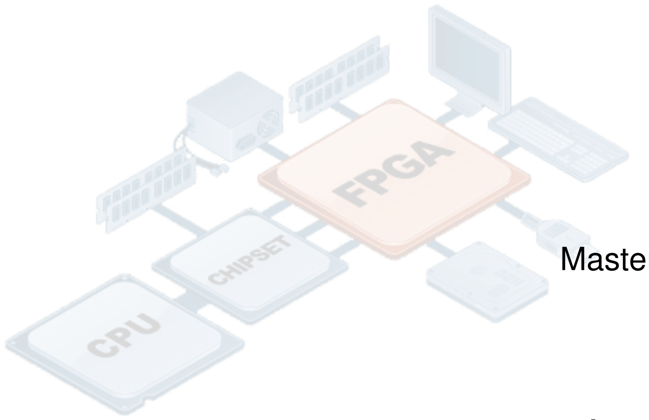
ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Full System Architectural Simulation on the HARP Integrated CPU-FPGA Platform

Konstantinos N. Kyriakidis



Master thesis submitted under the supervision of
Professor Dr. Dionisios N. Pnevmatikatos

the co-supervision of
Professor Dr. Apostolos Dollas
Associate Professor Dr. Ioannis Papaefstathiou

in order to be awarded the Degree of
Master of Science in Electrical and Computer Engineering
Major in Computer Hardware Architecture

Academic year
2018 – 2019

Title: Full System Architectural Simulation on the HARP Integrated CPU-FPGA Platform

Author: Konstantinos N. Kyriakidis
Master of Science in Electrical and Computer Engineering
Academic year: 2018 – 2019

Simulation is vital when developing novel software or hardware systems. Cycle accurate architectural simulators are extremely important tools for verifying experimental hardware platforms, system profiling, and advanced software development. Their main disadvantage is limited throughput when simulating large systems with multiple processing units and peripherals.

This Master's thesis describes the development process of a series of HW components for Intel's HARP CPU-FPGA hybrid platform, that will be used to synthesize a **Trace-Driven FPGA-Accelerated Full-System Architectural Simulator**. Essential development steps and protocols, that are required to incorporate accelerators on the HARP platform, are also highlighted. The developed modules, facilitate high-performance HW components that can accurately and efficiently simulate a highly configurable L1 Cache and 3 highly configurable Branch Predictor HW structures.

Optimal performance for the proposed HW simulator can be achieved when executed in coordination with a fast functional simulator running on SW. A state of the art API exports trace-data from the functional simulation at run time, in order to load the HW modules. Using these data, the HW modules can accurately and efficiently execute architectural simulation. Apart from **simulation results and timing statistics**, the models can generate the **system's state** at different timestamps, depending on the executed traces. **These architectural checkpoints can later be used to either validate the functionality of the components, determine the overall system's behavior using the sampling technique, execute new architectural simulations, or to warm-up other full system simulations**.

Keywords: Architectural Simulation, Trace-Driven, Sampling, Hybrid Platform, CCI-P, System-States, OPAE, HARP, AFU, ASE, FIU, HW, SW, BPs, API

Acknowledgements

First of all, I would like to thank my parents **N. Kyriakidis** and **A. Ariantzidou** for supporting me throughout my studies. I would also like to thank my colleagues **G. Pekridis** and **V. Amourgianos** with whom we worked and studied together. Furthermore, I would like to thank the **Intel Labs** for granting us access in order to develop and test on their environment (vLabs [7]). Finally, I would like to thank the Foundation for Research and Technology - HELLAS (FORTH) [8], the Pancretan Endowment Fund [9] and, of course, my supervisor **Dionisios N. Pnevmatikatos** for entrusting me with this particular research assignment and for helping me receive 3 scholarships and 2 grants:

1. **Postgraduate Student-Researcher Scholarship, FORTH, 10-12/2017**
2. **Postgraduate Student-Researcher Scholarship, FORTH, 1-6/2018**
3. **FORTH Grant to attend the ACACES 2018 summer school in Fiuggi, Italy, 8-14.7.2018**
4. **Postgraduate Student-Researcher Scholarship, FORTH, 7-12/2018**
5. **Pancretan Endowment Fund Grant, Award of Excellence, 2/2019**

Table of Contents

Chapter 1
Introduction

1.1 Background and Motivation

Accurate architectural simulation is a critical step in model-driven engineering and highly important for product certification. While prototypes can be very inefficient in terms of time and cost, full system simulation appears as a very promising candidate. Its main objectives are to **ensure the reliability and safety of newly developed platforms or products** and also **to lower the time to market by ensuring a short validation cycle**.

HW-SW co-simulation is able to simulate hardware platforms at clock and gate level. It is used to verify both the architecture and the software by simulating hardware behavior with respect to the software. However, not all simulators are capable of simulating every hardware detail, or, they do not do it intentionally. There are many possible simulation scenarios, depending on the simulator engine:

1. Simulators like OVP/Imperas [10] and DineroIV [11] 'sacrifice' accuracy for time, by executing fast functional simulation and references respectively.

2. On the other hand, gem5 [12] and SimpleScalar [13] trade time for accuracy.

3. The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor micro-architecture. It can perform both fast simulations and heavy-duty detailed timing simulations.

4. Some engines also incorporate **sampling**. Timing simulation can be a very time consuming process. The sampling technique arbitrarily selects and executes only a number of sections from an entire benchmark. The simulation results from each execution are combined to create statistics for the entire simulated program, thus massively reducing the verification/execution time.

Some key benefits of accurate HW-SW co-simulation are:

1. SW development can take place way before the actual hardware platform is ready.

2. Validation is less costly and faster because engineers can execute multiple simulations at the same time rather than sharing a single or a few HW prototypes.

3. While correct HW module initialization and internal observation can be easily inspected from a simulated environment, it is hardly done on an actual platform prototype.

4. Simulation results from multiple simulators can be combined to cross-check and validate a novel platform design.

1.2 Approach, Goals and Contributions

The main disadvantages of current full system simulators is limited throughput when simulating large systems with multiple processing units and/or peripherals. Cycle-accurate HW models written in VHDL and SystemVerilog HDLs are much too slow for HW-SW co-validation and often require a higher level of abstraction. However, the ideal scenario is to obtain 100% accuracy with realistic simulation performance, meaning that simulations need to run in minutes or hours but not days.

Trace driven simulation is a great candidate for this type of architectural simulation. It is a process performed by looking at traces of program execution or system component access with the purpose of performance prediction. **Using the sampling technique, the performance of very large applications can be accurately estimated based on the execution traces of much smaller batches**. Along this process, a system checkpoint can be exported for each batch. **Data from these checkpoints can be used for system verification, simulation and execution warm-ups and also to restart long executions processes that were either paused or stopped.**

It is hard to build a complete system from scratch due to the complexity of many components. However, this Master's Thesis describes the development of highly accurate HW modules capable of accelerating architectural HW simulation procedures while also exporting a series of valuable data, that can later be used on future simulations or to estimate the system's performance. During this research, **this series of developed modules were combined to integrate 3 standalone simulators**, that facilitate high-performance hardware components to simulate a system's Cache and Branch Prediction procedures accurately and efficiently.

Although simulator development usually takes into consideration the platform portability, a HW system is much faster if it is designed specifically for a target hardware to maximize speed and resource utilization. All 3 simulators were developed to run on Intel's **HARP CPU-FPGA** hybrid platform [14] [15] [16]. This platform was selected as it is an extremely promising piece of hardware due to its 12-core Xeon processor and integrated Arria 10 GX1150 FPGA connected with an QuickPath Interconnect (QPI) coherent link and 2 PCIe*8 links.

The **1st simulator, named 'BP Simulator',** is a combination of HW modules that can accurately simulate a highly configurable One-Level, 2-Level Adaptive or Tournament Predictor. This platform reads traces from an input trace file written in a predifined format, processes these traces and exports simulation results regarding these models.

The **2nd simulator, named 'BP+Cache Simulator',** extends the functionality of the 1st simulator by including the feature of simulating a highly configurable L1 Cache Module. Once again, the platform reads traces from 2 input trace files (one for the BPs and One for the Cache) written in a predifined format, processes these traces and exports simulation results regarding these models.

Finally, the **3rd** simulator, named **Full State Simulator**, is **an extended BP+Cache Simulator** that is capable of:

- **Exporting a system's state** containing data from each simulated memory module at a particular point in time. These data can later be used for future sampling simulations, as a system warm-up or simply to verify the contents of each memory module at a particular execution timestamp.

- Providing **simulation statistics** for each simulated component. This is a great way to monitor each module independently for evaluation and verification of correct functionality.

- Exporting **timing statistics** for the whole or parts of a benchmarking process. Partial statistics from different timestamps can be used to estimate the overall system performance (sampling), thus massively reducing the total simulation time.

- **High configurability** of the simulated components using the Verilog Header source files. The user can simulate many hardware configurations simply by altering the key parameters of the cache and the branch predictor modules.

- **Exporting Accurate and Verified Simulation Results**. All 3 simulators were validated using verified 'golden-model' simulators. This validation is thoroughly discussed in chapters 5 and 6.

Chapter 2
Related Work

2.1 Dinero IV

Dinero IV [17] [11] is a single-thread cache simulator for memory reference traces. It is **neither** a timing simulator, as there is no notion of simulated time or cycles, **nor** it is not a functional simulator, as data and instructions do not move in and out of the caches. The primary result of a Dinero IV Simulation is hit and miss information.

The Dinero IV basic idea is to simulate memory hierarchy consisting of various caches connected as one or more trees. The user can set the various parameters of each cache model, like the architecture, policy, and associativity. During initialization, the configuration to be simulated is built up, one cache at a time. For the requirements of this Master's Thesis, the Dinero IV was used to validate the custom Cache model for a large number of input traces. More details can be found in section 5.7.

2.2 SimpleScalar

SimpleScalar [13] was created by Todd Austin [18] in 2004, while he was a Ph.D. student at the University of Wisconsin in Madison. It is a system software infrastructure used to build modeling applications for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification for a range of modern processors and systems. This tool-set includes sample simulators ranging from a fast functional simulator to a detailed processor model that support non-blocking caches, speculative execution, and branch prediction.

The SimpleScalar tools are used widely in hardware research. As an example, in 2000 more than one-third of all papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs. Some other key features are visualization tools, statistical analysis resources, and debug and verification infrastructure.

SimpleScalar simulators can emulate the Alpha, PISA, ARM, and x86 instruction sets, while the host interface module permits cross-endian emulation. It also builds on most 32-bit and 64-bit versions of Linux-based and Windows-based operating systems. For the requirements of this Master's Thesis, SimpleScalar was used to validate the custom Branch Predictor models and also to generate input trace files for the verification-testing phase. More details can be found in section 6.7.

2.3 A Full System Simulator for dReDBox

The Full System Simulator for Disaggregated Computing Platforms and Cloud Data Centers [19] was developed in 2017 by the Micro-architecture and Hardware Laboratory of the Electrical and Computer Engineering School at the Technical University of Crete in Greece, as part of the dReDBox [20] disaggregated data center architecture research program.

The focus was targeted at creating a full system simulator platform, in order to test and evaluate a dReDBox server tray card before prototyping. One of the key contributions of this work is early development evaluation potential as a virtual platform model can be developed far quicker and cheaper than any hardware prototype. Another important aspect is observability, and controllability, of internal modules, complemented with accessibility, as prototype hardware designs tend to restrict the amount of testing that can be performed.

This particular work was presented at the ACACES [21] 2018 summer school poster session and was published in the session's booklet. The poster can be found in Appendix D.

2.4 FireSim

FireSim (2019) [22] is an **open-source, cycle-accurate**, FPGA-accelerated **full-system** hardware simulator that runs on Amazon's EC2 F1 [22] cloud FPGAs. It is actively under development by the Berkeley Architecture Research Group [23] of Electrical Engineering and Computer Sciences Department at the University of California, Berkeley. FireSim is open-source and can be downloaded from: **github.com/firesim/firesim**.

FireSim can simulate arbitrary hardware designs written in Chisel [24] like custom RTL processors, accelerators, etc. and run them at near-FPGA-prototype speeds, while obtaining cycle-accurate performance results (10-100MHz). A developer can also integrate custom software models for non-RTL components.

FireSim was originally developed to simulate data-centers by combining open RTL for RISC-V processors with a custom cycle-accurate network simulation. It cycle-accurately simulates RISC-V RocketChip-based clusters, with peripherals like disks and network interface cards. Also, it also provides a Linux distribution compatible with the RISC-V systems, something that automates the process of including new workloads. Simulations run fast enough to interact with Linux on the simulated system at the command line, like on a real computer. Users can even SSH into simulated systems in FireSim and access the Internet from within them.

2.5 ProtoFlex

ProtoFlex [25] is a simulation architecture that uses FPGAs to accelerate full-system multiprocessor simulation and to facilitate high-performance instrumentation. FPGA approaches are generally complex or require significant effort to provide full-system support. On the other hand, ProtoFlex virtualizes the execution of many logical processors onto a number of execution engines on the FPGA, thus eliminating the scaling shortcoming of other simulator platforms. Another key feature is called **transplanting**, a technique that allows the FPGA to implement only the frequently encountered behaviors, while a software simulator complements the simulation engine by executing all remaining behaviors.

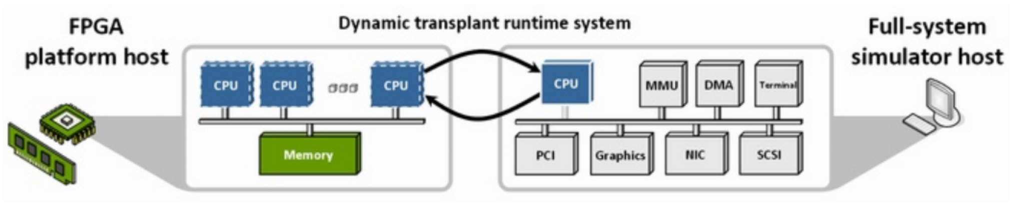


Figure 2.1 – ProtoFlex Logical Processor Virtualization and the Transplanting Technique. Image owned by [26]

Under ProtoFlex, a functional full system FPGA-based simulator called BlueSPARC was developed, that utilizes both processor virtualization and transplanting. BlueSPARC models the architectural behavior of a 16-CPU UltraSPARC III SMP symmetric multiprocessor server, hosted on a single Xilinx [27] Virtex-II XCV2P70 FPGA. Alongside BlueSPARC, a PowerPC or Microblaze runs on the FPGA to simulate unimplemented SPARC instructions (~10-50usec). On the host PC, the SIMICS full system simulator, simulates devices, memory-mapped I/O and DMAs (~500-1000usec). The outline of this simulator is shown in figure 2.2.

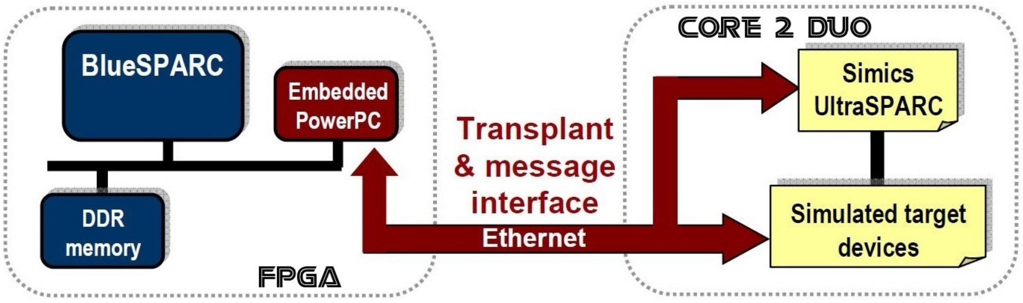


Figure 2.2 – ProtoFlex BlueSPARC Simulator. Image owned by [26]

2.6 Simics

Wind River [28] Simics creates a shared platform for software development by simulating a full target system. Simics can run unmodified target software from a physical target system, meaning the same boot loader, BIOS, firmware, operating system, board support package (BSP), middleware, and applications. It offers flexibility and customizability eliminating the restrictions of physical hardware. It is stated that it can simulate any size or complexity of a single computer or distributed system and even data and mobile networks.

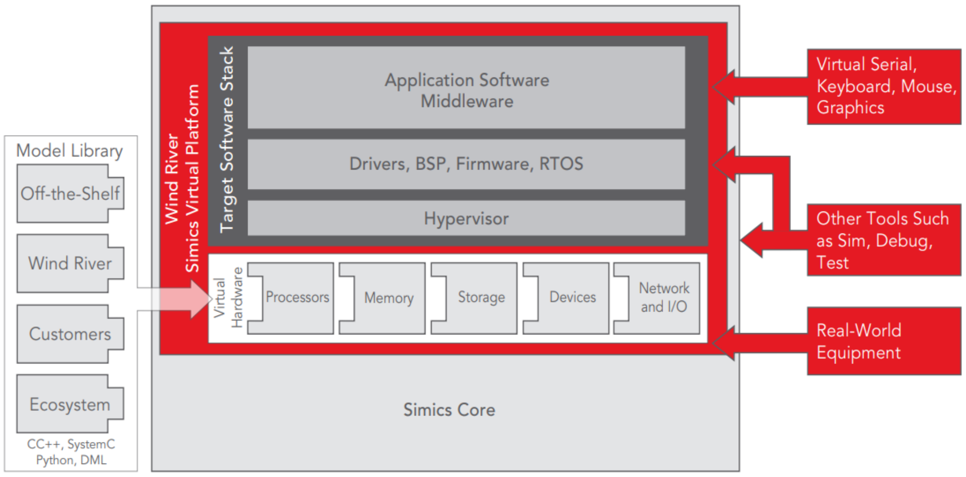

Model Library
Off-the-Shelf
Wind River
Customers
Ecosystem
CC++, SystemC Python, DML
Wind River Simics Virtual Platform
Target Software Stack
Application Software Middleware
Drivers, BSP, Firmware, RTOS
Hypervisor
Virtual Hardware
Processors
Memory
Storage
Devices
Network and I/O
Simics Core
Virtual Serial, Keyboard, Mouse, Graphics
Other Tools Such as Sim, Debug, Test
Real-World Equipment

Figure 2.3 – Simulating an entire target system using Simics. Image owned by [28]

The core of Simics is a **fast instruction set simulator** for the target processor. It also simulates memories, peripheral devices, interconnects, disks, and networks that complete a system model that is indistinguishable from a real target platform. The simulation can scale to simulate very large systems using multi-core hosts and even clusters of hosts. Because virtual platforms can be created and shipped way quicker than prototype hardware or silicon, vendors can use Simics to create multiple virtual platform models and build software to demonstrate the system's results while also providing early feedback to the developers for features and implementation, allowing changes to be incorporated into the final shipping product. An overview of a Simics full system simulation is shown in figure 2.3.

2.7 The gem5 Simulator

The gem5 [12] simulator is a modular platform for computer-system architecture research, as well as processor micro-architecture. It provides multiple interchangeable CPU models, a NoMali GPU and a fully integrated GPU model. It features a detailed **event-driven memory system** including caches, crossbars, snoop filters, and a fast and accurate DRAM controller model. It also facilitates a **trace based CPU** for memory-system performance exploration. Its main disadvantage is that despite being a very heavy-duty full system simulator it runs, almost entirely, **single threaded**.

Gem5 can simulate homogeneous and heterogeneous multi-core platforms for multiple ISAs:

- Alpha : DEC Tsunami system
- **ARM** : can boot unmodified Linux and Android
- SPARC : UltraSPARC T1
- **x86** : standard PC platform

Multiple systems can be instantiated within a single simulation process, with the user being able to extract **power** and **energy** consumption data from the detailed simulation execution. The gem5 runs on most operating systems and platform and comes under a **Berkeley-style open source license** (use anything but incorporate the copyrights). More information about this simulator can be found here [12].

2.8 Multi-ARM Architectural Simulation on HARP

This work is currently under development by the Technical University of Crete's Microprocessor and Hardware Laboratory and it complements the work described in this dissertation. The idea is to develop and integrate multiple ARM processors on the HARP hybrid platform. These processors combined with this work's BPs and Cache models will later be used to complete a hybrid full system simulator.

When completed, this platform will be able to handle 2 simulation scenarios. **First**, a QEMU-like simulator will execute fast functional simulation on the XEON CPU. For different timestamps set by the user, a state of the art SW API will extract trace data from the functional simulator in order to 'feed' the HW modules. That way, heavy-duty full system simulation can be initiated in order to export timing statistics and the system's state. On the other hand, the **second** implementation will include a QEMU-like simulator running entirely on the FPGA (ARM-Cache-BPs-Cache Model(L1-L2)) and only the complex instructions like I/O will be handled by the XEON via the transplanting technique.

With the completion of this work and the work described in this Master's Thesis, the final full system simulator will be capable of handling both of the pre-mentioned scenarios. It will also be able to provide different simulation data depending on the selected simulator implementation and also help evaluate which is the better approach in a quest to determine the best method to accelerate full system simulators using FPGAs or Hybrid Chips like HARP. The block diagram describing the finished product of the hybrid full system simulator is depicted in figure 2.4.

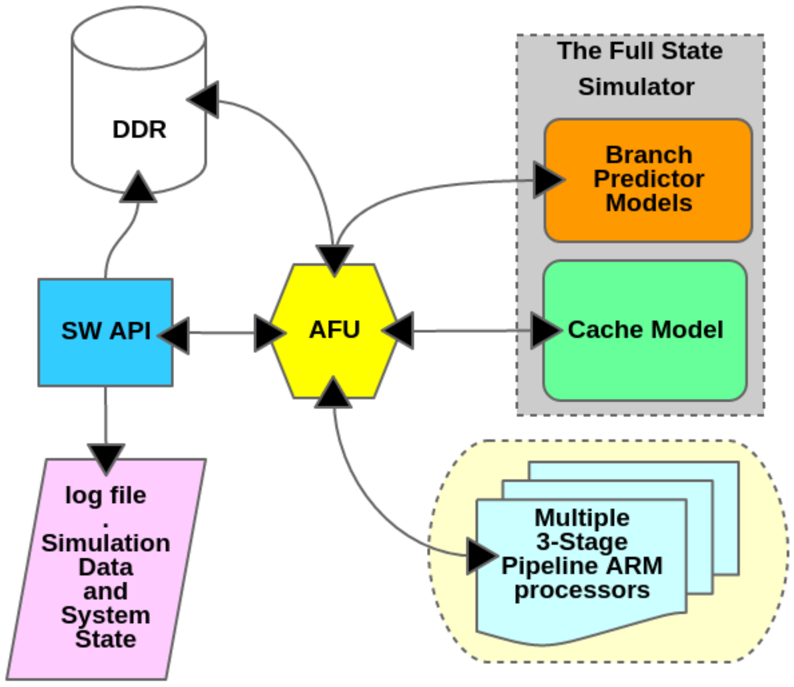


Figure 2.4 – The Complete Full-System Simulator Block Diagram.

