .o_wbm_we(o_wbm_we),
    .i_wbm_dat(i_wbm_dat),
    .o_wbm_dat(o_wbm_dat),
    .o_wbm_cyc(o_wbm_cyc),
    .o_wbm_stb(o_wbm_stb),
    .i_wbm_ack(i_wbm_ack),
    .i_wbm_err(i_wbm_err),
    #1
    .i_clk (i_clk),
    .i_rst (i_rst),
    .i_start (acc_start),
    .i_opt (options),
    .o_rdy (acc_rdy),
    #2
  );
  //assign signals
  assign buf_rst = i_rst || reset_buffer;

  //main operation
  always @(posedge i_clk)
  begin
    if(i_rst)
    begin
      o_wbs_dat <= 'h0;
      o_wbs_ack <= 'h0;
      o_wbs_err <= 'h0;
      options <= 'h0;
      main_i <= 'h0;
      in_adr <= 'h0;
      out_adr <= 'h0;
      state = IDLE;
      //wb buffer reset
      reset_buffer <= 0;
      buf_input_adr <= 'h0;
      buf_output_adr <= 'h0;
      buf_set_adr <= 'h0;
      buf_read_input <= 'h0;
      buf_write_output <= 'h0;
      buf_last_read <= 'h0;

    end//reset
    else
    begin
      case(state)
        IDLE:
        begin
          o_irq <= 0;
          reset_buffer <= 0;
          buf_last_read <= 0;
          if(i_wbs_stb)
            state = REC;
        end
        REC:
        begin
          if(i_wbs_stb)
          begin
            if(!o_wbs_ack)
            begin
              if(i_wbs_adr[11:4] == 8'h00)
              begin
                if(i_wbs_sel[3])
                begin
                  options <= i_wbs_dat[31:0];
                  buf_set_adr <= 1;
                  state = ADRS;
                end
                if(i_wbs_sel[7])
                  main_i <= i_wbs_dat[63:32];
                if(i_wbs_sel[11])
                  in_adr <= i_wbs_dat[95:64];
                if(i_wbs_sel[15])
                  out_adr <= i_wbs_dat[127:96];
                o_wbs_ack <= 1;
              end
            end
            else
              o_wbs_ack <= 0;
          end
          else
            o_wbs_ack <= 0;
        end//rec
        ADRS:
        begin
          buf_set_adr <= 0;
          o_wbs_ack <= 0;
          if(!buf_stall)
          begin
            buf_read_input <= 1;
            if(main_i == 1)
              buf_last_read <= 1;
            state = FETCH;
          end
        end//ADRS
        FETCH:
        begin
          if(!buf_stall)
            if(buf_read_input)
              buf_read_input <= 0;
            else
            begin
              acc_start <= 1;
              state = CALC;
            end
        end//FETCH
        CALC:
        begin
          acc_start <= 0;
          if(acc_rdy)
          begin
            buf_write_output <= 1;
            main_i <= main_i -1;
            state = STORE;
          end
        end//CALC
        STORE:
        begin
          if(buf_write_output)
            buf_write_output <= 0;
          else
          begin
            if(main_i == 0)
              state = IRQ;
            else if(main_i == 1)
            begin
              if(!buf_stall)
              begin
```

```
                 buf_read_input <= 1;
                 buf_last_read <= 1;
                 state = FETCH;
               end
             end
             else
             begin
               buf_read_input <= 1;
               state = FETCH;
             end
           end
         end //STORE
         IRQ:
         begin
           if(!buf_stall)
           begin
             reset_buffer <= 1;
             o_irq <= 1;
             state = IDLE;
           end
         end //IRQ
       endcase
     end//normal op
   end// @i_clk
endmodule
```

Listing 11: slave_sys_TEMP.v

```
  wire [127:0]         wb_dat_r_acc;
  wire                 wb_ack_acc;
  wire                 wb_err_acc;
  reg                  wb_stb_acc;
  wire         irq_acc;
    wire [15:0]      acc = `ACC;

  assign int_sources = {26'b0,irq_acc , irq_timers , i_irq};

  #0
    .i_clk(i_clk),
    .i_rst(i_rst),
    .i_wb_adr (o_wb_adr),
      .i_wb_sel (o_wb_sel),
      .i_wb_we  (o_wb_we),
      .o_wb_dat (wb_dat_r_acc),
      .i_wb_dat (o_wb_dat),
      .i_wb_cyc (o_wb_cyc),
      .i_wb_stb (wb_stb_acc),
      .o_wb_ack (wb_ack_acc),
      .o_wb_err (wb_err_acc),
    .o_irq (irq_acc)
  );