2.9 COSSIM

COSSIM [29] is a Cyber-Physical Systems (CPS) Simulator Framework. It is the first known open-source, high-performance simulator that handles holistically system-of-systems including processors, peripherals and networks. This approach helps HW and SW developers simulate Cyber-Physical Systems (CPS) and Highly Parallel Systems (HPS).
COSSIM is built on top of several well-established simulators:

1. GEM5 [12], to simulate the digital components of each processing node.
2. OMNeT++ [30], to simulate the networking infrastructure.
3. McPAT [31], to provide energy and power consumption estimations of the processing nodes.
4. MiXiM (OMNeT++ addon), to estimate the energy consumption of the network.

To bind the whole framework together, COSSIM employs the HLA architecture through the open-source CERTI package. Additionally, an Eclipse-based GUI, provides easy simulation set-up, execution and results visualization. More information can be found on the **COSSIM Tutorial on YouTube**.

Chapter 3

The HARP Platform

3.1 The CPU-FPGA Hybrid Chip

As mentioned before, the hardware device that will be hosting the simulator described in this dissertation is **Intel's Xeon-FPGA Hybrid Chip**. The new generation of these hybrid chips is called **Xeon Scalable Processors (SP) with integrated FPGA** [15]. However, the work described in this thesis was performed on a 12-core (24-thread) **E5-2628L v4** Xeon CPU at 1.6-2.20GHz with **75W TDP** [32]. The FPGA is an Arria 10 GX 1150 [16] rated at **70W**. With **427200** Adaptive Logic Modules (ALUs), **1708800** Registers and approximately **6,3 MB** of memory, it is an ideal candidate for a custom full system simulator capable of accurately simulating large multicore target hardware designs.

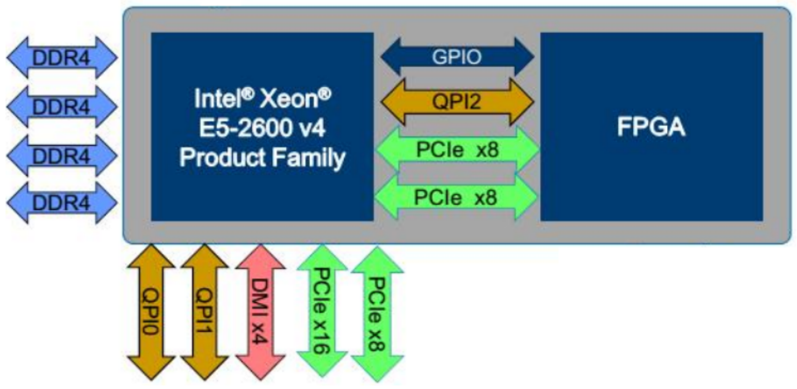


Figure 3.1 – Intel Xeon-FPGA Hybrid Chip. Image Owned by [32]

For the CPU-FPGA communication, the HARP hybrid chip utilizes a single **QuickPath Interconnect** (QPI) coherent link, as well as **2 PCIe 3.0 x8** links. The QPI is capable of **8 GT/s QPI (4000 MHz)**, meaning a unidirectional throughput of **14.9 GiB/s** (bi-directional ~30 GiB/sec), with **64 Bytes/transfer**. Using QPI, the CPU and the FPGA can be tightly coupled to each other and share the main memory. For all intents and purposes they appear like as a single processor as far as the operating system and applications are concerned. The Xeon Processor, also has two more QPI (0 and 1) channels that are used to make a two-socket system. Access to this particular platform has been granted to the Micro-architecture and Hardware Laboratory of the Technical University of Crete via Intel's vLabs [7].

3.2 The CCI-P

The **CCI-P** [33] is a host interface bus for the Accelerator Functional Unit (**AFU**) [34] with separate header and data wires. **It is used to connect the AFU to the FPGA Interface Unit** (**FIU**) within the Arria 10 GX 1150 FPGA. Custom accelerators are loaded on the AFU section, which is the FPGA's Partially Re-configurable Region (**PRR**), whereas the FIU is a **static** integration on the FPGA. Many AFU registers and wires are mandatory to design a CCI-P compliant AFU and they are all described in [33].

CCI-P implements to Main Memory (CPU Cache and DDR) and Memory Mapped I/O (MMIO) address spaces. Requests from the AFU to the main memory are called **upstream requests**. The I/O memory is implemented as CCI-P requests from the host to the AFU or the opposite, and its implementation is up to the developer. The CCI-P defines the format of each memory request. Requests to the MMIO are called **downstream** requests and an AFU's MMIO address space is fixed at 256 KB.

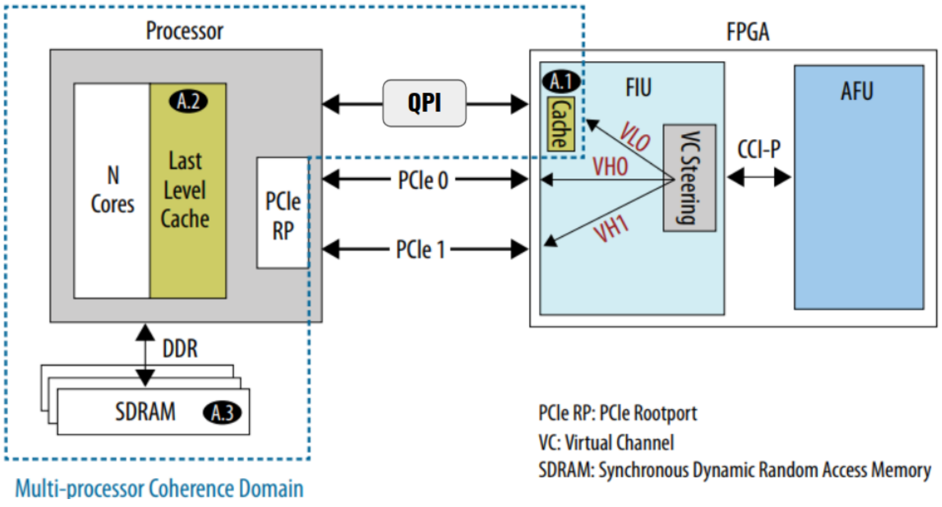


Figure 3.2 – Integrated FPGA Platform and Memory Hierarchy. Image owned by [33]

As regards to the CCI-P signals, there are 2 channels: **Tx** and **Rx**. From the AFU's point of view, Tx requests flow from the AFU to the FIU opposite to Rx that flow from the FIU to the AFU. Each CCI-P signal must be synchronous to the platform's pClk (400MHz), pClkDiv2 (200MHz) or pClkDiv4 (100MHz). It is important to note that by configuring the request header, the AFU can issue a read or write request for multiple cache-lines (512bits) using a single Rx/Tx request. Also, each CCI-P request has its own validation/confirmation response signal so that the AFU knows if this particular request has been successfully handled. Other important features like the one mentioned before can be found in the full CCI-P guide [33]. However, the use of the CCI-P requests are also described in section 7.3.1.

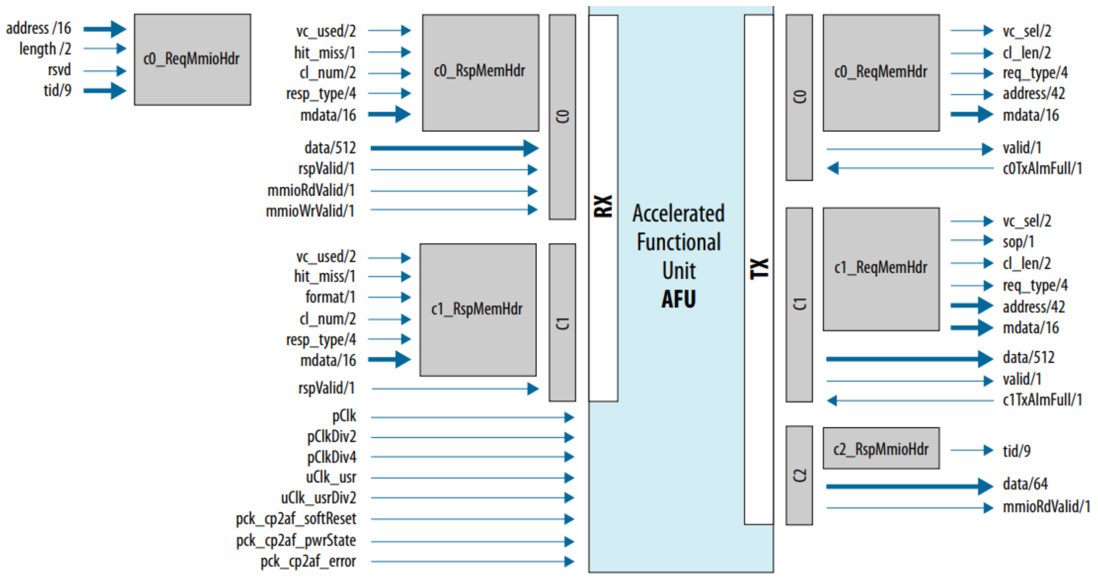


Figure 3.3 – CCI-P signals. Image owned by [33]

Chapter 4

The Tools

4.1 The OPAE

Open Programmable Acceleration Engine (**OPAE**) [35] is a software infrastructure that is used to simplify the integration of programmable accelerators, such as FPGAs, into software applications and environments. Its main goal is to accelerate FPGA adoption. It is a collection of drivers, user-space libraries, and tools to manipulate and reconfigure (re)programmable accelerators. OPAE is designed to support a layered, common programming model across different platforms and devices and is the default software stack for the Intel's Xeon-FPGA hybrid chip. It has 4 main parts:

- The OPAE Software Development Kit (OPAE SDK),
- The OPAE Linux driver for Intel's Xeon-FPGA hybrid chip with the Arria 10 GX FPGA,
- The Basic Building Block (BBB) library for accelerating the AFU development phase.
- The AFU Simulation Environment (ASE) [34] for end-to-end simulation of the accelerator's RTL together with a software application or handshake program, using the OPAE C API.

4.2 Intel's FPGA BBB

The best way to start developing on HARP is by using the Basic Building Blocks [36]. **BBB** is a suite of application building blocks (sample AFUs and SW applications) and shims for developing based on the CCI-P interface on Intel FPGAs. A Shim is a library that intercepts API calls and changes the arguments passed, handles the operation or it can redirect the operation elsewhere. The BBB repository tracks changes in the OPAE SDK and updates its blocks accordingly. Inside the BBB directory, a developer can find multiple tutorials and reference source codes on CCI-P, located inside the "samples" directory.

The BBB includes reference sample codes that developers can use or modify for their own work. All current BBB are tested with supplied examples on an Ubuntu 14.04 64-bit OS machine with an Integrated Xeon-FPGA. Also, every BBB is known to work with the OPAE AFU Simulation Environment (ASE). Several example AFUs are stored in the **intel-fpga-bbb** repository. README files are also present, describing each example, listing the contained files and providing instructions on how to build one. Furthermore, each example provides a script for setting up the ASE simulator. Finally, some examples include instructions for Quartus synthesis 4.4.

4.3 Development Phase - The ASE simulation and ModelSim

ASE is a **dual-process simulator**. Process one is responsible for running the RTL simulation, while the other, completes the AFU simulator by connecting the software with the RTL. This unified simulation significantly reduces the AFU's development time. As mentioned before, the OPAE distribution includes the ASE. This kind of dual-process simulation is depicted in figure 4.1. During the development phase, the developer utilizes C/C++ for the SW, Verilog/SystemVerilog

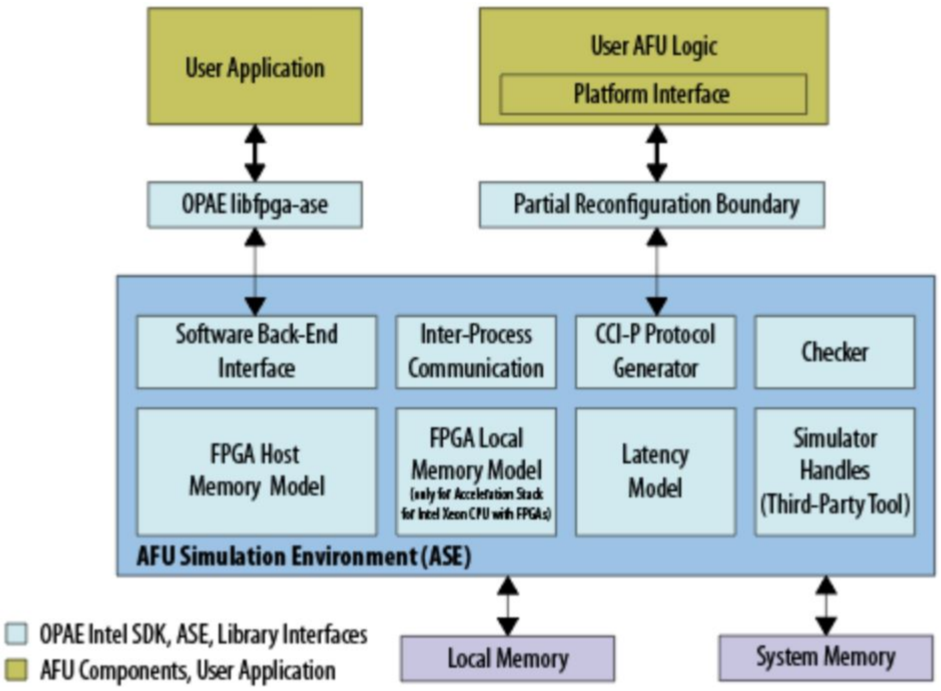

User Application
User AFU Logic
Platform Interface
OPAE libfpga-ase
Partial Reconfiguration Boundary
Software Back-End Interface
Inter-Process Communication
CCI-P Protocol Generator
Checker
FPGA Host Memory Model
FPGA Local Memory Model (only for Acceleration Stack for Intel Xeon CPU with FPGAs)
Latency Model
Simulator Handles (Third-Party Tool)
AFU Simulation Environment (ASE)
OPAE Intel SDK, ASE, Library Interfaces
AFU Components, User Application
Local Memory
System Memory

Figure 4.1 – ASE dual-process simulator overview. Image owned by [34]

for the RTL and an RTL simulator like Synopsys VCS-MX [37], ModelSim-SE [38] or QuestaSim [39]. As regards to this dissertation, **ModelSim** was the RTL simulator of choice.

In detail, ASE provides 2 interfaces to deploy a custom IP on the Integrated FPGA Platform:

- **Software**: An OPAE API implemented in the C/C++.
- **Hardware**: The CCI-P specification and the AFU implemented in SystemVerilog.

The SW and the AFU RTL are developed in a single workflow. ASE also includes a behavioral model of the FPGA Interface Manager (**FIM**) IP that provides immediate feedback on functionality during the development phase, that flags errors or warnings in the CCI-P protocol, memory accesses or simply syntax errors.

Furthermore, **the ASE presents a memory model to the AFU** that tracks memory requested as the accelerator's workspace. The memory model can inform about illegal memory transactions or requests to locations outside the memory spaces. However, it must be stated that **a correct ASE simulation does not guarantee a correct AFU synthesis**. For more information about the ASE functionality and capabilities, visit [34].

Regardless of the ASE, a developer can test custom IP cores before integrating them to the AFU, simply by using the ModelSim GUI. As this type of simulations is bare-metal, it is essential to state that before starting any ModelSim simulation, the user must first create and locate a custom testbench SystemVerilog file from the simulation's 'work' directory. This is done by selecting 'Design -> work -> <testbench_name>.sv' from the 'Start Simulation' prompt menu/window. Furthermore, from the same menu, the user must include the **altera_lnsim_ver** library by selecting 'Libraries -> Search Libraries First -> Add -> altera_lnsim_ver' 4.1.

4.4 Bitstream Generation - The Quartus Prime Pro

The Bitstream Generation phase follows the Development phase and requires the Quartus Prime Pro [6], more specifically, version **16.0.0**. Access to this Quartus, as well as the OPAE, was granted to the Micro-architecture and Hardware Laboratory of the Technical University of Crete in Greece by the Intel Labs or **vLabs** [7].

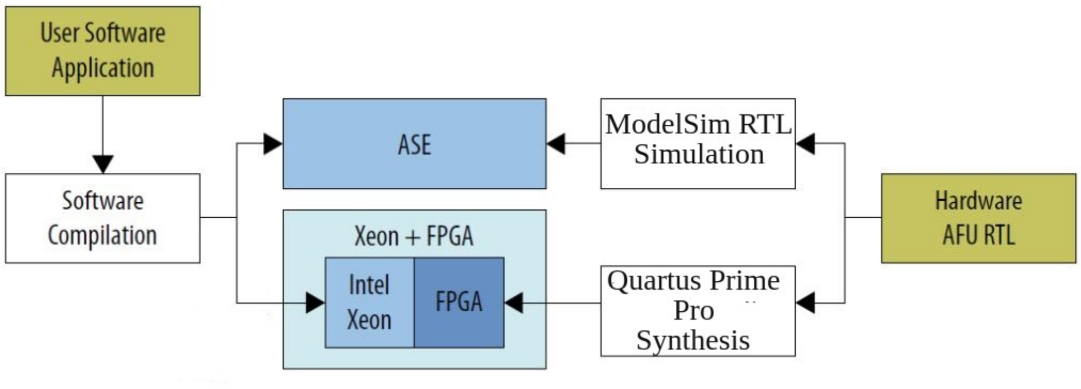


Figure 4.2 – ASE simulation and Quartus Synthesis overview. Image owned by [34]

Once AFU RTL and software are functionally correct, meaning that the simulation results from ASE are correct, the user proceeds to Quartus synthesis. The OPAE provides utilities that automate the synthesis procedure like the **'qsub-synth'** that starts the bitstream generation, place, and route using the Quartus Prime Pro. One of the reasons that made working on vLabs very efficient, was that their provided instance contained all the previously mentioned synthesis **scripts** and **utilities**, as well as the all **licenses required for the Quartus PRR bitstream synthesis**, that an AFU 'download' requires.

A bitstream generation can take hours depending on the modules, complexity, area, and architecture of the AFU. After synthesis, many **report/log files** are exported. The developer can use these files to correct problems regarding the AFU RTL or even the SW. If problems are detected, the development phase should be restarted so that the model can be re-validated. If not, a timing analysis is performed to check for setup and hold violations (slack) or clock closure (gated clocks) using the timing report/log files. At last, and when the AFU is error-free, the generated (**green**) bitstream can be downloaded. The HARP platform only allows configurations over its PRR, so the developers only generate the so-called **green bitstream or .gbs**. The second half called the **blue bitstream** is generated automatically or is already synthesized by the OPAE during the synthesizing phase.

Quartus prime pro was also used to create the Arria 10 Altera Syncram Memories and the Altera FIFOs used for the custom hybrid simulator described in this dissertation. As an example, to generate the on-chip memory module, a set of simple steps should be followed on the Quartus.v16.0.0 GUI. These steps are: 'IP Catalog -> Library -> Basic Functions -> On Chip Memory -> RAM 1 port' and finally, some parameter configurations like the target FPGA. Intel's HARP hybrid platform uses the Arria 10 GX 1150 FPGA so the developer needs to locate the model code: **10AX115U3F45E2SGE3** . Other module parameters like memory size(depth), block format(cache-line length), or FIFO size were configured in such a way so that they could be customized at compile time.

Chapter 5

The Cache Model

The cache model is a compilation of HW components that are combined in order to simulate the cache behavior inside the Full State simulator presented in this Master's Thesis. The model is highly configurable as regards to the cache size, replacement policies, and cache structure. After evaluating the most common L1 cache configurations, it was concluded that this model should be able to simulate **Direct mapped**, **2-way**, **4-way** and **8-way** structures, Write-Through (**WRT**) and Write-Back (**WRB**) Policies, as well as Least-Recently-Used (**LRU**) and Most-Recently-Used (**MRU**) Replacement Policies. This chapter contains all the information regarding this cache model's concept, architecture, functionality and verification for accurate cache simulations.

5.1 Purpose and Functionality

The Cache model was designed to meet the requirements of the future hybrid full-system simulator discussed in this Master's thesis, so, **it only store tags without any data**. It is capable of receiving **streaming address requests with interval 5** and determining cache hits and misses, just like the actual HW. The address size of each request is a user-defined simulation parameter. The cache controller is responsible for updating the cache memory, based on the user-defined cache structure and replacement policies.

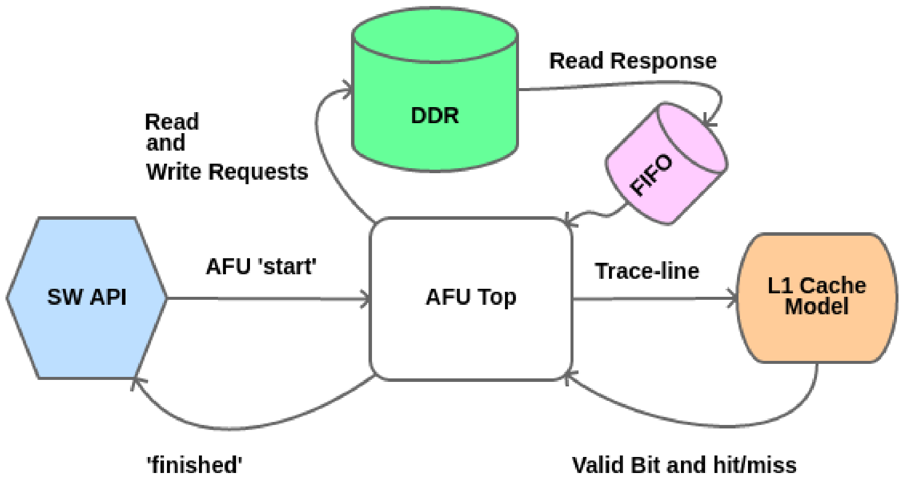


Figure 5.1 – Cache Model's Block Diagram.

For a successful L1 Cache simulation, the SW API is responsible for reading a cache input trace file and storing it on the DDR. The Cache model can then access these data via the AFU in order to process a new line from the input trace file every 5CCs. After processing the input data, the module exports simulation statistics about the simulated cache, as regards to the number of **read and write hits**. The Cache model only processes address requests to export simulation statistics, so **it can be 'called' multiple times** for as many input trace files as the user desires. When every input trace has been successfully processed, the Cache simulation is terminated.

5.2 The Cache Model's Architecture

The cache model consists of a cache controller and a single port Arria 10 Altera syncram memory, generated by the Quartus Prime Pro platform, specifically for the Arria 10 GX 1150 FPGA. These two modules are connected via a Cache Simulator top-level module. All modules are developed using Verilog HDL and SystemVerilog HDL. The cache model's top level along with all input and output signals is depicted in figure 5.2. The variable/signal names presented in this figure, are the same names used in the SystemVerilog module files and will be used to described the system throughout this chapter.

The memory module's read enable (**l1ren**) is always a logic '1' in order to read a new cache-line in each clock-cycle (CC) based on the cache address (**l1addr**) requested by the cache controller. The cache address is based on the index of the address request and has as many bits as the is index size (**DIDX_W**). **For this cache model, the index size also determines the size of the cache as regards to the number of cache-lines (memory depth)**. Based on a given cache address, a new cache-line from the single port memory module is read by the cache controller. This response is read via the **l1cacheline** registered input signal, which size varies depending on the cache-line-lenght. Note that the **l1store** registered output signal, has the same length as the **l1cacheline** and is used to update the cache's memory module. **To complete an update, the entire cache-line is loaded, edited and then exported as a new cache-line**.

For example, in a 4-way set associative cache model, the entire cache-line is loaded to the cache controller. Then, the appropriate way is tested for hit or miss and the cache-line is updated based on the replacement indexes and policy. Finally, the entire cache-line (all 4-ways and complementing bits) are offloaded to the memory module to complete a cache update through the **l1store** registered output along with a **l1wen='1'**.

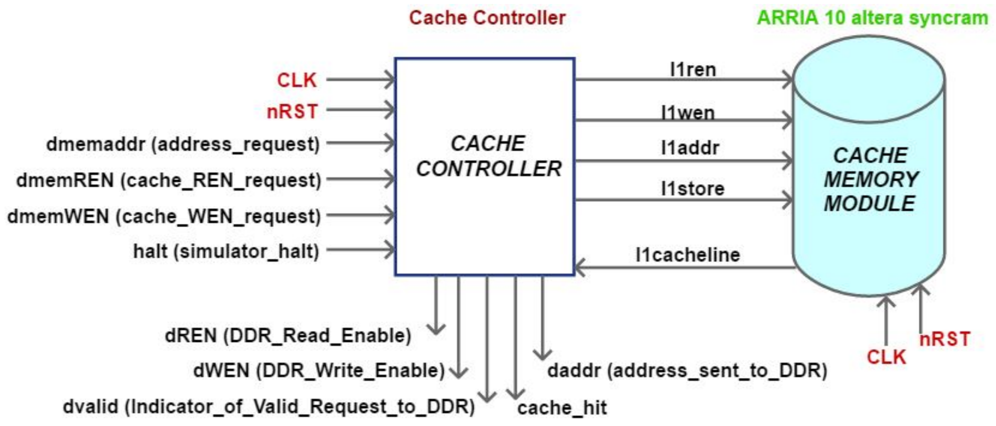


Figure 5.2 – Cache Model's Top Level. The internal signals presented in this figure have the exact same names as the ones in the Verilog and SystemVerilog files.

The cache controller determines whether the given address' tag was a cache hit or miss (**cache_hit='1'or'0'**). This address consists of a tag plus and an index. Their sizes are user-configurable using the **DTAG_W** and **DIDX_W** parameters. After determining a cache hit or miss and selecting the correct way-to-change (**explained in section 5.4**), there is a series of possible steps that can be followed based on the cache's configuration and type of the request that was issued.

5.2.1 The Cache Write Request

In all possible cases of a Cache Write Request, the cache-line is updated with the new/given address tag to the appropriate cache-way along with the valid bit set to '1', regardless of the replacement policy or cache hit/miss. Also, the cache-line index that holds the least recently used cache-way, is updated using a certain procedure inside the **WRITE_CACHE** state explained in section 5.4. Finally, the cache line index that holds the most recently used cache-way, is updated to indicate the way being updated.

1. **Cache Write Hit in Write-Through Policy**: In this case, the cache controller just has to inform about a cache write hit using the dedicated cache_hit signal.

2. **Cache Write Miss in Write-Through Policy**: In this case, the cache controller informs about a cache write miss and issues a Write Request to the DDR memory. This request consists of a DDR write enable (**dWEN=1**), a DDR valid request signal (**dvalid=1**), and a DDR address request (**daddr**), which is the same address (tag+index bits) requested from the cache controller, that resulted in a cache write miss. Note that there is no use of the cache's dirty bit in the Write-Through Policy.

3. **'Dirty' Cache Write Hit in Write-Back Policy**: In this case, the cache controller informs about a cache write hit and issues a Write Request to the DDR memory. Apart from the DDR write enable (**dWEN=1**), and DDR valid (**dvalid=1**), the DDR address request (**daddr**) consists of the cache-line's tag that was replaced by the new one, concatenated with the same index. The cache-line way's dirty bit remains '1'. Although there are no data, this cache model emulates this functionality by, at least, updating the tag to the DDR.

4. **'Clean' Cache Write Hit in Write-Back Policy**: In this case, the cache controller informs about a cache write hit and updates the cache-line way's dirty bit to '1'.

5. **'Dirty' Cache Write Miss in Write-Back Policy**: In this case, the cache controller informs about a cache write miss and repeats the same DDR write request as in the 'Dirty' Cache Hit from case 3. Although there are no data in this cache model, this simulates that in case of a dirty write miss, the cache controller needs to write-back the contents of the cache's way before updating it, otherwise, this information will be lost because it is not recorded on the DDR.

6. **'Clean' Cache Write Miss in Write-Back Policy**: In this case, the cache controller informs about a cache write miss and updates the cache-line way's dirty bit to '1'.

However, there is another case, that of a CACHE_READ_REQUEST.

5.2.2 The Cache Read Request

In all cases of a Cache Read Request, the cache-line index that holds the least recently used cache-way is updated using a certain procedure inside the **READ_CACHE** state explained in section 5.4 and the index that holds the most recently used cache-way, is updated to indicate the way being updated. The possible scenarios are:

1. **Every Cache Hit**: In this case, the cache controller just informs about a cache read hit.

2. **Cache Read Miss in Write-Through Policy**: In this case, the cache controller informs about a cache read miss and issues a Read Request to the DDR memory. This request

consists of a DDR read enable (**dREN=1**), a DDR valid request signal (**dvalid=1**), and a DDR address request (**daddr**), which is the same address (tag+index bits) requested from the cache controller, that resulted in a cache read miss. Simulating a new cache-line received from the DDR, the appropriate cache-line way is updated using the new tag and the way's valid bit is set to '1'. Once again, there is no use of the cache's dirty bit in the Write-Through Policy.

3. **'Dirty' Cache Read Miss in Write-Back Policy**: In this case, the cache controller informs about a cache read miss and issues a Write Request to the DDR memory. This request consists of a DDR write enable (**dWEN=1**), a DDR valid request signal (**dvalid=1**), and a DDR address request (**daddr**), which is the cache-line's tag that was replaced by the new one, concatenated with the same index. This simulates that in case of a dirty read miss, the cache controller needs to write-back the contents of the cache's way before updating it, otherwise, this information will be lost because it is not recorded on the DDR. Simulating a new cache-line received from the DDR, the appropriate cache-line way is updated using the new tag, the way's valid bit is set to '1' and the dirty bit is set to '0'.

4. **'Clean' Cache Read Miss in Write-Back Policy**: In this case, the cache controller informs about a cache read miss and issues a Read Request to the DDR memory. This request consists of a DDR read enable (**dREN=1**), a DDR valid request signal (**dvalid=1**), and a DDR address request (**daddr**), which is the same address (tag+index bits) requested from the cache controller, that resulted in a cache read miss. Simulating a new cache-line received from the DDR, the appropriate cache-line way is updated using the new tag, the way's valid bit is set to '1' and the dirty bit is set to '0'.

After this step, either its a read or a write request, the cache controller is ready to receive a new address request. This process can be repeated many times, in order to generate enough information required for creating a system state for the cache memory.

5.3 The Cache Memory: Size, Structure and Replacement Policies

As mentioned previously, this cache model offers great flexibility in terms of simulation options due to its customizability. The cache model described in this chapter, apart from the cache controller and the memory module, is complemented by a Verilog Header file (**cache_types _package.vh**). At compile time the user can configure the cache model's parameters that are stored in this header file. These parameters are:

Parameters

1. **MRU_OR_LRU**: This parameter is used to select the desired Cache Replacement Policy. The user can either select '0' for Most Recently Used (MRU) or '1' for Least Recently Used (LRU). Despite the fact that almost all caches utilize the LRU Policy, MRU is provided for a more complete set of simulation environment options.

2. **WRB_OR_WRT**: This parameter is used to select the desired Cache Update Policy. The user can either select '0' for Write Back Policy (WRB) or '1' for Write Through (WRT). Despite the fact that almost all caches are Write Back, Write Through is provided for a more complete set of simulation environment options.

3. **WAYS**: This parameter is used to select the cache's desired set-associativity. The user can select between Direct Mapped (or 1-Way), 2-Way, 4-Way, and 8-Way set associative cache configuration. **These parameters along with the address request tag size and index size are responsible for determining the cache's memory size at compile time**.

4. **DTAG_W**: This parameter determines the address request's tag size in bits. This is an essential piece of information as it is used not only to decode the 32-bit address request format but also for defining the data structs that determine the cache's overall size.

5. **DIDX_W**: This parameter determines the address request's index size in bits. This is yet another essential piece of information as it is used not only to decode the 32-bit address request format but also for defining the data structs that determine the cache's overall size. **Also, as mentioned before, it is used to determine the cache's overall depth (in words) as it is used as the cache address**.

6. **SIZE = (2**DIDX_W)**: This parameter depends on the predetermined **DIDX_W** and indicates the number of words (memory depth) that can fit in this particular cache memory model configuration. As mentioned before, the index is also the cache address, so the memory depth is equivalent to 2**DIDX_W words. For example, if the index size is 5 bits, the cache controller can address 2**5=32 words or cache-lines. Double '*' is used for exponent operations in SystemVerilog HDL.

7. **ADDR_W = DTAG_W+DIDX_W**: This parameter represents the size of an address request in bits.

8. **RPL_INX_SIZE = $clog2(WAYS)**: This extremely useful parameter represents the minimum number of bits that all the replacement indexes require in this model. Based on this parameter, all replacement indexes have exactly as many bits necessary in order to accurately point to the cache's way/set that is going to be updated in case of a cache read or write request.

Typedef Logic and Structs

1. **addr_t**: This is a custom variable used to hold the number of bits that synthesize a cache address request, meaning it consists of DIDX_W bits.

2. **addr_size**: This is a custom variable used to hold the number of bits that synthesize an address request, meaning it consists of ADDR_W bits.

3. **dcachef_t**: This struct represents the **components of an address request**. For that matter, it has **DTAG_W + DIDX_W** number of bits. The contents of this struct are shown in the following colorbox:

```
typedef struct packed {
logic [DTAG_W - 1:0] tag;
logic [DIDX_W - 1:0] idx;
} dcachef_t;
```

4. **dcache_entry**: This very useful struct is used to represent all the **components of a single cache way/set**. For that matter, it has **1 + 1 + DTAG_W + RPL_INX_SIZE** number of

bits. **Each way has its own valid bit, dirty bit, tag and replacement index** that indicates how recently this way has been accessed/updated. The contents of this struct are shown in the following colorbox:

```
typedef struct packed {
logic v;
logic dirty;
logic [DTAG_W - 1:0] replacement_index;
logic [RPL_INX_SIZE-1:0] replacement_index;
} dcache_entry;
```

5. **dcache_frame**: This struct represents all the **components of an entire cache-line**. For that matter, it has **WAYS*($bits(dcache_entry)) + RPL_INX_SIZE + RPL_INX _SIZE** number of bits. Each cache line contains **WAYS** number of ways, along with two replacement indexes, one for storing the MRU way and another for storing the LRU way. The contents of this struct are shown in the following colorbox:

```
typedef struct packed {
dcache_entry [WAYS - 1:0] set;
logic [RPL_INX_SIZE - 1:0] most;
logic [RPL_INX_SIZE - 1:0] least;
} dcache_frame;
```

All of the above features are 'responsible' for this cache model's customizability potential at compile time. An entire cache-line format for N-WAY set associativity is shown in figure 5.3.

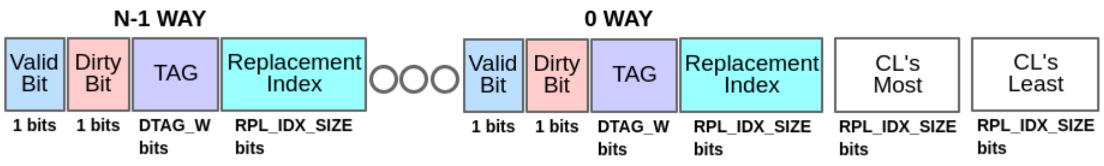


Figure 5.3 – A L1 cache-line format.

5.4 The Cache Controller

Determining if an address request is a cache hit or miss is essential for the correct functionality of a cache model, as well as selecting the correct way to access and subsequently update. The cache controller is a **7-state** FSM that is responsible for receiving and decoding an address request, fetching a cache-line from the cache memory module, checking for cache hits or misses, updating the cache's memory and issuing responses to "outside" modules.

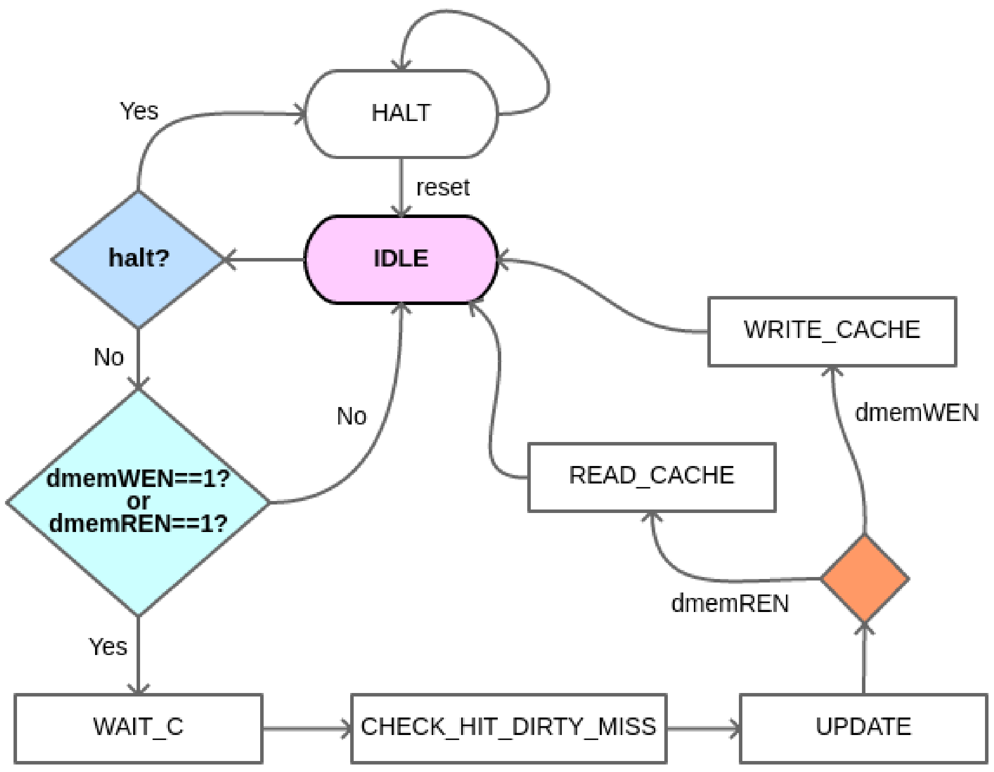


Figure 5.4 – The Cache Controller FSM

In detail, these are the 7 states of this FSM:

- IDLE: After receiving a new cache read or write request, the cache controller is responsible for fetching the appropriate cache-line based on the request's index. This cache-line, of course, will be available after 1CC. The cache controller is also responsible for storing this address request, as it will be needed in the following FSM states. If either a dmemWEN or a dmemREN signal is received (cache write or read request enables), the next state will be the **WAIT_C** state. However, if a 'halt' signal is received, the next state becomes the **HALT** state, in which the FSM remains until a reset signal is issued to the AFU. If none of the above happens, the FSM remains in the **IDLE** state.

- WAIT_C: Using this state, the FSM proceeds to state **CHECK_HIT_DIRTY_MISS** after 1CC, in order to provide time for the cache memory to respond with the correct cache-line, based on the address requested from the previous state.

- CHECK_HIT_DIRTY_MISS: In this state, the cache controller stores the received cache-line and starts a procedure to determine if this is a cache hit or miss. With the use of **'ifdef**, at compile time, the cache controller's FSM is preconfigured to support a certain set associativity. That way, there is **significant resource reduction** due to the fact that the cache model will not contain the HW to process unselected cache configurations. For better understanding, assume that the user selected a 4-way set associative cache configuration. At compile time, this cache model is configured to support **only** this 4-way set associative memory configuration.

  First, the cache controller checks all the tags of the cache-line, in order to determine if this was a cache hit. In case of a hit, a **set_to_change** register is updated with the number of the matched set in order to be accessed, updated and used in the following FSM states.

Also, the cache controller checks if this tag was a 'dirty' tag. If there are no matches, the cache controller proceeds to check each set, in order to find if there is an empty slot in this cache-line. This is done simply by checking the set's valid bit. If a set has valid = '0' the set_to_change signal is updated to match this set's number. If there are no empty slots, the cache controller updates the the set_to_change register with the correct set number based on the user-selected cache replacement policy. As mentioned earlier, this is very simple because each cache line stores the most and least recently used sets. When all these steps are completed, the FSM proceeds to state **UPDATE**.

- **UPDATE**: As mentioned in section 5.3, the replacement indexes are **RPL_INX_SIZE-bits-wide**. After determining the correct set that is to be accessed/updated, this cache-line's replacement indexes need to be updated. There are 3 possible scenarios. **First**, if the set-to-be-changed is the most recently used, the cache line indexes remains the same. **Second**, if the set-to-be-changed is the least recently used, for every cache set different from the one that was the LRU, the replacement index is **reduced by '1'**. Then, the set-to-be-changed's index is set to the maximum value **WAYS-1**. **Third**, if the set-to-be-changed is neither the MRU or LRU, the principle is the same as the second control step. However, it has the additional control that **the cache replacement index that is reduced by '1', must greater than the previous replacement index of the set-to-be-changed**. After all these checks, depending on the **dmemWEN** (cache write request) or **dmemREN** (cache read request) signal received in the **IDLE** state, the next state will either be the **READ_CACHE** or the **WRITE _CACHE**.

- **READ_CACHE and WRITE_CACHE**: Read and Write requests have been thoroughly described in subsections 5.2.1 and 5.2.2. The only thing to note is that on both states, the accessed cache-line's **.least** field is updated using the following method depending on the cache-line's valid bits and replacement indexes of each set. After the **UPDATE** state, the cache-line to be written, already contains the new/updated replacement indexes for each set. Going over all of these indexes, the cache controller finds the smallest one and updates the cache-line's **.least** field with the appropriate value/least-way.

- **HALT**: The FSM remains in this state until a reset (nRST) signal is issued.

After the READ_CACHE or WRITE_CACHE state is completed, the FSM returns to the IDLE state where it is ready to receive a new address request, and the same process is repeated all over again.

5.5 Generating Trace Files for Cache Simulations

To test this cache model, a **creator.c** program was developed that is capable of producing a random input trace file. This file is used to determine if the cache model functions properly. The trace file is nothing more than a series of address requests that this model should be able to handle in a simulation environment. Apart from the tag and the index, each address' most significant bit (MSB), indicates if this is a read or write request. **MSB==0 stands for a read and MSB==1 for the write request**.

After compiling the creator.c file using GCC, the user can execute it in order to generate two trace files, a **trace.bin** and a **trace.din**. The .bin file is used for simulating the cache model using the ModelSim simulation platform and the .din for testing on the Dinero IV Cache Simulator that will be discussed in section 5.7. For simplicity and handling reasons for the Full State Simulator's development phase, each line from the trace.bin was converted to a single 32-bit integer, using the **int32_cache_trace_converter.c** program developed as part of this research program. This converter program is included in the Full State Simulator's source files.

For this program's execution, the user must provide the desired **names** for the .bin and .din files as well as the desired **trace size** (number of addresses to generate), the **tag size** and the **index size**. All these parameters can be viewed in the help menu developed for this application by simply executing the **< $ ./creator -help>** command. As an example, in order to create a trace.bin and a trace.din file containing 10000000 addresses for a cache model that has tag_size=22 and index_size=5, the user should execute the following:

```
./creator -binfile trace.bin -hexfile trace.din -trace_size 10000000 -tag 22 -idx 5
```

5.6 Testing On ModelSim

ModelSim [38] is part of the Intel Quartus Prime Pro installation and was a great simulation environment to test this cache model properly. A new project was created on ModelSim that utilizes the generated the trace.bin file from section 5.5. This project consists of the cache model (cache controller and cache memory module) and the cache model's header file (cache_types _package.vh), which contains all the cache model's parameters.

For this model to be tested, a **cache_testbench.sv** testbench file was developed in order to import and handle all the address requests from the trace.bin file one-by-one and issue them to the cache model. After opening the trace.bin file using the **<$fopen("trace.bin", "r")>**, this testbench reads traces from the input file line-by-line and stores them on a local integer variable. The address is then split into to parts. The MSB, as mentioned before, is used to determine if this is a read or write request. The following bits are the tag and the index, which are directly forwarded to the cache controller. Depending on the MSB, a dmemREN or dmemWEN is issued at the same clock cycle (CC).