  #1

            acc[15:12]: begin
```

```
                wb_dat_r_cpu = wb_dat_r_acc;
                wb_ack_cpu = wb_ack_acc;
                wb_err_cpu = wb_err_acc;
                o_wb_stb = 0;
                wb_stb_acc = wb_stb_cpu;
            end
```

Listing 12: slave_TEMP.v

```
//////////////////////////////////////////////////////////////////////
//                                                                  //
//   Slave Interface Module                                         //
//                                                                  //
//  Description                                                     //
//   This is a generated Accelerator interface module connected     //
//   To the Amber Cores wishbone interface in the SHMAC system      //
//                                                                  //
//   Author(s):                                                     //
//       - Marton Teilgard, mteilgard@gmail.com                   //
//                                                                  //
//////////////////////////////////////////////////////////////////////


`include "common_defs.v"

//ACC IF parameters generated
#0
   (
    input                     i_clk,
    input                     i_rst,
    input [31:0]              i_wb_adr,
    input [15:0]           i_wb_sel,
    input                     i_wb_we,
    output reg [127:0]        o_wb_dat,
    input [127:0]          i_wb_dat,
    input                     i_wb_cyc,
    input                     i_wb_stb,
    output                    o_wb_ack,
    output                    o_wb_err,
  output                   o_irq
  );


  reg [31:0] acc_in [0:(`ACC_INPUTS - 1)];
  reg [31:0] acc_out[0:(`ACC_OUTPUTS - 1)];
  reg [31:0] acc_opt;
  reg acc_start;
  wire acc_rdy;

  assign o_irq = acc_rdy;
  //Output wires generated
  #1
  .i_clk (i_clk),
  .i_rst (i_rst),
  .i_start (acc_start),
  .i_opt (acc_opt),
  .o_rdy (acc_rdy),
  #2
);
   // Wishbone interface
   wire                                              wb_start_write;
   wire                                              wb_start_read;
   reg                                               wb_start_read_d1 = 'd0;


   // ======================================================
   // Wishbone Interface
   // ======================================================

   // Can't start a write while a read is completing. The ack for the read
       cycle
   // needs to be sent first
   assign wb_start_write = i_wb_stb && i_wb_we && !wb_start_read_d1;
   assign wb_start_read  = i_wb_stb && !i_wb_we && !o_wb_ack;

   always @( posedge i_clk or posedge i_rst) begin
      if(i_rst)
        wb_start_read_d1 <= 1'b0;
      else
        wb_start_read_d1 <= wb_start_read;
   end

   assign o_wb_err = 1'd0;
   assign o_wb_ack = i_wb_stb && ( wb_start_write || wb_start_read_d1 );


  always @(posedge acc_rdy)
  begin
  #3
  end
   // ======================================================
   // Register Writes
   // ======================================================
  always @( posedge i_clk or posedge i_rst) begin
  if(i_rst)
  begin//ACC IF generate this list
        acc_start <= 'h0;
  end
    else
  begin
    if(acc_start)
      acc_start <= 0;
        if ( wb_start_write && !i_wb_adr[11])
      begin
        if(i_wb_sel[3])
          acc_in[{i_wb_adr[10:4],2'b00}] <= i_wb_dat[31:0];
        if(i_wb_sel[7])
          acc_in[{i_wb_adr[10:4], 2'b01}] <= i_wb_dat[63:32];
        if(i_wb_sel[11])
          acc_in[{i_wb_adr[10:4], 2'b10}] <= i_wb_dat[95:64];
        if(i_wb_sel[15])
        begin
          if(i_wb_adr[10:2] == 9'h1ff)
          begin
```

```
            acc_opt <= i_wb_dat[127:96];
            acc_start <= 1;
          end
          else
            acc_in[{i_wb_adr[10:4], 2'b11}] <= i_wb_dat[127:96];
        end
      end
    end
  end