After 5 CCs, a **cache_valid** signal indicates that the **cache_hit** signal (cache controller output) has the 'correct' value for this particular address request. This signal is evaluated and depending on its value, the appropriate **read_hit** and **write_hit** counters are updated. Then, the testbench loads a new address from the trace file and repeats this process until the end-of-file is reached (**<$feof(input_file)>**). Helpful simulation messages like 'Read Hit', 'Read Miss', 'Write Hit', 'Write Miss', 'Total Read Hits' and 'Total Write Hits' are present in order to help with evaluation and debugging.

For reference, in a 4-way set associative LRU cache configuration with tag_size=22 and index_size=5, for a trace.bin file that consisted of 10000000 addresses, the simulated cache model worked flawlessly and delivered a final result of 6 Read Hits and 5 Write Hits, exactly the same as the result of the equivalent cache model simulated using the Dinero IV Cache Simulation platform described in section 5.7. Needless to say that many possible cache model configurations were tested to verify the correct functionality of this model.

5.7 Hardware Verification Using the Dinero IV Cache Simulator

Dinero IV [17] is a Trace-Driven Uniprocessor Cache Simulator. It is not a timing simulator. There is no notion of simulated time or cycles, only references. It is mainly used to gain insights into replacement policies and set-associativity of single or multi-level caches. Dinero IV is a verified cache simulator, so it was used in association with ModelSim in order to cross-check the simulation results.

The Dinero IV installation is a simple **<$ gunzip d4-7.tar.gz.>** unzip command. After this step, the user can simulate any desired cache model configuration using the trace.din file generated as described in section 5.5. **Over 30 possible configurations were executed** to validate the Cache module in comparison to the ModelSim bare-metal results. For reference, in order to test all **10000000** traces from the generated **trace.din** file on a the 4-way set-associative L1 cache model with **size**=4096 bytes and **block-size**=32 bits and print the output on a **log.out** file, the user should simply execute:

```
$ ./dineroIV -l1-dsize 4096 -l1-dassoc 4 -l1-dbsize 32 -informat d <trace.din > log.out
```

This results in 6 Read Hits and 5 Write Hits, exactly the same as the ModelSim equivalent. By comparing results from various cache model configurations tested on ModelSim with their Dinero IV equivalents, it was verified that this cache model, build for the hybrid CPU(XEON)-FPGA(ARIA10GX) platform, can accurately simulate each and every one of the supported cache model configurations.

It is noted that the simulation results on the Dinero IV are available much faster than on ModelSim. This is due to the fact that ModelSim simulates the entire custom Hardware and because the developed ModelSim testbench is not ideal, with clock cycle delays and possibly unnecessary log printing, whereas Dinero IV simulates only references.

Chapter 6

The Branch Prediction Models

Every modern processor uses branch prediction so it is essential in a full system simulator platform. The Full State Simulator described in this Master's thesis, can implement 3 branch predictor models: **One-Level** 6.3 (n-bit or BIMOD), 2**-Level Adaptive** 6.5 (GAg, PAg, and PAp) and **Tournament** 6.7 (Combination of N-bit and a 2-Level branch predictor). A Branch Target Buffer (**BTB**) model is included in each predictor implementation. Every predictor model is highly configurable as regards to its memory table sizes, predictor size, internal parameter configuration, and functionality in order to simulate many architectures. The user may define parameters such as the PC size, the opcode size, the predictor memory's word-length and overall depth, the size of the n-bit predictors and of course the length of a line from the input trace-file that enters the Branch Predictor for simulation. All models are capable of accurately simulating actual hardware behavior in order to deliver a solid internal system state and simulation statistics after an input trace is processed.

6.1 Purpose and Functionality

The purpose of all branch prediction models is to accurately simulate the behavior of this system as an actual hardware implementation. The versatility of these particular models resides on the fact that there are a lot of user-defined parameters, meaning that many hardware configurations can be implemented/simulated. All modules are developed using the SystemVerilog HDL. The simulation's end product is a **branch predictor state** that includes branch prediction **statistics** (address and direction hits and misses) as well as all the **memory instances**, meaning the branch target buffers, branch history tables, pattern history tables, and history shift registers. However, this state's extraction will be discussed in a different chapter. **Address hits occur in case of a correct prediction from the Branch Target Buffer, while direction(taken or not-taken) hits occur when the model would have correctly predicted a taken or not taken branch**. All this information will be combined with other partial simulation output files in order to form an entire **system state** for a particular number of instruction, also known as a checkpoint. Of course, as mentioned earlier, many of these "partial" states can be exported throughout a single simulation.

Each model receives one line at a time from a .txt trace-file as input. **Each trace-line consists of information about a single branch instruction in the form of a 64-bit integer**. With this information, the branch predictor simulator, based on its hardware configuration, begins to process this trace-line just as the actual branch prediction hardware would process a branch type instruction. **Apart from the branch's PC, Opcode and Target address, these information must specify if the branch was actually taken or not.** After processing all the pieces of information combined with the simulator's Branch Target Buffer and Predictor's state, the model determines if this would have been a correct or incorrect address or direction prediction. It then updates the model's memories accordingly and moves to process the next trace-line. This procedure continues until each line from the input trace file has been processed. After that, a branch predictor state can be exported, containing all the information about this particular set of executed instructions. More information about the input trace file is stated in section . All the user configured parameters for all branch predictor models are specified in a header file called **branch_header.vh**, that complements each branch predictor model.

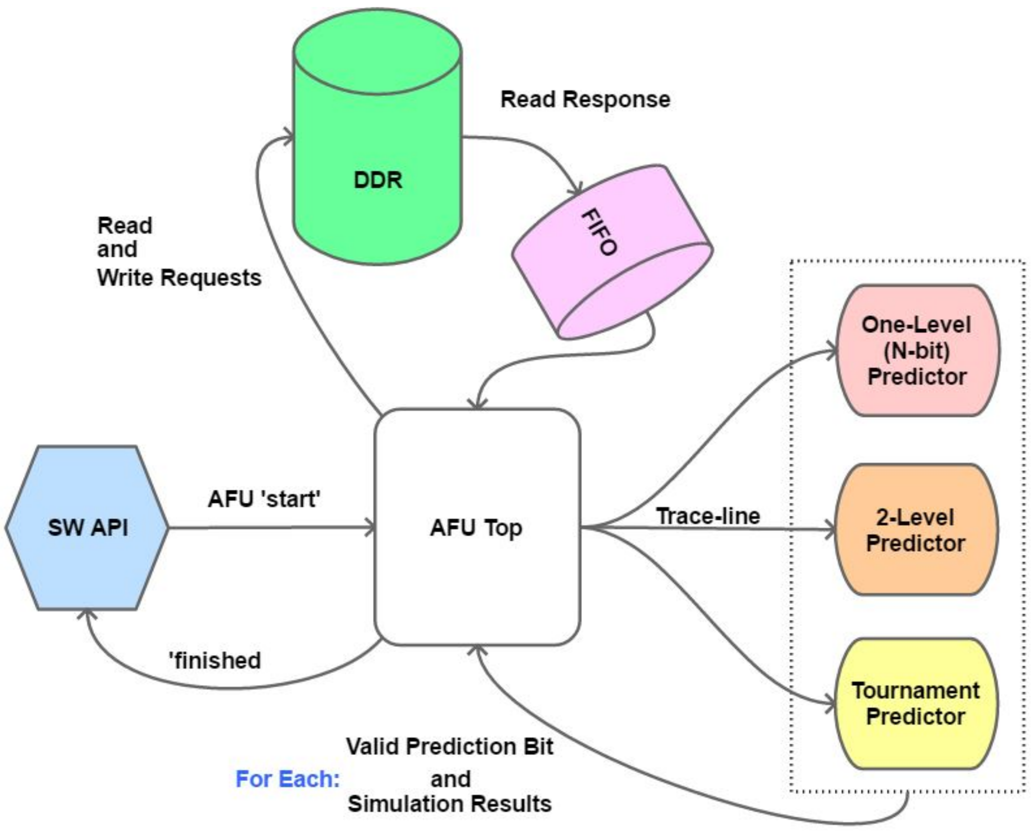


Figure 6.1 – General Block Diagram showcasing the AFU, the SW API and all the BP models that are available in the Full State Simulator

6.2 The Input Trace File

To run the branch predictor simulator, an input trace file is required. This is a binary file that consists of multiple trace-lines. The size of the input file depends on the number of branch instructions required for a BP state, or are exported from a benchmark execution and it is yet another user-defined value. Each trace-line contains information about a single branch instruction. These pieces of information are the branch's **PC**, **Target PC**, **opcode**, and the information about whether this branch would have been "actually" **taken or not taken**. These parameters' sizes are all user-defined and determine the overall trace-line length such as this:

parameter TRACE_LINE_LENGTH = PC_SIZE + PC_SIZE + 1 + OP_SIZE

Starting from the left-hand side, the first **PC_SIZE** bits are the branch's PC. The next **PC_SIZE** bits are the branch's target PC, that the program's PC will jump to if this branch is a "taken" one. The next bit displays whether this particular branch is actually taken or not taken. This information comes after processing the branch parameters. Finally, the last **OP_SIZE** number of bits is the branch's opcode. This is a crucial piece of information that determines the functionality of any branch predictor. For example, if the branch's opcode suggests that this is a "jump" instruction, the branch predictor would always correctly predict that this branch is taken and so, the internal components should be updated accordingly. More information about the input trace files that were used to test all the branch predictor models can be found in sections 6.7 and 6.8 explaining how all models were verified using the SimpleScalar simulator platform.

6.3 The Branch Target Buffer

The Branch Target Buffer (BTB) behaves the same for all kinds of branch predictor models, so it is best to describe it separately. The BTB memory consists of words containing information **only** about a taken branch's opcode and target address. The BTB memory size or number-of-words is calculated using the number of memory addressing bits. The BTB word format and memory size are shown below:

```
parameter BTB_SIZE = OP_SIZE + PC_SIZE;
parameter MEM_DEPTH = (2**IDX_SIZE);
```

OP_SIZE and **PC_SIZE** are user-defined parameters. The first **OP_SIZE** bits are the taken branch's opcode and the last **PC_SIZE** bits are the taken target PC. **IDX_SIZE** is also user-defined and indicates how many bits are used to address the BTB and other memory modules. So, with IDX_SIZE bits, (2****IDX_SIZE**) number of words can be addressed. When a new input trace-line enters the simulator, the branch predictor's controller is responsible for reading the correct BTB memory word in order to then evaluate if this would have been an address hit or miss. Addressing the BTB memory module is done like this:

```
btb_address <= pc » 3;
```

This addressing method is the same as the one used in the predictor models of SimpleScalar, described in section 6.7 and only uses the branch's PC, right-shifted by 3 bits. The reason this shift occurs is that the PC is increased by a factor of 4 in every clock cycle, so the 2 LSBs of the PC are always the same (zeros).

A correct BTB address prediction happens on 3 occasions. **First**, a BTB hit occurs when the branch is actually a jump instruction. In this case, the model always predicts correctly that the branch is taken and the new PC will be the target that comes with the instruction. **Second**, if the predictor correctly predicts a taken branch and the target address from the BTB matches the trace-line's target. **Third**, when the predictor correctly predicts a not-taken branch, the predicted PC will have the correct value PC + 4. Every other case leads to BTB address miss.

In case of a BTB address hit or in case of a jump instruction, there is no need to update the BTB memory. **It is important to note that the BTB is updated only for a taken branch**. So, the only time the BTB is updated is when a taken branch was mispredicted. In that case, on the same requested address, the BTB memory is updated with the new trace-line's opcode and target address. This update method matches the BTB update policy of the SimpleScalar described in section 6.7.

6.4 The One-Level (n-bit or Bimod) Predictor

The One-Level, or n-bit branch predictor, consists of a Branch Target Buffer (**BTB**) and a Branch History Table (**BHT**) memory module, managed by the n-bit predictor's control module. First, the user specifies the simulated hardware's parameters inside the **branch_header.vh**. All the parameters that the n-bit predictor needs to determine the BTB and BHT memory sizes, as well as the addressing bits, data inputs and outputs, are shown in the colored box below. Each parameter's name is self-explanatory. However, a short description is added on their right-hand side, while many of them have already been analyzed on previous sections. Of course, this header file alters between the different types of predictors.

```
parameter PC_SIZE = 32; // PC length
parameter OP_SIZE = 5; // Opcode length
parameter IDX_SIZE = 10; // Number of bits used to address the BTB and BHT
parameter MEM_DEPTH = (2**IDX_SIZE); // number of words in BHT and BTB
parameter BTB_SIZE = OP_SIZE + PC_SIZE; // BTB memory line length
parameter TRACE_LINE_LENGTH = PC_SIZE + PC_SIZE + 1 + OP_SIZE; // sec:6.2
parameter N_BIT = 2; // size of the n-bit predictors in the BHT
```

The memory modules used to implement the BTB and BHT are both ARRIA 10 Altera syncrams, generated by the Quartus Prime Pro platform specifically for the desired target hardware. Each of them is a **single port syncram component**. These memory modules along with their input and output signals are depicted in figure 6.2. The BTB's functionality has already been described in section 6.3. As for the BHT memory, it contains **MEM_DEPTH** number of n-bit predictors. **By knowing the MSB of a BHT memory's word, the controller decides whether to predict taken (MSB == 1) or not taken (MSB == 0).**

6.4.1 Module Architecture

The n-bit predictor's controller is a **3-state FSM**. The only time this FSM leaves the first state is when the predictor model receives a new valid trace-line. **This is possible via a valid-bit provided by an outside component alongside a new input trace-line**. When a new valid trace-line enters the simulator, the n-bit predictor's controller reads the corresponding words from the BTB and BHT, using the trace-line instruction's PC to address these two memory modules. The request addresses are the following:

btb_address <= pc » 3; **bht_address <= (pc » 19)&(pc » 3);**

The addressing method of the BTB is described is section 6.3 along with the right-shifting of the PC by 3 bits. As regards to the BHT's addressing technique, the shifting principle remains the same. However, the bitwise AND between the (pc»3) and the branch's PC right-shifted by 19 is the same used to address the BHT on the SimpleScalar predictor models. Meanwhile, the trace-line's pc, taken-not-taken, target address, and opcode information are stored for processing in the next FSM state.

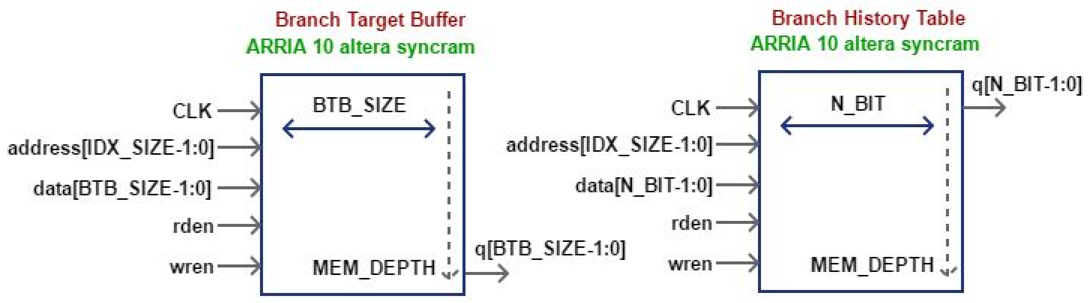


Figure 6.2 – The BTB and BHT ARRIA 10 Altera syncram memory modules

The 2nd state is a wait state for the memory responses. In the 3rd FSM state, the fetched BTB and BHT words are evaluated along with the information from the trace-line in order to determine if this would have been a correct or incorrect address or direction prediction. This state is also responsible for updating the BHT and BTB tables, based on their previous state and predictor outcome (hit/miss). The updating of the BTB as well as how the predictor model determines an address hit or miss are already mentioned in section 6.3. So now, the focus is shifted on how the model predicts taken or not-taken, whether this prediction was correct and how does it update the BHT memory. There are **7 possible cases**. In these cases, **bht_data** represents the updated word that will be written to the BHT, whereas **bht_q** is the word that was read from the BHT using the **bht_address**.

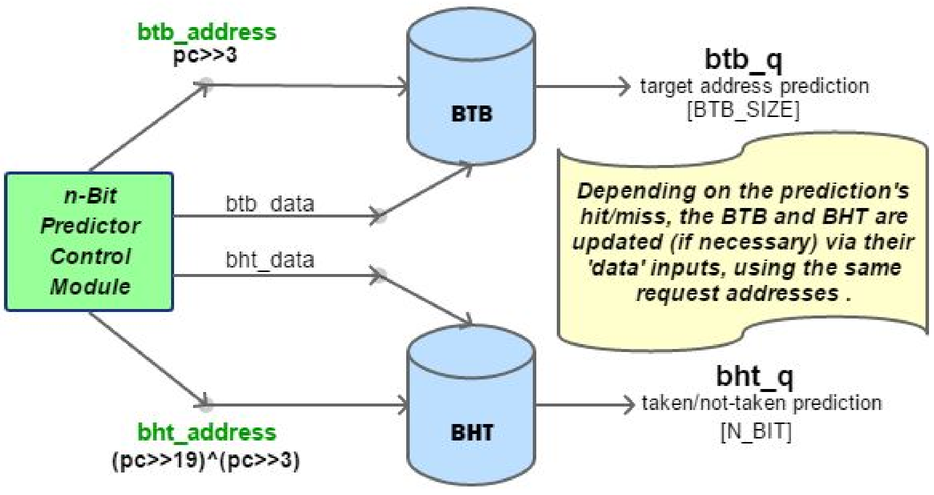

btb_address
pc>>3
BTB
btb_q
target address prediction
[BTB_SIZE]
n-Bit Predictor Control Module
btb_data
bht_data
Depending on the prediction's hit/miss, the BTB and BHT are updated (if necessary) via their 'data' inputs, using the same request addresses .
BHT
bht_q
taken/not-taken prediction
[N_BIT]
bht_address
(pc>>19)^(pc>>3)

Figure 6.3 – The N-bit branch predictor model

1. **Case 1 - Jump instruction:** The branch's jump-opcode is known to the predictor. If it detects a jump instruction, the direction prediction is always correct because all jumps are taken. In this case, the BHT is not updated.

2. **Case 2 - BHT word is all-zeros and the branch is not taken:** In this case, the n-bit predictor would have predicted a not taken branch, so, the direction prediction is correct. Because the BHT's predictor bits are all zeros, there is no need to update the BHT as it is already at a strong-not-taken state.

3. **Case 3 - BHT's word MSB==0 and the branch is not taken:** Same as in case 2 with the direction prediction being correct. Only now, the BHT's n-bit predictor is updated by subtracting 1 from its previous value such as this: bht_data <= bht_q **-** 1. This is done in order to be moved closer to the strong-not-taken state.

4. **Case 4 - BHT's word MSB==0 and the branch is taken:** Here, the n-bit predictor would have predicted a not taken branch, meaning that the direction prediction is incorrect. In this event, the BHT's n-bit predictor is updated by adding 1 to its previous value such as this: bht_data <= bht_q **+** 1. This is done in order to be moved closer to the strong-taken state.

5. **Case 5 - BHT's word is all-ones and the branch is taken:** In this case, the n-bit predictor would have predicted a taken branch, so, the direction prediction is correct. Because the BHT's predictor bits are all ones, there is no need to update the BHT as it is already at a strong-taken state.

6. **Case 6 - BHT's word MSB==1 and the branch is taken:** Same as in case 5 with the direction prediction being correct. However, the BHT's n-bit predictor is updated by adding 1 to its previous value in order to be moved closer to the strong-taken state.

7. **Case 7 - BHT's word MSB==1 and the branch is not taken:** The final state where the predictor incorrectly predicts a taken branch. The direction prediction is incorrect and the BHT's n-bit predictor is updated by subtracting 1 from its previous value, just like the BHT update in case 3;

These steps conclude the workflow of the n-bit predictor's FSM, that returns to its first state **after 3 clock cycles (3CCs)**. There, it idles, until a new valid trace-line enters the branch predictor simulator. An overview of the N-bit predictor along with all of its major components, key parameters and procedures are depicted in figure 6.3

6.5 The 2-Level Predictor

The 2-Level branch predictor consists of a Branch Target Buffer (**BTB**), a Shift Register (**SR**) and a Pattern History Table (**PHT**) memory module, managed by the 2-Level predictor's control module. Just like in the n-bit, the user specifies the desired hardware's parameters inside the **branch_header.vh**. All the parameters that the 2-Level predictor needs to determine the BTB, SR and BHT memory sizes, as well as all the addressing bits, data inputs and outputs, are shown in the colored box below.

The parameter names are self-explanatory with a short description added on their right-hand side. The first six parameters are the same as the ones in the n-bit predictor's header file. Parameter **GAP** is used to determine the type of predictor model that is going to be simulated. If the user sets this parameter to '1' the predictor becomes a Global Adaptive predictor (**GAg**), meaning that the SR memory will only have one history register of size **SR_SIZE**. The **SR_IDX** represents the number of bits that will be used to address the SR memory. This parameter also determines the number of words and thus the size of the SR memory which is the parameter **N**. If the size of the SR memory equals that of the PHT's, the predictor becomes a Per-Branch Adaptive predictor (**PAp**). In any other case, it is just a general case 2-Level Adaptive predictor with N number of global shift registers (**PAg**). The PHT is just like the Branch History Table (BHT). By knowing the MSB of a PHT memory's word, the controller decides whether to predict taken (MSB == 1) or not taken (MSB == 0). However, it is called that way because it is not addressed directly by the PC but the appropriate SR's shift register. The PHT contains **PHT_MEM_DEPTH** words that are k-bit predictors of size **K_BIT**.

```
parameter PC_SIZE = 32; // PC length
parameter OP_SIZE = 5; // Opcode length
parameter IDX_SIZE = 10; // Number of bits used to address the BTB
parameter MEM_DEPTH = (2**IDX_SIZE); // number of words in the BTB
parameter BTB_SIZE = OP_SIZE + PC_SIZE; // BTB memory line length
parameter TRACE_LINE_LENGTH = PC_SIZE + PC_SIZE + 1 + OP_SIZE; // sec:6.2
parameter GAP = 0; // '1' if the user wants a global shift register
parameter SR_SIZE = 10; // history register's number of bits/SR memory line length
parameter SR_IDX = 2; // number of bits used to address the SR memory
parameter N = (2**SR_IDX); // number of shift registers/number of words in the SR memory
parameter PHT_MEM_DEPTH = (2**SR_SIZE); // number of words in the PHT
parameter K_BIT = 2; // size of the n-bit predictors in the PHT
```

The memory modules used to implement the BTB, SR and PHT memories are all ARRIA 10 Altera syncrams, generated by the Quartus Prime Pro platform specifically for the desired target hardware. Each of them is a **single port syncram component**. The BTB syncram is shown in figure 6.2, so, figure 6.4 depicts only the SR and PHT memory modules along with their input and output signals.

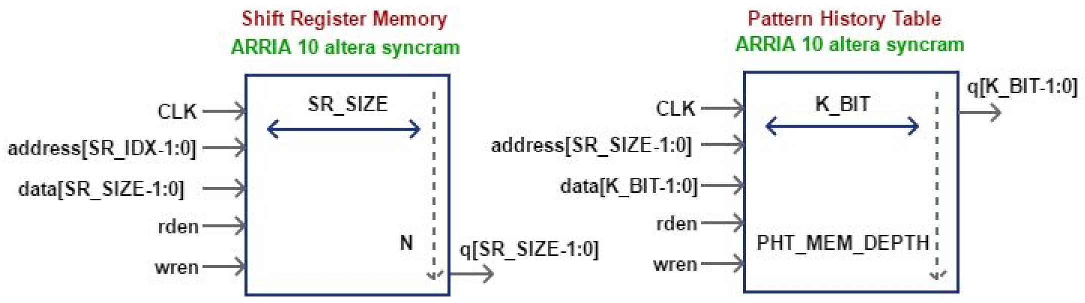


Figure 6.4 – The SR and PHT ARRIA 10 Altera syncram memory modules

The BTB's functionality has already been described in section 6.3. As for the SR memory, it holds N number of history shift registers that are used to address the PHT. **After each new branch, unless it is a jump, the appropriate shift register is left shifted by one bit and its LSB is replaced with '1' if the new branch instruction was taken or '0' if it was not taken.** Hence, it gets the name history register because it actually represents the pattern that a branch instruction was either taken or not-taken throughout a benchmark simulation or number of executed instructions in general.

6.5.1 Module Architecture

The 2-Level predictor's controller is a **5-state FSM**. The **2nd** and **4th** state are just 'wait-states'. The only time this FSM leaves the first state is when the predictor model receives a new valid trace-line just like the one described in the previous section. When a new valid trace-line enters the simulator, the 2-Level predictor's controller reads the corresponding word from the SR memory, using the instruction's PC for addressing. The requested address is the following:

IF parameter GAP = '1' : sr_address <=0; ELSE sr_address <= (pc » 3);

The addressing method for the SR is the same as the one used for the BTB. Although the BTB read request can happen in this state, it is done in the 3rd for simplicity, because the predictor still needs to get the appropriate history register from the SR memory in order to address the PHT. Thus, the BTB word would be of no use in the second FSM state. Meanwhile, the trace-line's pc, taken-not-taken, target address, and opcode information are stored for processing in the next FSM states.

In the 3rd state, the fetched history register is ready for use on the **sr_q** registered input. Only when the instruction is not a jump, the fetched shift register is left shifted by one bit and its LSB is replaced with '1' if the new branch instruction was taken or '0' if it was not taken. That is because **a jump is always taken and has no significance to a pattern history register**. Then the SR memory is updated using the **sr_data** output signal on the same **sr_address** used in the first FSM state. After 1 CC, the 2-Level predictor's controller reads the corresponding words from the BTB and PHT, using the trace-line instruction's PC and fetched **sr_q** to address these two memory modules. As regards to the PHT's addressing technique, it is the same used to address the PHT on the SimpleScalar predictor models. The request addresses are the following:

btb_address <= pc » 3; pht_address <= (sr_q | ((pc » 3) « SR_SIZE));

In the **5th** and final FSM state, the fetched BTB and PHT words are evaluated along with the information from the trace-line in order to determine if this would have been a correct or incorrect address or direction prediction. This state is also responsible for updating the PHT and BTB tables, based on their previous state and predictor outcome (hit/miss). The updating of the BTB as well as how the predictor model determines an address hit or miss are already mentioned in section 6.3. Furthermore, the 2-Level predictor model predicts taken or not-taken and updates the PHT exactly the same way as the N-bit predictor handles information and updates the BHT. The same 7 possible cases described in subsection 6.4.1 for a correct prediction, misprediction and BHT update, apply for the PHT and the 2-Level predictor in general. The only things that differ are the different input and output signal names and case names. **pht_data** represents the updated word that will be written to the PHT memory, whereas **pht_q** is the word that was read from the PHT memory using the **pht_address**. For this reason, there is no need to rewrite all 7 cases, but only to inspect the 7 cases of subsection 6.4.1 simply by replacing the <bht> letters with <pht>. After all, **the PHT is a BHT that is addressed in a different way**.

These steps conclude the workflow of the 2-Level predictor's FSM, that returns to its first state **after 5 clock cycles (5CCs)**. There, it idles, until a new valid trace-line enters the branch predictor simulator. An overview of the 2-Level predictor along with all of its major components, key parameters and procedures are depicted in figure 6.5

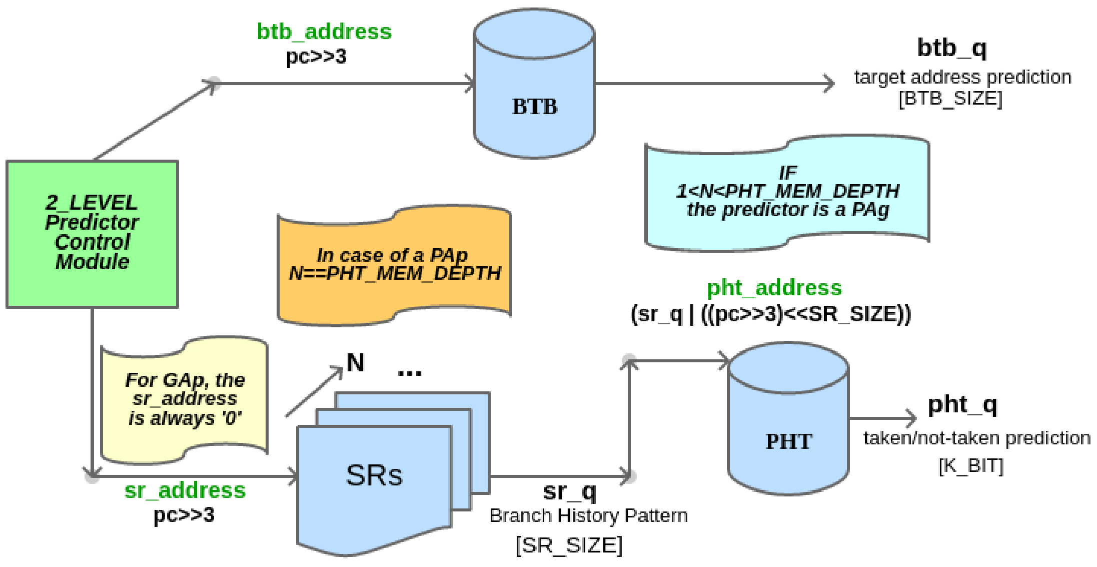

btb_address
pc>>3
BTB
btb_q
target address prediction
[BTB_SIZE]
2_LEVEL Predictor Control Module
IF 1<N<PHT_MEM_DEPTH the predictor is a PAg
In case of a PAp N==PHT_MEM_DEPTH
pht_address
(sr_q | ((pc>>3)<<SR_SIZE))
For GAp, the sr_address is always '0'
N
...
SRs
PHT
pht_q
taken/not-taken prediction
[K_BIT]
sr_address
pc>>3
sr_q
Branch History Pattern
[SR_SIZE]

Figure 6.5 – The 2-Level branch predictor model

6.6 The Tournament Predictor

The Tournament branch predictor combines the two previously described predictor models. A One-Level (n-bit) and a 2-Level branch predictor work **independently** side by side, in order to provide two separate predictions. A second **One-Level-type predictor called the Meta-Predictor (SEL memory)** decides which of these two predictions will be chosen as the final Tournament prediction. The BTB as mentioned in section 6.3 works separately to provide an address prediction. The Tournament predictor consists of a Branch Target Buffer (BTB), a Branch History Table (BHT), a Shift Register (SR), a Pattern History Table (PHT) and a Selector (SEL) memory module, managed by the Tournament predictor's control module.

Just like in the other predictor models, the user specifies the desired hardware's parameters inside the branch_header.vh. All the parameters that the n-bit predictor, the 2-Level predictor, and the Meta-Predictor need to determine the SEL, PHT, BTB, SR and BHT memory sizes, as well as all the addressing bits, data inputs and outputs, are shown in the colored box below:

```
parameter PC_SIZE = 32; // PC length
parameter OP_SIZE = 5; // Opcode length
parameter IDX_SIZE = 10; // Number of bits used to address the BTB memory
parameter MEM_DEPTH = (2**IDX_SIZE); // number of words in the BTB memory
parameter BTB_SIZE = OP_SIZE + PC_SIZE; // BTB memory line length
parameter TRACE_LINE_LENGTH = PC_SIZE + PC_SIZE + 1 + OP_SIZE; // sec:6.2
parameter N_BIT = 2; // size of the n-bit predictors in the BHT memory
parameter GAP = 0; // '1' if the user wants a global shift register
parameter SR_SIZE = 10; // history register's number of bits/SR memory line length
parameter SR_IDX = 2; // number of bits used to address the SR memory
parameter N = (2**SR_IDX); // number of shift registers/number of words in the SR memory
parameter PHT_MEM_DEPTH = (2**SR_SIZE); // number of words in the PHT memory
parameter K_BIT = 2; // size of the n-bit predictors in the PHT memory
parameter S_BIT = 2; // size of the n-bit predictors in the SEL memory
```

All the parameters have already been described in the previous sections and have a short description added on their right-hand side. The only new parameter is the **S_BIT** that determines the size of the s-bit predictors in the **SEL** memory used by the Meta-Predictor. The size of the SEL memory has been chosen as equal to the one of the BTB so it has no separate parameter.

The memory modules used to implement the BTB, BHT, SEL, SR and PHT memories are all ARRIA 10 Altera syncrams, generated by the Quartus Prime Pro platform specifically for the desired target hardware. Each of them is a **single port syncram component**. All memory models have been explained in the previous sections and displayed in figures 6.2, and 6.4, except for the Selector memory module that is displayed along with its input and output signals in figure 6.6. **The SEL memory is nothing but a BHT for the Meta-Predictor and performs exactly the same. The only difference is that its predictors decide between predictor zero (One-Level) or predictor one (2-Level) rather than taken or not-taken**.

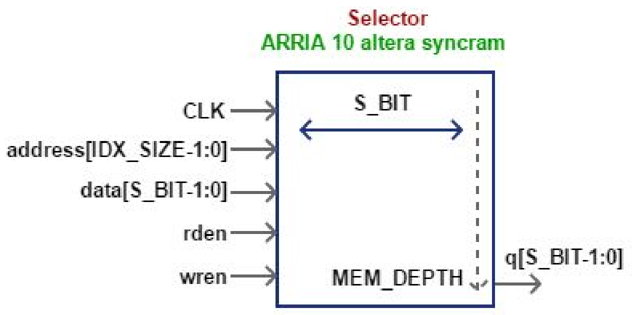


Figure 6.6 – The Selector (SEL) ARRIA 10 Altera syncram memory modules

6.6.1 Module Architecture

The Tournament predictor is actually **3 separate predictors working side-by-side**. The models used are the 2-Level predictor(section 6.5) and 2 n-bit predictors (section6.4) **without the**

implementation of the BTB inside them. As in any other predictor so far, the BTB works independently and delivers a prediction based on the Tournament prediction and BTB memory data. One of the n-bit predictors serves as the Meta-Predictor that selects between the other 2 predictors, to determine which will deliver the Tournament prediction.

Both the N-bit and the 2-Level predictors work and update their memories independently, depending only on their own correct or incorrect predictions. The Tournament predictor's controller is only responsible for fetching the correct **s-bit** predictor from the SEL memory and with the use of a multiplexer determine which of these two separate predictions to chose as the final one. The SEL memory works just like the BHT so the controller only needs to know the MSB of the fetched SEL word. **Arbitrarily, if MSB==0, the Tournament prediction comes from the n-bit predictor. Otherwise (MSB==1), the prediction comes from the 2-Level.**

The n-bit prediction is 'ready' in 3 CCs. However, because the 2-Level predictor needs 5 CCs to deliver its prediction, the Tournament predictor's control was created as a **5-state FSM**. **Due to the Tournament Predictor model's architectural, the the N-bit's prediction will be available at the 4th CC and the 2-Level's at the 5th CC**.

In the first state, just as in both previous predictor models, the controller waits for a valid input trace line to enter the simulator. When it does, it proceeds to the 2nd state just to wait for 1 CC. Meanwhile, the N-bit and 2-Level have also 'started' their individual jobs. Requests to the BTB and the SEL memory are not necessary for this state as their words will be useful in the 5th FSM state.

On the 3rd state, a request to the BTB memory is done on the same address just like on both previously described predictors. The Meta-Predictor's SEL memory is accessed using the PC exactly like the BHT on the N-bit predictor:

sel_address <= (pc » 19)&(pc » 3);

In the **4th** state, the N-bit prediction is received and stored on a register to be processed in the 5th FSM state.

On the final FSM state, the prediction from the 2-Level is also ready and the Meta-Predictor's prediction is stored in the registered input **sel_q**. After determining whether this prediction was correct based on the taken-not-taken information that came with the input trace-line, the FSM returns to its 1st state **after 5CCs**, where it idles until a new valid trace-line appears.

As mentioned before, both the n-bit and the 2-Level predictors work and update independently based on their own correct or incorrect predictions. However, the Meta-Predictor depends on the n-bit, 2-Level and Tournament predictions to update the SEL memory. **The SEL memory is updated only if the prediction from n-bit is different from the 2-Level's prediction**. If this is the case, there are only 2 kinds of possible SEL memory updates:

1. **Case 1 - The 2-Level predictor is correct:** In this case, if the SEL memory's s-bit predictor is not already 'all ones', it is updated by adding 1 to its previous value such as this: sel_data <= sel_q **+** 1. That way, the predictor moves closer to 'all ones' where it strongly selects the 2-Level predictor.

2. **Case 2 - The N-bit predictor is correct:** In this case, if the SEL memory's s-bit predictor is not already 'all zeros', it is updated by subtracting 1 from its previous value such as this: sel_data <= sel_q **-** 1. That way, the predictor moves closer to 'all zeros' where it strongly selects the N-bit predictor.

This update concludes the workflow of the Tournament predictor and happens on the 5th FSM state. An overview of the Tournament predictor along with all of its major components and procedures are depicted in figure 6.7

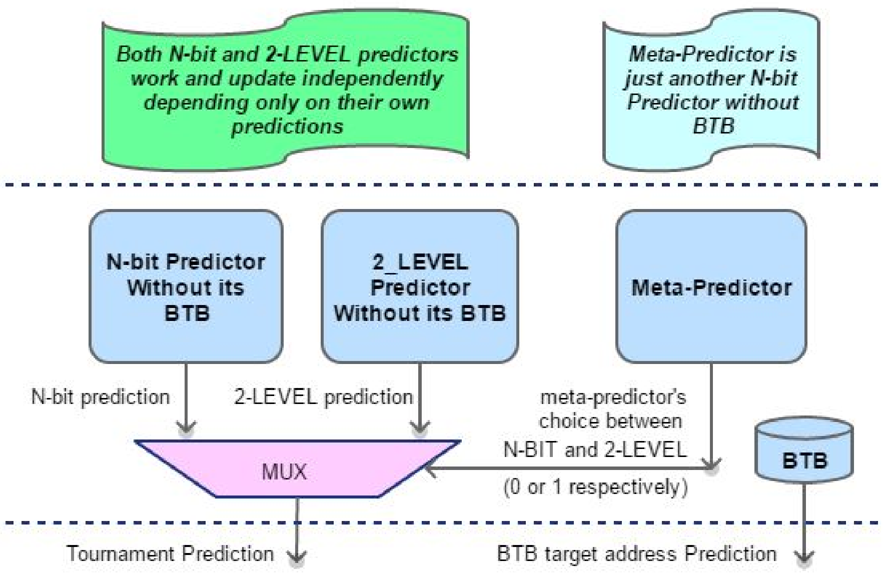

Both N-bit and 2-LEVEL predictors work and update independently depending only on their own predictions
Meta-Predictor is just another N-bit Predictor without BTB
N-bit Predictor Without its BTB
2_LEVEL Predictor Without its BTB
Meta-Predictor
N-bit prediction
2-LEVEL prediction
meta-predictor's choice between N-BIT and 2-LEVEL (0 or 1 respectively)
MUX
BTB
Tournament Prediction
BTB target address Prediction

Figure 6.7 – The Tournament branch predictor model

6.7 SimpleScalar

The SimpleScalar tool set [40] is a system software infrastructure used for program performance analysis, detailed micro-architectural modeling, and hardware-software co-verification. Users can simulate real programs running on a variety of processor and platform configurations. Furthermore, core architectural modules can be simulated individually to showcase their behavior through detailed simulation statistics and output log files. **The simulators in this tool set range from a fast functional simulator to a detailed processor model that supports non-blocking caches and state-of-the-art branch prediction.** Since 2000, a plethora of papers published in top computer architecture conferences used the SimpleScalar tools to evaluate their designs. That is why SimpleScalar was used as a "Golden-Model" in order to test and verify all of the Branch Predictors mentioned in the previous sections. More importantly, the SimpleScalar tools are available free-of-charge to academic and non-commercial users. Documentation, Tools, Benchmarks, Guides and Installation Tutorials can be found in the official SimpleScalar web-page [13].

6.7.1 Utilizing Key Features for Branch Prediction Development

As mentioned, SimpleScalar has many simulation capabilities. However, only Branch Prediction Simulation is going to be utilized to help validate all the previously described Branch Predictor models. SimpleScalar Branch Prediction is performed by using **-sim-bpred**, that can simulate 2 static and 3 dynamic predictors: Always Taken, Always Not-Taken, Bimodal (n-bit), 2-Level adaptive (GAg, PAg, and PAp) and finally the Combined (or Tournament).

Throughout its execution, **-sim-bpred** uses many functions from the SimpleScalar's .c and .h source files. The most interesting is the **bpred.c** that contains all the necessary procedures that create this simulator model including memory searches, memory updates and all the data cross-checking that deliver the branch predictor statistics. By analyzing the key features of the bpred.c source file along with the theoretical study of branch prediction, helped determine the memory addressing techniques and update policies that were used on the all branch predictor hardware models. These key features are:

- **Addressing the BTB memory**: btb_address <= pc » 3;
- **Addressing the BHT memory**: bht_address <= (pc » 19)&(pc » 3);
- **Addressing the SR memory in case of a PAp or PAg predictor**: sr_address <= (pc » 3);
- **Addressing the PHT memory using the appropriate history shift register**:
  pht_address <= (sr_q | ((pc » 3) « SR_SIZE));
- **Addressing the Meta-Predictor's SEl memory**: sel_address <= (pc » 19)&(pc » 3);
- **BTB update policy**: The BTB memory is updated only for taken branches in case of an address miss. These taken branched do not include jumps.
- **BHT and PHT update policy**: BHT and PHT memories do not get updated in case of jump instructions because their predictions would always be correct and not part of an actual prediction process.
- **SR update policy**: The SR memory does not get updated in case of jump instructions because they are always taken and are not part of a history pattern.
- **SEL update policy**: The Meta-Predictor's SEL memory gets updated only when the predictions from N-bit and 2-Level are different, for otherwise there are both right or wrong.

6.7.2 Benchmarking on SimpleScalar

To verify the hardware models, the **SPEC95** benchmarks (little endian) namely **compress95**, anagram, **go**, cc1 and perl were used. Compress95 and go were used the most, because of their execution time and output trace-file sizes. Detailed documentation about running the benchmarks can be found here [41].

In order to run these benchmarks on the SimpleScalar and more specifically -sim-bpred, the simulator needs to be configured. This configuration is possible through the simulator's execution command. These configurations are:

- **-bpred X**, where X represents the type of predictor that is going to be simulated, namely bimod, 2lev or comb.
- **-bpred:ras Y**, where Y represents the return address stack size (0 for no return stack)
- **-bpred:btb Z A**, where Z represents the number of words in the BTB memory and A the BTB memory associativity.
- **-bpred:bimod B**, where B represents the number of predictors in the BHT memory.
- **-bpred:2lev N M W X**. N stands for the number of history registers in the SR memory. M the number of predictors in the PHT memory. W is the size of the history registers. X (yes-1/no-0) shows whether to XOR the history and branch's PC as the PHT address.
- **-bpred:comb S**, where S represents the number of predictors in the Meta-Predictor's SEL memory.

In all predictors, the Return Address Stack (RAS) and BTB parameters must be specified. The BHT, PHT and SEL memories contain only 2-bit predictors. Also, in order to fully configure the Tournament predictor, both the Bimod (n-bit) and the 2-Level must be configured as well. As an example, in order to run the compress95 benchmark to simulate a Tournament predictor, the user should execute:

```
$ ./sim-bpred -bpred comb -bpred:comb 1024 -bpred:bimod 1024 -bpred:2lev 4 1024 10 0 -bpred:btb 1024 1 -bpred:ras 0 BenchMarks_Little/Programs/compress95.ss < BenchMarks_Little/Input/compress95.in 2> BenchMarks_Little/Results/compress95.trace > BenchMarks_Little/Results/compress95.out
```

Here the model has no RAS and the Bimod's BHT size 1024 words. The BTB contains 1024 words with associativity=1. The 2-Level is a PAg with 4 shift registers of size 10bits each. The PHT contains 2**10=1024 words and XOR with the PC is disabled. Finally, the Combined predictor has a Meta-Predictor containing 1024 words. Simulation statistics about each run are exported on .log and .trace files.

6.7.3 Extracting a Trace-File

The SimpleScalar's installation includes compiling all the source files contained in its downloaded directory, in order to create the executables that will become the various simulator instances like for example, the sim-bpred. As mentioned in section 6.2, the Branch Predictor Models need an input trace file to operate. In order to create one, before the SimpleScalar's installation, **the sim-bpred.c source file has been modified**.