  // =========================================================
   // Register Reads
   // =========================================================
   always @( posedge i_clk or posedge i_rst ) begin
      if(i_rst)
         o_wb_dat <= 127'h0;
      else
    begin
      if ( wb_start_read && i_wb_adr[11])
      begin
        if(i_wb_sel[3])
          o_wb_dat[31:0] <= acc_out[{i_wb_adr[10:4],2'b00}];
        if(i_wb_sel[7])
          o_wb_dat[63:32] <= acc_out[{i_wb_adr[10:4],2'b01}];
        if(i_wb_sel[11])
          o_wb_dat[95:64] <= acc_out[{i_wb_adr[10:4],2'b10}];
        if(i_wb_sel[15])
          o_wb_dat[127:96] <= acc_out[{i_wb_adr[10:4],2'b11}];
      end
      else
        o_wb_dat[127:0] <= 128'b0;
    end
   end

endmodule
```

Listing 13: wb_arbiter_TEMP.v

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////////////////////

// Company:
// Engineer:
//
// Create Date:    16:52:09 05/03/2014
// Design Name:
// Module Name:    wb_arbiter
// Project Name:
// Target Devices:
// Tool versions:
// Description: This is a basic two input, one output round robin wishbone
     master
// arbiter. It checks on start and end of cycle on the wb_cyc signals of
    the two
// masters, and therefore it supports multi-read/write-cycle operations
    without interrupting
// the transfer.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////
module wb_arbiter(
  input                   i_rst ,

  // Wishbone Master I/F
  output    [31:0]          o_wbm_adr,
  output    [15:0]          o_wbm_sel,
  output                  o_wbm_we,
  input     [127:0]         i_wbm_dat,
  output    [127:0]       o_wbm_dat,
  output                    o_wbm_cyc,
  output                  o_wbm_stb,
  input                   i_wbm_ack,
  input                   i_wbm_err,

  //in0
  input   [31:0]            in0_o_wbm_adr,
  input     [15:0]          in0_o_wbm_sel,
  input                     in0_o_wbm_we,
  output    [127:0]         in0_i_wbm_dat,
  input     [127:0]       in0_o_wbm_dat,
  input                     in0_o_wbm_cyc,
  input                     in0_o_wbm_stb,
  output                    in0_i_wbm_ack,
  output                    in0_i_wbm_err,

  //in1
  input   [31:0]            in1_o_wbm_adr,
  input     [15:0]          in1_o_wbm_sel,
  input                     in1_o_wbm_we,
  output    [127:0]         in1_i_wbm_dat,
  input     [127:0]       in1_o_wbm_dat,
  input                     in1_o_wbm_cyc,
  input                     in1_o_wbm_stb,
  output                    in1_i_wbm_ack,
  output                    in1_i_wbm_err

);

  reg sel;
  reg cyc0;
  reg cyc1;
  wire cyc;

  //outputs
  assign o_wbm_adr = (sel) ? in1_o_wbm_adr : in0_o_wbm_adr;
  assign o_wbm_sel = (sel) ? in1_o_wbm_sel : in0_o_wbm_sel;
  assign o_wbm_we = (sel) ? in1_o_wbm_we : in0_o_wbm_we;
  assign o_wbm_dat = (sel) ? in1_o_wbm_dat : in0_o_wbm_dat;
  assign o_wbm_cyc = (sel) ? in1_o_wbm_cyc : in0_o_wbm_cyc;
  assign o_wbm_stb = (sel) ? in1_o_wbm_stb : in0_o_wbm_stb;

  //inputs
  assign in0_i_wbm_dat = (sel) ? 127'h0 : i_wbm_dat;
  assign in1_i_wbm_dat = (sel) ? i_wbm_dat: 127'h0;
  assign in0_i_wbm_ack = (sel) ? 0 : i_wbm_ack;
  assign in1_i_wbm_ack = (sel) ? i_wbm_ack: 0;
  assign in0_i_wbm_err = (sel) ? 0 : i_wbm_err;
  assign in1_i_wbm_err = (sel) ? i_wbm_err: 0;

  assign cyc = cyc0 || cyc1;

  always @*
  if(i_rst)
  begin
    sel <= 0;
    cyc0 <= 0;
    cyc1 <= 0;
  end
  else
  begin
    if (!cyc)
    begin
      if (in0_o_wbm_cyc)
      begin
        sel <= 0;
        cyc0 <= 1;
      end
      if (in1_o_wbm_cyc)
      begin
        sel <= 1;
        cyc1 <= 1;
      end
    end
    else
    begin
      if(cyc0)
      begin
        if (!in0_o_wbm_cyc)
          if (in1_o_wbm_cyc)
          begin
            sel <= 1;
            cyc1 <= 1;
          end
          else
            cyc0 <= 0;
      end
      if(cyc1)
      begin
        if (!in1_o_wbm_cyc)
          if (in0_o_wbm_cyc)
          begin
            sel <= 0;
            cyc0 <= 1;
          end
          else
            cyc1 <= 0;
      end
    end
  end
endmodule
```

Listing 14: wb_buffer_TEMP.v

```
  (
  input i_clk,
  input i_rst,

  input   [31:0]    i_input_adr,
  input   [31:0]    i_output_adr,
  input           i_set_adr,
  input             i_read_input,
  input             i_write_output,
  input           i_last_read,
  output            o_stall,
  //wishbone IF here
  output reg  [31:0]    o_wbm_adr,
  output reg  [15:0]    o_wbm_sel,
  output reg            o_wbm_we,
  input     [127:0]   i_wbm_dat,
  output    [127:0]   o_wbm_dat,
  output reg            o_wbm_cyc,
  output reg            o_wbm_stb,
  input                 i_wbm_ack,
  input                 i_wbm_err,
  //script generated interface
  #0
  );

  //regs
  reg [31:0]  input_adr;
  reg [31:0]  output_adr;
  reg [31:0]  output_buffer [0:`ACC_OUTPUTS-1];
  reg [31:0]  output_fifo   [0:`OUTQUEUESIZE-1];
  reg [31:0]  input_fifo    [0:`INQUEUESIZE-1];
  reg [31:0]  read_buffer   [0:3];
  reg [31:0]  write_buffer  [0:3];
  integer   out_i;  //TODO generate size of these?
  integer   in_i;
  reg     new_dataout;
  reg     new_datain;
  reg     read;
  reg     first_read;
  reg     read_stall;
  reg     write_stall;
  reg     write;
  integer i;

  assign o_stall = read_stall || write_stall;
  assign o_wbm_dat [31:0] = write_buffer[0];
  assign o_wbm_dat [63:32] = write_buffer[1];
  assign o_wbm_dat [95:64] = write_buffer[2];
  assign o_wbm_dat [127:96] = write_buffer[3];