For a successful **sim-bpred** simulation, the sim-bpred.c source file contains all the controls that detect a branch instruction and utilize the functions from the bpred.c to search the memories, determine address and direction hits or misses and also update the memory tables. **The sim-bpred's functionality has been extended with a feature that stores information about each detected branch instruction, regarding the branch's PC, Target Address, Opcode and information about whether it is actually taken or not-taken.** Then, by issuing a simple fprintf, as the one shown below, all these information are concatenated to composed a single trace-line of a newly created input trace-file. For simplicity and handling reasons for the Full State Simulator's development phase, each trace line is then converted to a single 64-bit integer, using the **int64_bp_trace_converter.c** program developed as part of this research program. This converter program is included in the Full State Simulator's source files. **The number of trace-lines equals to the number of branches detected** in a single benchmark run or a simulation in general.

```
...
fprintf(<bp_trace_file_name>,"%s%s%d%s\n",badd,btr,taken_not_taken,opcode);
...
```

With this modification present in the source code of the sim-bpred.c, the SimpleScalar is re-compiled/re-installed and is ready for testing, verification and trace-file generation. After running several benchmarks from SPEC95, for different predictor configurations on SimpleScalar using sim-bpred, multiple simulation statistics and input trace-files have been accumulated. All these were used in order to verify the Branch Predictor Models' functionality. The verification process is shown in section 6.8.

6.8 The Hardware Model's Verification

6.8.1 Executing on ModelSim

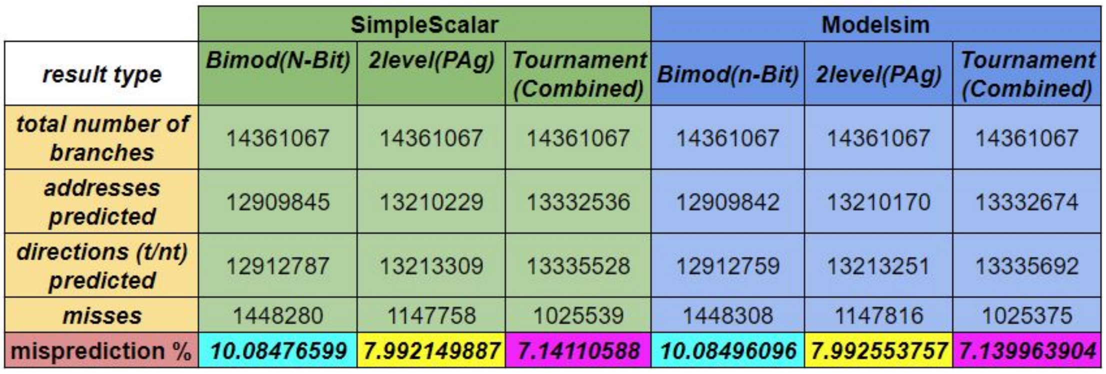
After collecting simulation statistics from the SimpleScalar, it is time to cross-check those results with the ones exported from the hardware branch predictor models. To create these statistics, all models need to be tested using the input trace-files described in subsection 6.7.3. This testing has been performed by using the ModelSim multi-language HDL simulation environment [38].

After loading a BP's source Verilog, SystemVerilog, and Verilog Header files on ModelSim, the user creates a new project. For this project, a **testbench.v** file that is capable of providing the branch predictor simulator with instructions (lines) from the trace-file needs to be developed. It is responsible for providing and receiving data to and from the BP's controller at the **correct moment**. For example, in the case of an n-bit predictor, a new trace-line can be processed every 3 CCs. If lines from the trace file come at a different rate, the predictor model will either waits for a new one or will miss some. Furthermore, the testbench needs to read the BP's output data only at the 3rd CC, otherwise, the simulation data will be invalid. This can also be performed by reading the 'valid_output' signal, exported from the BP. In the cases of the 2-Level and the Tournament predictor, the same things apply but for 5 CCs.

The Verilog testbench consists of a file-open and a while-loop. After opening the trace-file, the loop continues until it reaches the end-of-file. Depending on the predictor model, every 3 or 5 CCs a new line is read. This line of size **TRACE_LINE_LENGTH** is then split into **4 parts: pc, target, taken-not-taken** and the **branch opcode**. In this stage, the testbench also informs the predictor controller about a **valid** trace-line. After 3 or 5 CCs, the testbench reads the simulator's outputs and updates its counters that store information about total branches processed, branch address hits and direction hits.

6.8.2 Comparing the Results

After collecting all the simulation results and statistics from both SimpleScalar and ModelSim, the only thing left is to compare them. A side-by-side comparison of the ModelSim data and the SimpleScalar simulation results for each BP model, using the SPEC95's compress95 benchmark, is displayed in figure 6.8. **The trace-file that 'feeds' the BP models on ModelSim, is the one exported when running the same benchmark on SimpleScalar.**



	SimpleScalar			Modelsim		
result type	*Bimod(N-Bit)*	*2level(PAg)*	*Tournament (Combined)*	*Bimod(n-Bit)*	*2level(PAg)*	*Tournament (Combined)*
total number of branches	14361067	14361067	14361067	14361067	14361067	14361067
addresses predicted	12909845	13210229	13332536	12909842	13210170	13332674
directions (t/nt) predicted	12912787	13213309	13335528	12912759	13213251	13335692
misses	1448280	1147758	1025539	1448308	1147816	1025375
misprediction %	*10.08476599*	*7.992149887*	*7.14110588*	*10.08496096*	*7.992553757*	*7.139963904*

Figure 6.8 – Output and Statistics comparison of all branch predictor models from ModelSim and the SimpleScalar, using the SPEC95's compress95 benchmark.

The BPs tested on the SimpleScalar have the same configuration parameters as the ones tested on ModelSim. For the SPEC95's compress95 benchmark comparison shown in figure 6.8, these parameters are:

- **Parameter Definitions**:
  - **RAS**: 0=no and 1=yes for a return address stack,
  - **BTB_ASSOC**: BTB memory associativity,
  - **BTB_SIZE**: Number of words in the BTB memory,
  - **N_BIT**: Size of the predictors in the BHT memory,
  - **BHT_SIZE**: Number of words in the BHT memory,
  - **K_BIT**: Size of the predictors in the PHT memory,
  - **M**: Number of words in the PHT memory,
  - **N**: Number of history shift registers (number of words in the SR memory),
  - **W**: Bit-width of the history shift registers,
  - **XOR**: 0=no and 1=yes to enable 'XOR' between history register and branch pc to target the PHT memory,
  - **SEL_SIZE**: Number of words in the Meta-Predictor's SEL memory,
  - **S_BIT**: Size of the predictors in the SEL memory.
- **One-Level (n-bit or Bimod) predictor**:
  - **RAS** = 0,
  - **BTB_ASSOC** = 1,
  - **BTB_SIZE** = 1024,
  - **N_BIT** = 2,
  - **BHT_SIZE** = 1024.
- **2-Level PAg predictor**: PAg was chosen as it is the more generic implementation out of all 3, GAg, PAg and PAp predictors.
  - **RAS** = 0,
  - **BTB_ASSOC** = 1,
  - **BTB_SIZE** = 1024,
  - **K_BIT** = 2,
  - **M** = 1024,
  - **N** = 4,
  - **W** = 10,
  - **XOR** = 0.
- **Tournament (Combined) predictor**:
  - **RAS** = 0,
  - **BTB_ASSOC** = 1,
  - **BTB_SIZE** = 1024,

- **N_BIT** = 2,
- **BHT_SIZE** = 1024.
- **K_BIT** = 2,
- **M** = 1024,
- **N** = 4,
- **W** = 10,
- **XOR** = 0.
- **SEL_SIZE**: = 1024,
- **S_BIT**: = 2.

It is clear that the results from SimpleScalar and ModelSim between the same type of predictors are *almost* identical. The slight variation between the two platform results is caused by differences in addressing and update policies for some of the memory modules. However, **for every simulated benchmark execution and BP configuration, this difference always appears to be between 0.0004% - 0.002%, which is way below the acceptable margin of error**. That indistinguishable pattern continued throughout all the benchmarks results that were performed for various system configurations and for different benchmarks of the SPEC95 suite. Figure 6.8 is chosen as a general representation of the verified and correct functionality of all BP models developed for this Master's Thesis.

Chapter 7

Hardware Integration

7.1 Prerequisites and workflow

The majority of the prerequisites for developing a platform for Intel's HARP hybrid chip have already been mentioned, mostly on chapter 4. OPAE, ModelSim and Quartus Prime pro ver.16.0.0 should be pre-installed on the developers host platform. The first step is to create a SW API able to communicate with the AFU. The SW is responsible for the correct communication between the FPGA and the 'outside world', meaning it should be able to provide information like memory address spaces, location of buffers that the FPGA will utilize and of course, handle requests from the FPGA. The second step is creating an AFU that must adhere to the CCI-P limitations and rules. The AFU is the top-level of the custom accelerator that ports all of the accelerator's sub-modules and is the link to the FIU that connects the AFU to the the SW API.

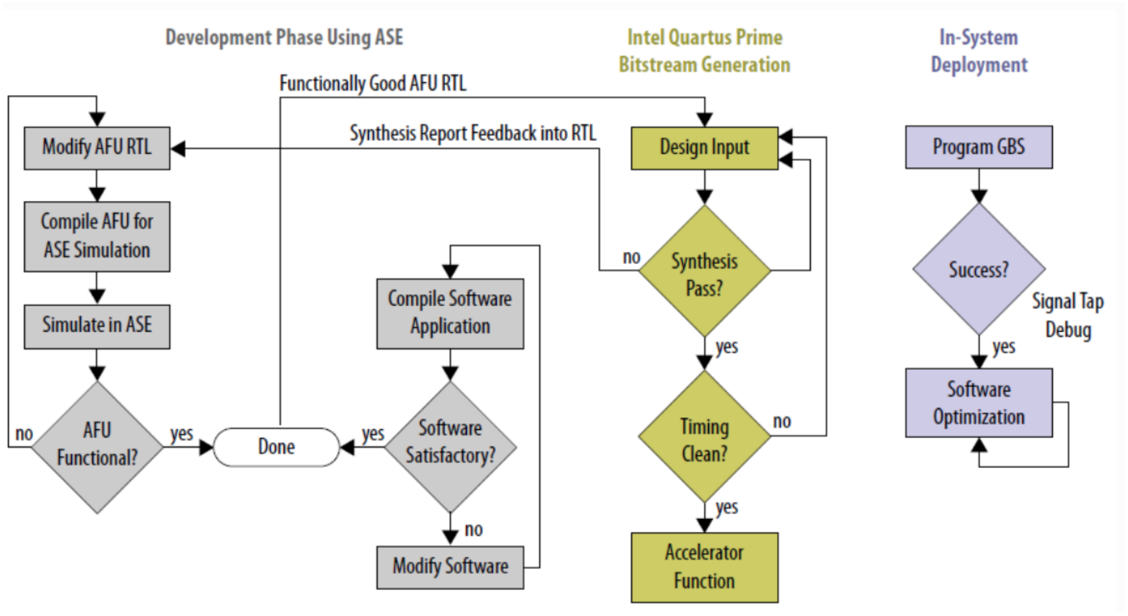


Figure 7.1 – Workflow for ASE Simulation, Quartus Synthesis and FPGA configuration. Image owned by [34]

The development phase is completed by the ASE simulation. ASE is part of the OPAE and helps for the HW-SW co-simulation. It is the main tool for verifying the correct functionality of both the AFU and the SW API. When the simulation results are correct the next step is the Bitstream Generation. Using the Quartus Prime Pro, a green bitstream is produced. This, along with the Intel's blue bitstream will be used to program the FPGA unless an error such as the ones described in section 4.4 exists. The final step is to run the SW and test if the FPGA works correctly. Note that incorrect functionality can either mean a SW error, a HW error, or both.

By using the OPAE and Intel's Basic Building Blocks (BBBs), apart from the ASE simulator, sample source codes and general guidelines, the developer gains access to scripts that automate the simulation and synthesis procedures. However, **executing these scripts requires an appropriate structure of the working directory**. The correct structure of a project that can utilize the OPAE scripts for ASE simulation and Quartus synthesis is shown in the following colorbox.

```
<project_folder_name>
| - - hw       : Hardware must be staged here
| | - - rtl    : RTL files
| | - - sim    : Simulation files list
| - - sw       : SW codes or header files
```

A crucial component for any HARP accelerator developed using the OPAE and BBB is the **.json** file, required by both HW and SW components. Located in the <project_name>/hw/rtl directory, the .json file is required both for simulation and synthesis. Many .json templates are available inside the BBB directories. The following colorbox depicts the .json file used for the full state simulator described in this dissertation. All key features are highlighted in red.

```
{ "version": 1,
"platform-name" : "HARP",
"afu-image":
"magic-no": 488605312,
"interface-uuid":
"00000000-0000-0000-0000-000000000000",
"clock-frequency-low": "auto",
"clock-frequency-high": "auto",
"power": 0,
"afu-top-interface":
{ "class": "ccip_std_afu",
"module-ports" :
[ { "class": "cci-p",
"params":
"clock": "pClkDiv2"
} ] },
"accelerator-clusters":
[{ "name": "cci_hello",
"total-contexts": 1,
"accelerator-type-uuid": "4634522f-0d1f-4fd3-983f-8fb7c50b552d" }]
}
```

The **magic-no** parameter is a default value provided in all of the BBB templates. The **accelerator-type-uuid** parameter is a unique ID that will be used to connect to the accelerator of choice and to issue SW API functions described in section 7.2. Finally, the clock is arguably the most important parameter of the .json file. By setting it to **pClk**, **pClkDiv2** or **pClkDiv4** the developer can set the desired RTL clock speed to 400MHz, 200MHz or 100MHz respectively.

7.2 Building the SW

After installing the latest version of the OPAE and downloading the BBBs, the developer has a plethora of simple accelerators and software interfacing examples to start with. Addressing the HW and reading its responses is possible through the SW API by utilizing the MMIO address space, specialized OPAE function libraries, and the DDR. Of course, this communication follows the CCI-P interface protocol. The OPAE provides a great number of header files, containing all the necessary functions that can be used to interface the AFU. The most useful header files reside in the **opae-sdk/common/include/opae/**, with the most useful being:

- **fpgaOpen(fpga_token token, fpga_handle *accel_handle, int flags)**: The accel_handle variable should have a unique value based on the unique accelerator ID called **AFU _ACCEL_UUID**. This will allow the SW API to connect to an accelerator of choice. Located in the **access.h** header file, **fpgaOpen()** is responsible for assigning/returning the correct value to the **accel_handle** variable.

- **fpgaPrepareBuffer(fpga_handle accel_handle, uint64_t size, void **buf_addr, uint64_t *wsid, int flags)**: Located in the **buffer.h**, this function allocates space for a buffer (in DDR) for shared access between an accelerator and the SW API process. It stores the starting address to the **buf_addr** pointer and updates the **wsid** variable that will later be used to free this buffer. Once again, the handle variable is the unique accelerator ID. **The size variable holds the buffer size in Bytes**.

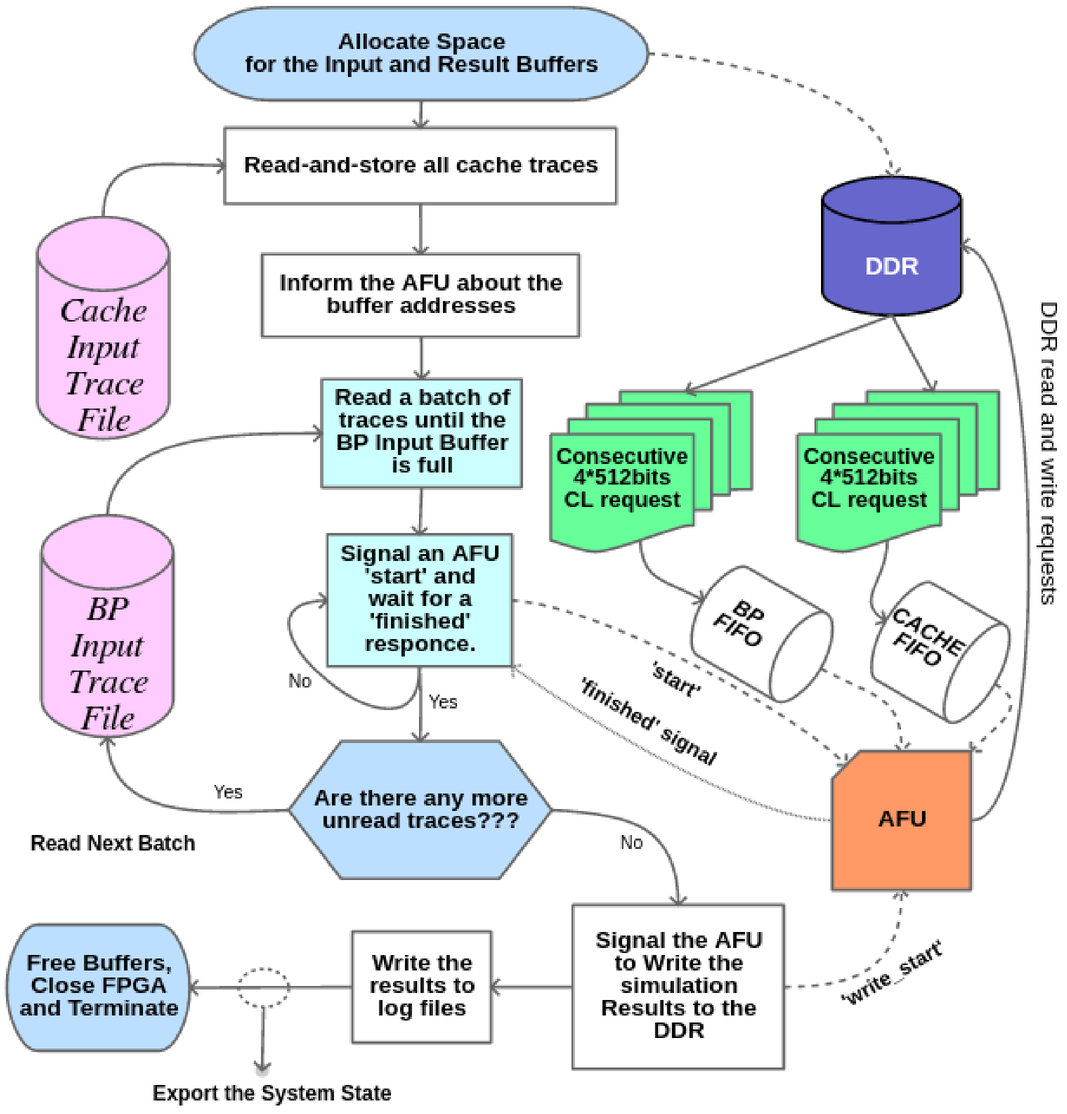
- **fpgaWriteMMIO64(fpga_handle accel_handle, uint32_t mmio_num, uint64_t offset, uint64_t value)** : Located in the **mmio.h**, this function is used to transmit a 512-bit value to the FPGA while also **informing a CSR register in the MMIO space at a specific offset**. This function is very useful because it is a great way to inform the AFU about the location of shared buffers and address spaces in the DDR. The **mmio_num is fixed-set to 0** for the main MMIO address space. For the Full State Simulator platform, the **fpgaWriteMMIO64()** function was used to inform about the address of the input trace buffer, the result buffer, the overall trace size and a 'start' and 'write_start' signal to initiate the accelerator. There are 2 key features that a developer needs to know about this function:

  1. There is a discrete way of addressing the MMIO space. Starting from offset == 0, **adding '+1' to the offset means increasing the address by 32 bits**. Therefore, to traverse an MMIO with 64-bit registers, it is intuitive that the offset needs to be escalated by +2 for each different request to avoid conflicts (i.e. 0, 2, 4, 8 etc.). **In case of a SW request, this translates to a byte-offset**. Therefore, to use **fpgaWriteMMIO64()** correctly, the offset should be incremented by '+8' at a time (0, 8, 16, 24 etc.).
  2. The '**value**', given to the **fpgaWriteMMIO64()** is either the address of a buffer or a 512-bit value in DDR. To handle an address efficiently, **the physical address should be divided by the size of a cache-line request (512bits)**. That way, the AFU can request 512 bits of a buffer simply by adding '+1' to the start address. For the 1st 512 bits, the address is [value/512 + 0]. For the 3rd 512 bits, the address is [value/512 + 2] and so on.

- **fpgaReadMMIO64(fpga_handle accel_handle, uint32_t mmio_num, uint64_t offset, uint64_t *value);** : Located in the **mmio.h**, this function is used to read a 64-bit value from a CSR MMIO register at a specified offset and placing it on the 'value' 64-bit variable. This function was used to read output signals from the AFU, like the '**finished**' signal informing that the accelerator has finished its work.

- **fpgaReleaseBuffer(fpga_handle accel_handle, uint64_t wsid)** and **fpgaClose(accel _handle)**: Located in the **buffer.h**, these functions are used at the end of the application to release the buffers provided the correct wsid and also to close the FPGA.

The SW API for the Full State Simulator requires 3 buffers, two for the input traces and one for the results from the AFU. Depending on the simulator platform's configuration, more buffers may be needed especially when 2 or more Full System Simulators may run simultaneously on the HARP FPGA. **The size of the input buffers is chosen experimentally and will be discussed in the following chapter**. Allocating buffer sizes as well as informing the AFU about the buffer addresses and sizes is performed by using the functions mentioned earlier.



Allocate Space for the Input and Result Buffers
Read-and-store all cache traces
Cache Input Trace File
Inform the AFU about the buffer addresses
DDR
DDR read and write requests
Read a batch of traces until the BP Input Buffer is full
Consecutive 4*512bits CL request
Consecutive 4*512bits CL request
BP Input Trace File
Signal an AFU 'start' and wait for a 'finished' responce.
BP FIFO
CACHE FIFO
'start'
'finished' signal
No
Yes
AFU
Yes
Are there any more unread traces???
Read Next Batch
No
Free Buffers, Close FPGA and Terminate
Write the results to log files
Signal the AFU to Write the simulation Results to the DDR
'write_start'
Export the System State

Figure 7.2 – The Full State Simulator's SW API and AFU (Macroscopic View)

Because the size of the input trace file may be larger than the input buffer, the input trace files can be read in whole or in batches. However, this implementation is developed only for the BP trace file. The Cache Trace File is read in whole. This was intentional in order to determine if the size of the input buffer can affect the system's performance. The results of this experiment are highlighted in the Results Chapter.

For each batch from the BP trace-file, the input buffer is loaded and the AFU is initiated. A 'start' signal to the AFU is given by an MMIO CSR register using **fpgaWriteMMIO64()**. The SW API is capable of waiting for the AFU to finish its work, using the **fpgaReadMMIO64()**, by waiting for a 'finished' signal. When this happens, the input buffer is refilled with new trace data and the process is repeated. Once all traces from all input trace files have been processed, the AFU is notified using the 'write_start' signal, to update the result buffer. These results are read by the SW and log files are exported, containing all these simulation and state data. After everything is finished, the buffers are freed and the FPGA is closed.

7.3 Building the HW - AFU

The AFU for the Branch Predictor Simulator, the Cache simulator and the Full State Simulator is based upon the AFU templates provided by the BBB for Intel FPGAs and has **6 main FSM's** as its core. The AFU works as the top-level for the Branch Predictors and the Cache model. It communicates with the SW API, consecutively reads trace lines from the DDR, stores them in FIFOs (unless they are full), 'feeds' the models with new input trace lines and exports simulation results depending on the model outputs. It is also responsible for reading the 'start' and 'write_start' signals from SW and creating the 'finished' and 'write_finished' signals to start and stop the FPGA and this simulator platform in general.

After configuring the .json file described in section 7.1 the clock and reset signal configuration is loaded like this:

```
logic clk;
logic reset;
assign clk = `PLATFORM_PARAM_CCI_P_CLOCK;
assign reset = `PLATFORM_PARAM_CCI_P_RESET;
```

The AFU module has a standard set of inputs and outputs to comply with the requirements of the CCI-P. Apart from the clock and the reset, 2 signals stand out:

- **input t_if_ccip_Rx pck_cp2af_sRx**: This is the AFU read channel or **Rx Port**. For simplicity, it was assigned to an **sRx** signal. This port also has a **sRx.c0.rspValid** signal that informs about a completed read request.

- **output t_if_ccip_Tx pck_af2cp_sTx**: This is the AFU transmit/write channel or **Tx Port**. For simplicity, it was assigned to an **sTx** signal. This port also has a **sRx.c1.rspValid** signal that informs about a completed write request.

The accelerator's unique ID that was set in the .json file, is passed through the **AFU_ACCEL_UUID** parameter like so:

```
logic [127:0] afu_id = `AFU_ACCEL_UUID;
```

Handling MMIO requests from the SW API is crucial for the correct functionality of this platform. The **sRx.c0.mmioRdValid** and **sRx.c0.mmioWrValid** signals check if there is a CSR read or write request active this cycle and are assigned to the **is_csr_read** and **is_csr_write** signals for simplicity. By utilizing the **mmio_req_hdr.address** signal, the AFU can read or write from/at specific offsets in the MMIO address space.

```
t_ccip_c0_ReqMmioHdr mmio_req_hdr;
assign mmio_req_hdr = t_ccip_c0_ReqMmioHdr'(sRx.c0.hdr);
```

There are 2 synchronous processes that handle CSR/MMIO read and write requests:

1. **The first process handles read requests to the MMIO**. A read request to the MMIO (**is_csr_read==1**) translates to a CSR register write at a certain offset from the AFU's perspective. The sTx.c2.mmioRdValid signal is set to '1' and depending on the offset of the

MMIO read request, sTx.c2.data is updated with the appropriate value. For example, it is mentioned that the SW API waits for a **'finished'** signal from the AFU. To view this, the SW issues multiple MMIO read requests to offset **40** using the **fpgaReadMMIO64()** function. This translates to **offset 40/4=10** from the AFU side. To handle this case, the process looks like this:

```
always_ff @(posedge clk)
begin
if (reset) begin
sTx.c2.mmioRdValid <= 1'b0;
end else begin
sTx.c2.mmioRdValid <= is_csr_read;
...
case (mmio_req_hdr.address)
...
10: sTx.c2.data <= t_ccip_mmioData'(finished);
...
default: sTx.c2.data <= t_ccip_mmioData'(0);
endcase
end
```

2. **The second process handles the write requests to the MMIO**. A write request from the SW API (**is_csr_write==1**), translates to an MMIO and data read from the AFU's perspective. **Whenever an MMIO write request reaches the AFU, 512-bits of data can be read from the sRx.c0.data signal**. Depending on the request's offset, the AFU knows which kind of data complement this particular request and updates its registers accordingly. For example, the SW API informs the AFU about the start-address of the input file buffer at offset **80** using the **fpgaWriteMMIO64()** function. The AFU is capable of receiving this address by checking for the is_csr_write==1 signal and an **mmio_req_hdr.address=80/4=20**. The process that handles this kind of write requests looks like this:

```
always_ff @(posedge clk)
begin
...
if (is_csr_write) begin
case (mmio_req_hdr.address)
...
20: mem_addr_trace[511:0] <= t_ccip_clAddr'(sRx.c0.data);
...
endcase
end
```

7.3.1 DDR Read and Write Requests

Now that the AFU has all the necessary features to communicate with the SW API, another thing to explain the communication with the DDR. **Read and write requests are issued from the AFU by configuring the read and write headers as well as the valid request signals of Channel 0**

and 1 respectively. For a read request, performed using the sRx Port on Channel 0, here are all the necessary signals:

- t_ccip_c0_ReqMemHdr **rd_hdr** : Define a read request header.
- **rd_hdr =** t_ccip_c0_ReqMemHdr**'(0)** : Initialize the read request header.
- **rd_hdr.mdata =** t_ccip_mdata**'(16-bit_value)** : The mdata is a very useful 16-bit value that the AFU can issue with a read request to monitor its progress. When the AFU receives an **sRx.c0.rspValid==1** it knows that a read request is completed. **However, valid responses and read requests are received out-of-order**. So, the mdata that complement a valid signal, guarantee that this response matches a particular request, otherwise this valid signal may originate from a different read request.
- **rd_hdr.address =** t_ccip_clAddr**'(buffer_address + index)** : This sets the DDR address that the AFU will read from. The buffer_address marks the starting address of a buffer in DDR. By adding an index, the AFU traverses this buffer. As mentioned in section 7.1, incrementing the address by +1, translates to a 512-bit shift.
- **rd_hdr.cl_len =** t_ccip_clLen**'(2-bit_value)** : By setting the 2-bit_value to 01, 10 or 11, the AFU can request 1 (512 bits), 2 (1024 bits) or 4 (2048 bits) cache-lines from the DDR with a single read request. Once again, it is noted that the cache-lines may come out-of-order so the use of mdata is mandatory, unless the order of data is insignificant.
- **sTx.c0.valid** : By setting this to '1' the AFU initiates a new read request to the DDR with all the necessary read request header configurations mentioned above.
- **sRx.c0.rspValid** : When this signal equals to '1', it means that a read request has been successfully processed.

For a write request, performed using the sTx Port on Channel 1, here are all the necessary signals:

- t_ccip_c1_ReqMemHdr **wr_hdr** : Define a write request header.
- **wr_hdr =** t_ccip_c1_ReqMemHdr**'(0)** : Initialize the write request header.
- **wr_hdr.address = write_buffer_address + index** : This sets the DDR address that the AFU will write to. The write_buffer_address marks the starting address of a buffer in DDR. By adding an index, the AFU traverses this buffer. As mentioned in section 7.1, incrementing the address by +1, translates to a 512-bit shift.
- **sTx.c1.valid** : By setting this to '1' the AFU initiates a new write request to the DDR with all the necessary write request header configurations mentioned above.
- **sRx.c1.rspValid** : When this signal equals to '1', it means that a write request has been successfully processed. **However, this valid signal is received a few CCs before the actual write of the data in DDR so the developers need to take note of this behavior**.

For more information about the CCI-P refer to [33].

7.3.2 The Read FSMs

There are **2 identical FSMs** for reading traces from the DDR and feeding them into FIFOs for the BPs and the Cache model by issuing **quadruple CL read requests**. **Arbitrarily, the FSM that feeds the BPs has priority over the one that feeds the Cache model, in case they both are about to issue a read request at the same CC**. The reason for using 2 FSMs instead of one is that the input trace file and the trace-lines for the BPs are different from the ones used to simulate the Cache. As the FSMs are identical, only the FSM that feeds the BPs is going to be described.

The Read FSM is responsible for issuing read requests to the DDR, reading the DDR responses, validating the received data, loading the input FIFO and informing the SW API that the AFU has finished its work when all the trace-lines from the DDR have been successfully processed. **Both Input FIFOs used in this simulator are modules that contain an entire cache-line, meaning 8 trace lines (512/64=8) from the BP input trace file and 16 trace lines (512/32=16) from the Cache input trace file**.

This FSM has **10 states**:

1. **IDLE**: This is the FSM's initial state. When the AFU receives the '**start**' signal, the FSM jumps from this state to the **READ_REQUEST** state, otherwise, it remains on the same state.

2. **READ_REQUEST**: In this state the FSM checks if Channel 0 is busy by using the **sRx.c0TxAlmFull** signal. If it is, it repeats the check until it is not. If it is not busy, it issues a 4-cache-line (4CLs == 4*512 bits) request to the DDR and proceeds to state **WAIT_AND_STORE**.

3. **WAIT_AND_STORE**: After issuing a read request, the AFU has to wait for all 4 CLs to be received. As mentioned previously, the CLs may come out-of-order. So, with the use of 4 helper variables/registers, the AFU tracks the reception of each of the 4 read valid signals. **As soon as CL reaches the AFU, it is stored on a temporary register**. The FSM remains in this state until all CLs are read and are successfully loaded on 4 temporary registers. Then it proceeds to state **INITIAL_CHECK**.

4. **INITIAL_CHECK**: After the **WAIT_AND_STORE** state, 4 CLs are reserved in 4 temporary registers. In this state, the AFU checks if the CLs received are valid. Especially in the later stages of the simulation, the input trace buffer may have been zero-padded so if every CL equals to zero, there is no need to feed them to the simulator modules and the FSM jumps straight to state **CHECK_FIFO_FULL**. If there is even a single CL that is not all-zeros, the FSM proceeds to state **WRITE_FIFO_0**.

5. **WRITE_FIFO_0**: Because the order of traces that are fed to the simulator modules has to be precise, in this state the AFU writes the 1st CL to the FIFO. **The 1st CL means the 1st of the 4CL-batch, not the 1st one received**. After this FIFO write, the FSM proceeds to state **WRITE_FIFO_1**.

6. **WRITE_FIFO_1**: Similar to state **WRITE_FIFO_0**, in this state the AFU writes the 2nd CL to the FIFO. After this FIFO write, the FSM proceeds to state **WRITE_FIFO_2**.

7. **WRITE_FIFO_2**: Similar to state **WRITE_FIFO_1**, in this state the AFU writes the 3rd CL to the FIFO. After this FIFO write, the FSM proceeds to state **WRITE_FIFO_3**.

8. **WRITE_FIFO_3**: In this state, the AFU writes the 4th and final CL to the FIFO. After this FIFO write, the FSM proceeds to state **CHECK_FIFO_FULL**.

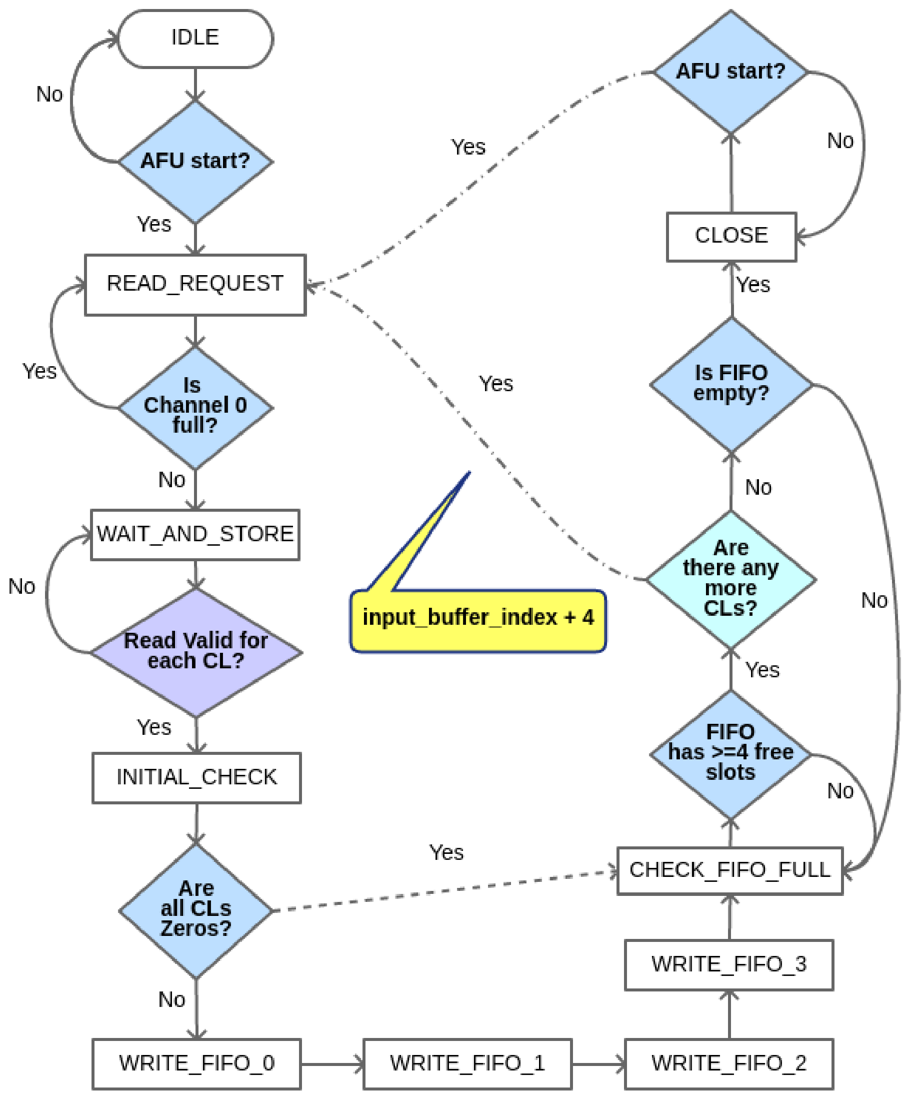


Figure 7.3 – The AFU's Branch Predictor Read FSM

9. **CHECK_FIFO_FULL**: This state has multiple controls. **First**, it checks whether the FIFO is almost full. Because the AFU only issues quadruple (4CL) read requests, the FIFO should have at least 4 empty slots. If this is not the case, the FSM remains in this state. **Second**, if there are 4 or more empty slots, the AFU checks whether every cache-line from the input buffer has already been read and if the FIFO is empty. **Note that the input buffer is built in such a way so that it is a complete multiple of 32, thus guaranteeing that every 4CL read request has valid cache-lines**. If all lines are processed and the FIFO is empty, the FSM proceeds to state **CLOSE**. However, if the FIFO is not-empty, the FSM remains in this state until every FIFO-line is successfully processed. **Finally**, if there are 4-or-more free FIFO slots and there are still unprocessed traces, the read address index is incremented by '4' (4*512 == The next 4 CLs) and the FSM returns to state **READ_REQUEST**.

10. **CLOSE**: This is the final state where the AFU informs the SW API, via the 'finished' signal, that it has completed its work. However, because the trace file might be larger than the input trace buffer, t**he AFU should be capable of restarting** and executing its procedures for a new input trace buffer. This is why the FSM Idles in this state until a new 'start' signal is received from the SW API. When it does, it proceeds to state **READ_REQUEST**.

7.3.3 The Feed FSMs

There are **2 identical FSMs** for 'feeding' traces to the BPs and the Cache model. The reason for using 2 FSMs instead of one is that the number of traces included inside a line from the BP FIFO is different from the one in the Cache FIFO. A line from the BP FIFO contains 8 trace lines from the BP's input trace file, meaning that the BP's Feed FSM can 'feed' all-three BP models for 8 consecutive times before issuing a new FIFO read request. On the other hand, a line from the Cache FIFO contains 16 trace lines from the Cache's input trace file, meaning that the Cache's Feed FSM can 'feed' the Cache model for 16 consecutive times before issuing a new FIFO read request.

Both FSMs have 3 states for waiting the AFU start signal and issuing read requests to their respective FIFOs. However, the BP Feed FSM has 16 Feed-Check (19 total states) and the Cache Feed FSM has 32 Feed-Check (35 total states) because they feed 64-bits and 32-bits respectively. As the FSMs are practically identical, only the FSM that feeds the BPs is going to be described:

1. **WAITING**: In this state, the Feed FSM waits for the '**start**' signal from the SW API. When it comes, it jumps to state **READ_FROM_FIFO**, otherwise, it remains in **WAITING**. While in this case, the Feed FSM is in 'idle' mode.

2. **READ_FROM_FIFO**: In this case, the Feed FSM issues a read request to the BP FIFO and proceeds to state **WAIT_FIFO**. However, if the BP FIFO is empty, the Feed FSM remains in this state.

3. **WAIT_FIFO**: The FSM waits for 1CC for the BP FIFO response and proceeds to state **FEED _1**.

4. **FEED_1**: In this state, the FSM reads the response from the BP FIFO and stores the entire CL into a 512-bit register for processing. **First**, it checks the trace-line's [63:0] bits. If they are all zeros, the trace is not useful so the FSM returns to state **READ_FROM_FIFO**. **This is done because zero traces appear only when the input trace buffer is zero-padded at its end, meaning that all the following traces are also all-zeros**. If not, it loads this trace to the Branch Predictors (or [31:0] for the Cache Model) and proceeds to state **CHECK_1**.

5. **CHECK_1**: Now, the FSM has to wait for each simulated module (BPs or Cache) to finish its work with the trace provided in **FEED_1**. When a module has completed its work, it exports a '**valid_output**' signal that the FSM reads and proceeds to state **FEED_2**, otherwise, the state remains the same.

6. **FEED_2**: Here, the FSM checks the trace-line's [127:64] bits. If they are all zeros, the trace is not useful so it moves straight to state **READ_FROM_FIFO**. If not, it loads this trace to the BPs (or [61:32] for the Cache Model) and proceeds to state **CHECK_2**.

7. **CHECK_2**: Once again, the FSM has to wait for each simulated module to finish its work with the trace provided in **FEED_2**. When every module has completed its work, the FSM proceeds to state **FEED_3**, otherwise, the state remains the same.

8. **FEED_3**: Here, the FSM checks the trace-line's [191:128] bits. If they are all zeros, the trace is not useful so it moves straight to state **READ_FROM_FIFO**. If not, it loads this trace to the BPs and proceeds to state **CHECK_3**.

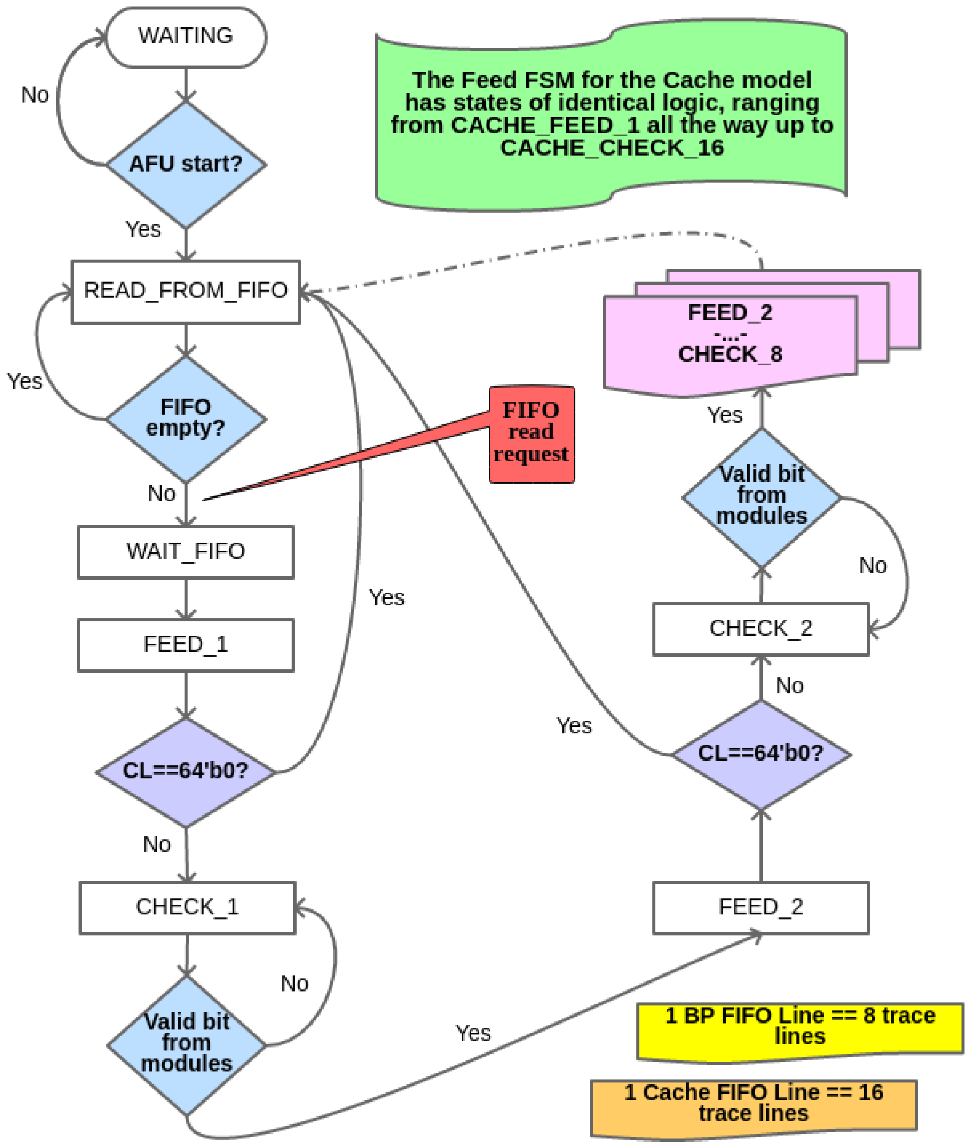


Figure 7.4 – The AFU's Branch Predictor Feed FSM

9. **CHECK_3**: Once again, the FSM waits for each simulated module to finish its work with the trace provided in **FEED_3**. Only then it proceeds to state **FEED_4**, otherwise, the state remains the same.
10. **FEED_4**: Same as in every other **FEED_x** state, only in this case it checks the trace-line's [255:192] bits.
11. **CHECK_4**: Same as in every other **CHECK_x** state.
12. **FEED_5**: Same as in every other **FEED_x** state, only in this case it checks the trace-line's [319:256] bits.
13. **CHECK_5**: Same as in every other **CHECK_x** state.
14. **FEED_6**: Same as in every other **FEED_x** state, only in this case it checks the trace-line's [383:320] bits.
15. **CHECK_6**: Same as in every other **CHECK_x** state.