```

```
always @(posedge i_clk or posedge i_rst)
begin
  if (i_rst)
  begin
    for (i=0; i<`ACC_OUTPUTS; i=i+1)
      output_buffer[i] = 'h0;
    for (i=0; i<4; i=i+1)
      write_buffer[i] = 'h0;
    for (i=0; i<`OUTQUEUESIZE; i=i+1)
      output_fifo[i] = 'h0;
    for (i=0; i<`INQUEUESIZE; i=i+1)
      input_fifo[i] = 'h0;
    out_i = 'h0;
    in_i = 'h0;
    new_dataout = 'h0;
    new_datain = 'h0;
    read_stall = 'h1;
    write_stall = 'h0;
    read = 'h0;
    first_read = 'h0;
    input_adr = 'h0;
    output_adr = 'h0;
    //wb rst
    o_wbm_adr <= 'h0;
    o_wbm_sel <= 'h0;
    o_wbm_we <= 'h0;
    o_wbm_cyc <= 'h0;
    o_wbm_stb <= 'h0;
    write = 'h0;
  end// rst
  else
  begin
    if(i_set_adr)
    begin
      input_adr = i_input_adr;
      output_adr = i_output_adr;
      read_stall = 1;
      read = 1;
      first_read = 1;
    end
    else if (!o_stall)
    begin
      if(i_read_input)
      begin
        #1
        for(i = 0; i < `INQUEUESIZE-`ACC_INPUTS; i = i+1)
          input_fifo[i] = input_fifo[i+`ACC_INPUTS];
        in_i = in_i - `ACC_INPUTS;
        read_stall = 1;
      end //read input
      if(i_write_output)
      begin
        write_stall = 1;
        new_dataout = 1;
        #2
      end// write out
    end// if not stalls
```

```
      else // stalls
      begin
        if(write_stall)
        begin
          if(new_dataout)
          begin
            for(i = 0; i < `ACC_OUTPUTS; i = i+1)
            begin
              output_fifo[out_i] = output_buffer[i];
              out_i = out_i + 1;
            end
            new_dataout = 0;
          end // new dataout
          if (write)
          begin
            //write to wbm
            if(o_wbm_stb == 0)
            begin
              o_wbm_adr <= output_adr;
              o_wbm_sel <= 16'hffff;
              o_wbm_cyc <= 1;
              o_wbm_stb <= 1;
              o_wbm_we <= 1;
            end//strobe == 0
            else if(i_wbm_ack)
              begin
                output_adr = output_adr + 'h10;
                write = 0;
                for (i=0; i<4; i=i+1)
                  write_buffer[i] = 'h0;
                o_wbm_adr <= 32'h0000;
                o_wbm_sel <= 16'h0000;
                o_wbm_cyc <= 0;
                o_wbm_stb <= 0;
                o_wbm_we <= 0;
              end//ack
          end//write
          else if(out_i > 3)
          begin
            for(i = 0; i < 4; i = i+1)
              write_buffer[i] = output_fifo[i];
            for(i = 0; i < `OUTQUEUESIZE-4; i = i+1)
              output_fifo[i] = output_fifo[i+4];
            out_i = out_i - 4;
            write = 1;
          end
          else if(i_last_read)
          begin
            case(out_i)
              'h0: write_stall = 0;
              'h1:
              begin
                write_buffer[0] = output_fifo[0];
                write_buffer[1] = 32'h0;
                write_buffer[2] = 32'h0;
                write_buffer[3] = 32'h0;
                write = 1;
              end
              'h2:
              begin
                write_buffer[0] = output_fifo[0];
                write_buffer[1] = output_fifo[1];
                write_buffer[2] = 32'h0;
                write_buffer[3] = 32'h0;
                write = 1;
              end
              'h3:
              begin
                write_buffer[0] = output_fifo[0];
                write_buffer[1] = output_fifo[1];
                write_buffer[2] = output_fifo[2];
                write_buffer[3] = 32'h0;
                write = 1;
              end
            endcase
            out_i = 0;
          end// last_read
          else
            write_stall = 0;
        end// o_write_stall
        else  // o_read_stall
        begin
          if(first_read)
          begin
            if(read)
            begin
              if(!o_wbm_stb)
              begin
                o_wbm_adr <= input_adr;
                o_wbm_sel <= 16'hffff;
                o_wbm_cyc <= 1;
                o_wbm_stb <= 1;
                o_wbm_we <= 0;
              end//!strobe
              else if(i_wbm_ack)
              begin
                o_wbm_adr <= 0;
                o_wbm_sel <= 16'h0;
                o_wbm_cyc <= 0;
                o_wbm_stb <= 0;
                o_wbm_we <= 0;

                read_buffer[0] = i_wbm_dat[31:0];
                read_buffer[1] = i_wbm_dat[63:32];
                read_buffer[2] = i_wbm_dat[95:64];
                read_buffer[3] = i_wbm_dat[127:96];
                input_adr = input_adr + 'h10;
                new_datain = 1;
                read = 0;
              end

            end// read
            else if(new_datain)
            begin
              for(i = 0; i < 4; i = i + 1)
              begin
                input_fifo[in_i] = read_buffer[i];
                in_i = in_i + 1;
              end
              new_datain = 0;
              if(in_i >= `ACC_INPUTS || i_last_read)
                read_stall = 0;
              else
                read = 1;
            end
            else if(in_i < `ACC_INPUTS)
              read = 1;
            else
              read_stall = 0;
          end//first read
        end//read_stall
      end//else stalls
    end//rst else
  end//posedge i_clk

  endmodule
```

.3 AppendixC: Area calculations

Table 1: Area calculations for Coprocessor IFM

Module	**Nr of slice registers**	**Nr of slice luts**
coproc IFM with acc 4 4 IO	460	183
coproc IFM with acc 4 1 IO	364	102
coproc IFM with acc 1 4 IO	364	214
base size	204	116,3333333333
size of one input	32	-10,3333333333
size of one output	32	27

Table 2: Area calculations for Slave IFM

Module	**Nr of slice registers**	**Nr of slice luts**
Slave IFM with acc 4 4 IO	418	301
Slave IFM with acc 4 1 IO	322	208
Slave IFM with acc 1 4 IO	322	234
base size	162	87,6666666667
size of one input	32	22,3333333333
size of one output	32	31

Table 3: Area calculations for For loop IFM

Module	**Nr of slice registers**	**Nr of slice luts**
Floop IFM with acc 4 4 IO	1301	3162
floop IFM with acc 4 1 IO	1204	1516
floop IFM with acc 1 4 IO	1205	3174
base size	1043,6666666667	983,3333333333
size of one input	32	-4
size of one output	32,3333333333	548,6666666667
wb_arbiter	3	444
Constant for size calc	**Nr of slice registers**	**Nr of slice luts**
base size	1046,67	1427,33
size of one input	32,00	-4,00
size of one output	32,33	548,67