16. **FEED_7**: Same as in every other **FEED_x** state, only in this case it checks the trace-line's [447:384] bits.

17. **CHECK_7**: Same as in every other **CHECK_x** state.

18. **FEED_8**: Same as in every other **FEED_x** state, only in this case it checks the trace-line's [511:448] bits.

19. **CHECK_8**: Same as in every other **CHECK_x** state. Only this time the FSM returns to state **READ_FROM_FIFO**.

20. **NOTE**: The Cache Feed FSM has states from **CACHE_FEED_1** to **CACHE_CHECK_16**

7.3.4 The Write FSM

After the SW API is informed that the AFU has finished its work processing the input trace files, the next step is to export the simulation statistics. The '**write_start**' signal informs the AFU about exporting the simulation results. **Throughout the simulation process, the AFU has separate processes that detect valid output signals from each individual BP and the Cache Model. When these valid signals become '1', the AFU updates the simulation statistics for the appropriate simulated model that has triggered this valid bit**. For example, if the AFU detects a valid signal from the 2-Level Predictor, this triggers an update to the values of the **lev_total_number _of_branches**, the **lev_total_number _of_address_hits**, and the **lev_total_number _of_direction_hits** 64-bit registers, which are used to store the simulation results for this particular predictor model.

The SW API allocates space for a **Result Buffer** and informs the AFU about its address on the DDR. Using this address and an appropriate **write_address_index**, the AFU will issue single or multiple write requests in order to export the simulation statistics. **The Write FSM has 6 states**:

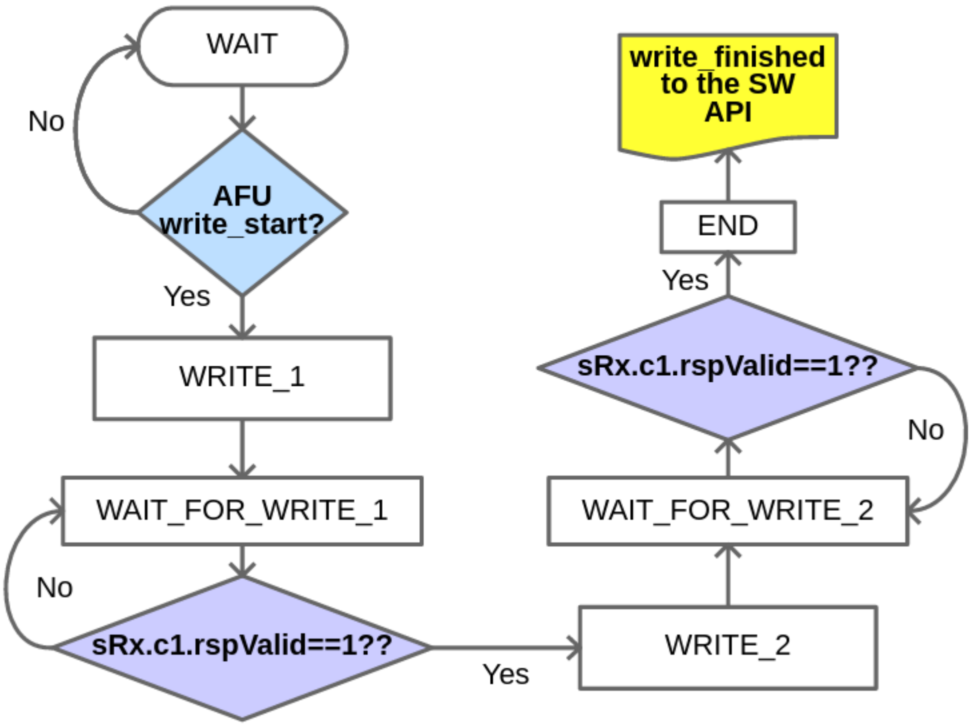


Figure 7.5 – The AFU's Write FSM

1. **WAIT**: In this state, the Write FSM waits for the '**write_start**' signal from the SW API. When this signal comes, it moves to state **WRITE_1**, otherwise, it remains in **WAIT**.

2. **WRITE_1**: At this point, the Write FSM issues a write request to the DDR in the location of the Result Buffer. After the DDR write request, the FSM advances to state **WAIT_FOR_WRITE_1**. A single DDR write request has 512-bits of data and requires:

   - Assigning the **sTx.c1.hdr.address** to the result_buffer_address.
   - Asserting the **sTx.c1.valid** bit to '1';
   - Loading the 512-bits of **sTx.c1.data** with the simulation results. **In this FSM state, the DDR write request contains simulation results about the AFU total clock-cycles (CCs), the One-level (n-bit) predictor and the 2-level predictor**.

3. **WAIT_FOR_WRITE_1**: Now the FSM has to wait for the **sRx.c1.rspValid==1** signal that will mark a completed write request. **Note that the sRx.c1.rspValid==1 signal may come multiple (arbitrary value) CCs before the actual DDR write so the developers have to be cautious when issuing DDR write requests**. Only when this signal arrives the FSM can proceed to state **WRITE_2**.

4. **WRITE_2**: At this point, the Write FSM issues a 2nd write request to the DDR in the location of the Result Buffer + 1 (the next 512-bits). After the DDR write request, the FSM advances to state **WAIT_FOR_WRITE_2**. **This write request contains simulation results about the Tournament predictor and the Cache model**.

5. **WAIT_FOR_WRITE_2**: Now the FSM has to wait for the **sRx.c1.rspValid==1** signal for the 2nd write request. Only then the FSM can proceed to state **END**.

6. **END**: Finally, the Write FSM informs the SW API that it has finished writing the simulation statistics to the DDR with a dedicated '**write_finished**' CSR register of the MMIO address space at offset **16** (or 16*4=64).

7. **NOTE**: The Write FSM is only used in the BP and BP+Cache Simulators. For the Full State Simulator, the Write Cache is replaced by the State FSM.

7.3.5 The State FSM

The State FSM is a replacement of to the Write FSM and is present only in the Full System Simulator. Once again, it is used to print the **simulation results** that are exported from the simulator modules, however, it is also responsible for printing the **system's state**. Both the simulation results and the system's state are exported in the form of a log file. This FSM stores the state data, with a specific format, inside the results buffer, created by the SW API. The format of the state-data will be discussed along with the analysis of each FSM state. It should be noted that in order for the simulator to print the correct data to the log file, all memory sizes need to be provided to the SW API at execution time. **All 7 SW API parameters are depicted in figure 7.6.**

In order to print a system state, during the processing of the traces by the HW simulator modules, the internal memories are copied to 'mirror-memory-modules'. These modules are exact copies of the internal memory modules and are used to print the system state with ease. The processes that are responsible for handling the data exported from the BPs and the Cache Model are responsible for updating these memory modules whenever a new output is prompted. The State FSM is only responsible for reading these data in order to print them with a certain format

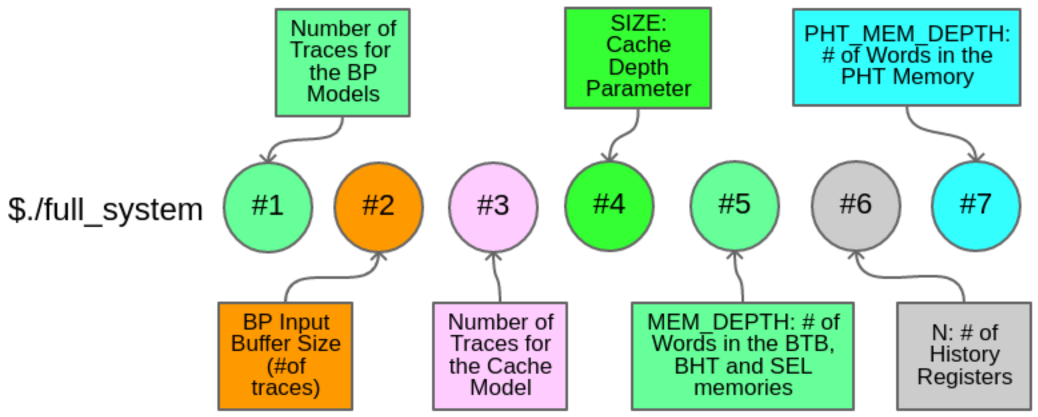

Number of Traces for the BP Models
SIZE: Cache Depth Parameter
PHT_MEM_DEPTH: # of Words in the PHT Memory
$./full_system
#1
#2
#3
#4
#5
#6
#7
BP Input Buffer Size (#of traces)
Number of Traces for the Cache Model
MEM_DEPTH: # of Words in the BTB, BHT and SEL memories
N: # of History Registers

Figure 7.6 – SW API execution parameters.

inside the final simulation log file. **All the data included inside the log file as well as the format of the state data will be analysed in this section and in section 8.3.**

In the upcoming enumerated bullets, **all 35 states** of the State FSM are going to be thoroughly described state-by-state. A key feature of this FSM and the CCI-P in general, is that the DDR Write Request are **streaming requests**, so there is no additional delay in order to wait for a write-valid response from channel 1. **However, the SW API is responsible for allowing the AFU to write all the data to the DDR, by providing a sufficient delay between the completion of the State FSM (write_finished=1 AFU signal to the API) and the actual printing of the final log file.**

Because each batch of data being written with a write request is 512 bits, a printing index **state_index_for_the_ddr_writes** is used to increment the write header's address, in order to store the data inside the result buffer at the desired place and to avoid overlapping data.

1. **STATE_WAIT**: This is the initial and reset state of the State FSM. Here, the FSM idles until a '**write-start**' signal is issued from the SW API. Only then, the FSM proceeds to state **STATE_WRITE_1**.

2. **STATE_WRITE_1**: Compared to the Write FSM, in this state the FSM prints all the **simulation data as 32 bit integers**. These simulation data are the number of CCs after reset, the total number of branches, address hits and direction hits for the One-Level, 2-Level and Tournament predictors, the number of times the 2-Level was the predictor of choice by the Tournament's Meta Predictor, the number of traces processed by the Cache model, the number of Cache Hits and finally the number of Cache Misses. The write header is set to point at the start of the results buffer and all these data are included inside a single 512-bit CL. The printing index **state_index_for_the_ddr_writes** is incremented by '1' and the FSM proceeds to state **STATE_L1_READ**.

3. **STATE_L1_READ**: Using a dedicated address index **l1_state_mem_address**, created to traverse the cache mirror memory module, the FSM issues a read request to fetch a single cache-line from the cache's mirror memory. It then proceeds to state **STATE_L1_WAIT**.

4. **STATE_L1_WAIT**: The data requested in state **STATE_L1_READ** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_L1_PROCESS**.

5. **STATE_L1_PROCESS**: In this state, the FSM receives the data requested in state **STATE**

_L1 _READ. Using a helper 512-bit register **state_line_to_write** the FSM formats a 512-bit value to be written to the result buffer. Because the Cache model supports up to an 8-way memory configuration, an entire line from the cache memory can be stored in these 512-bits. Each way is contained using 31 bits of a 32-bit integer and the replacement indexes of the cache line are stored in a different 32-bit integer. Depending on the number of ways, the **state_line_to_write** has more or less useful data with all unused ways being replaced with 32-zero-bits. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_L1_WRITE**. The format of a **state_line_to_write**, that is part of the Cache Model's state data, is depicted in figure 7.7.

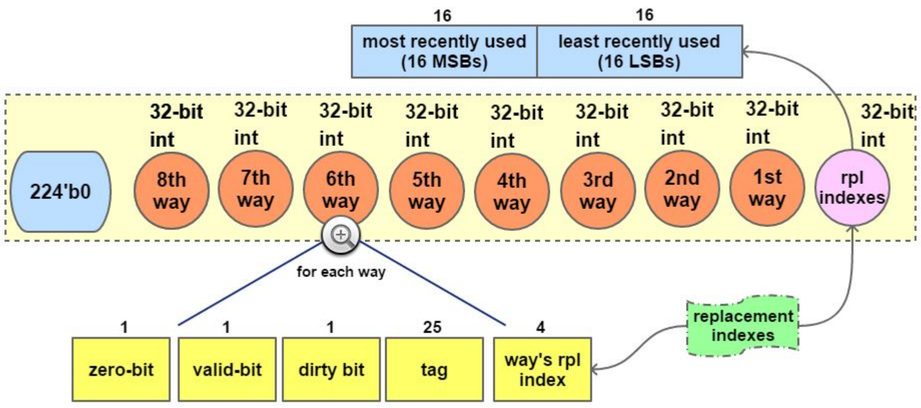


Figure 7.7 – The format of a **state_line_to_write** containing Cache Model's state data.

6. **STATE_L1_WRITE**: At this point, the **state_line_to_write** is ready, so, the FSM issues a DDR write request for these 512-bits to the result buffer and the printing index **state_index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire mirror cache memory. If this is the case, the **l1_state_mem_address** address index is set to zero and the FSM proceeds to state **STATE_NBIT_READ**. Otherwise, the **l1_state_mem_address** address index is incremented by '1' and the process is repeated by returning to state **STATE _L1 _READ**.

7. **STATE_NBIT_READ**: The next step is to export a state for the One-Level Branch Predictor. Both the One-Level's BTB and the BHT have the same **MEM_DEPTH** number of words, so, using a single dedicated address index **nbit_state_btb_and_bht_address**, the FSM can traverse the BHT and BTB mirror memory modules. In this state, the FSM issues a read request to fetch a single line from the BHT and the BTB mirror memories and then moves to state **STATE_NBIT_WAIT**.

8. **STATE_NBIT_WAIT**: The data requested in state **STATE_NBIT_READ** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_NBIT_PROCESS**.

9. **STATE_NBIT_PROCESS**: In this state the lines from the One-Level's BTB and BHT mirror memories are received and these data are used to synthesize the 512-bits of the **state_line_to_write** register. Once again the data are stored as 32-bit integers inside the 512-bit register in order to be exported to the DDR. These data are the branch's target PC and opcode from the BTB as well as the n-bit predictor from the BHT. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_NBIT_WRITE**. The format of a **state_line_to_write** is depicted in figure 7.8.

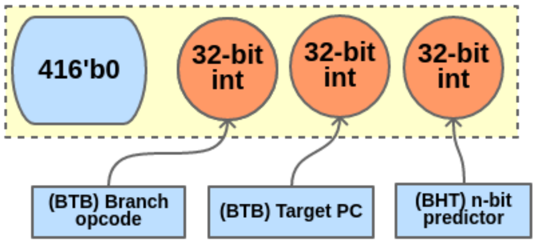


Figure 7.8 – The format of a **state_line_to_write** containing One-Level Predictor's state data.

10. **STATE_NBIT_WRITE**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire One-Level's mirror BTB and BHT memories (**MEM_DEPTH # of words**). If this is the case, the **nbit _state _btb_and_bht_address** address index is set to zero and the FSM proceeds to state **STATE_2LEV_READ_SR**. Otherwise, the **nbit_state _btb_and_bht_address** address index is incremented by '1' and the process is repeated by returning to state **STATE_NBIT _READ**.

11. **STATE_2LEV_READ_SR**: The next step is to export a state for the 2-Level Branch Predictor. First, the FSM has to read the 2-Level's SR mirror memory using a dedicated **twolev_state_sr _address** address index. In this state, the FSM issues a read request to the SR mirror memory and advances to state **STATE_2LEV_WAIT_SR**.

12. **STATE_2LEV_WAIT_SR**: The data requested in state **STATE_2LEV_READ_SR** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_2LEV_PROCESS_SR**.

13. **STATE_2LEV_PROCESS_SR**: In this state the data from the 2-Level's SR mirror memory are received and are used to synthesize the 512-bits of the **state_line_to_write** register. Each line from the SR represents a single history register whose data are exported using a single 32 bit integer. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_2LEV_WRITE_SR**. The format of a **state_line_to_write** is depicted in figure 7.9.

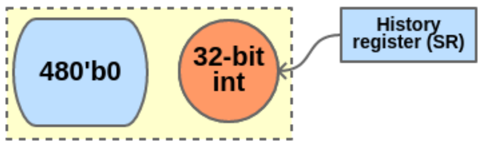


Figure 7.9 – The format of a **state_line_to_write** containing a history register for the 2-Level Predictor's state data.

14. **STATE_2LEV_WRITE_SR**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire 2-Level's mirror SR mirror memory (**N # of words**). If this is the case, the **twolev_state_sr _address** address index is set to zero and the FSM proceeds to state **STATE_2LEV_READ_BTB**. Otherwise, the **twolev_state_sr _address** address index is incremented by '1' and the process is repeated by returning to state **STATE_2LEV_READ _SR**.

15. **STATE_2LEV_READ_BTB**: The second step is to read the 2-Level's BTB mirror memory using a dedicated **twolev_state_btb_address** address index. In this state, the FSM issues a read request to the BTB mirror memory and advances to state **STATE_2LEV_WAIT_BTB**.

16. **STATE_2LEV_WAIT_BTB**: The data requested in state **STATE_2LEV_READ_BTB** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_2LEV_PROCESS_BTB**.

17. **STATE_2LEV_PROCESS_BTB**: In this state the data from the 2-Level's BTB mirror memory are received and are used to synthesize the 512-bits of the **state_line_to_write** register. The branch's opcode and the target PC contained inside a single line from the BTB mirror memory are exported using two 32 bit integers. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_2LEV_WRITE_BTB**. The format of a **state_line_to _write** is depicted in figure 7.10.

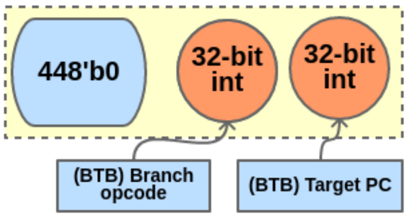


Figure 7.10 – The format of a **state_line_to_write** containing the BTB's branch opcode and the target PC for the 2-Level Predictor's state data.

18. **STATE_2LEV_WRITE_BTB**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire 2-Level's mirror BTB mirror memory (**MEM_DEPTH # of words**). If this is the case, the **twolev_state_btb_address** address index is set to zero and the FSM proceeds to state **STATE_2LEV_READ_PHT**. Otherwise, the **twolev_state_btb_address** address index is incremented by '1' and the process is repeated by returning to state **STATE_2LEV _READ_BTB**.

19. **STATE_2LEV_READ_PHT**: The third and final step is to read the 2-Level's PHT mirror memory using a dedicated **twolev_state_pht_address** address index. In this state, the FSM issues a read request to the PHT mirror memory and advances to state **STATE_2LEV _WAIT_PHT**.

20. **STATE_2LEV_WAIT_PHT**: The data requested in state **STATE_2LEV_READ_PHT** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_2LEV_PROCESS_PHT**.

21. **STATE_2LEV_PROCESS_PHT**: In this state the data from the 2-Level's PHT mirror memory are received and are used to synthesize the 512-bits of the **state_line_to_write** register. The k-bit predictor contained inside a single line from the PHT mirror memory is exported using a single 32 bit integer. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_2LEV_WRITE_PHT**. The format of a **state_line_to _write** is depicted in figure 7.11.

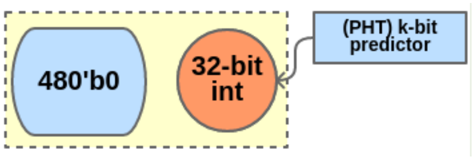


Figure 7.11 – The format of a **state_line_to_write** containing the PHT's k-bit predictor for the 2-Level Predictor's state data.

22. **STATE_2LEV_WRITE_PHT**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire 2-Level's mirror PHT mirror memory (**PHT_MEM_DEPTH # of words**). If this is the case, the **twolev_state_pht_address** address index is set to zero and the FSM proceeds to state **STATE_TOUR_READ_NBIT_AND_BTB_AND_META**. Otherwise, the **twolev_state_pht_address** address index is incremented by '1' and the process is repeated by returning to state **STATE_2LEV _READ_PHT**.

23. **STATE_TOUR_READ_NBIT_AND_BTB_AND_META**: The final job of the state FSM is to export a state for the Tournament Branch Predictor. All three of the Tournament's BTB, SEL and BHT mirror memories have the same **MEM_DEPTH** number of words, so, by using a single dedicated address index **tournament_state_btb_bht_ and_meta_address**, the FSM can traverse the BHT, SEL and BTB mirror memory modules. In this state, the FSM issues a read request to the SR mirror memory and advances to state **STATE_TOUR_WAIT _NBIT_AND_BTB_AND_META**.

24. **STATE_TOUR _WAIT_NBIT_AND _BTB_AND_META**: The data requested in state **STATE _TOUR_READ _NBIT_AND_BTB_AND_META** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_TOUR _PROCESS _NBIT _AND_BTB_AND_META**.

25. **STATE_TOUR_PROCESS_NBIT_AND_BTB_AND_META**: In this state the data from the Tournament's BTB, BHT and SEL mirror memories are received and are used to synthesize the 512-bits of the **state_line_to_write** register. These data are the s-bit predictor from the SEL memory, the n-bit predictor from the BHT memory and the branch opcode and target PC from the BTB, are exported as four 32-bit integers. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_TOUR_WRITE_NBIT_AND _BTB_AND_META**. The format of a **state_line_to_write** is depicted in figure 7.12.

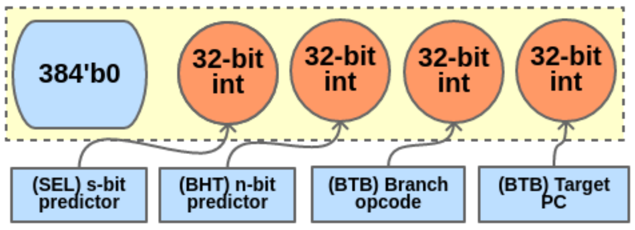


Figure 7.12 – The format of a **state_line_to_write** containing the Tournament Predictor's BTB, BHT and SEL memory state data.

26. **STATE_TOUR_WRITE_NBIT_AND_BTB_AND_META**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer

and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire Tournament Predictor's mirror BTB, BHT and SEL mirror memories (**MEM_DEPTH # of words**). If this is the case, the **tournament_state_btb_bht_ and_meta_address** address index is set to zero and the FSM proceeds to state **STATE_TOUR_READ_SR**. Otherwise, the **tournament_state_btb_bht_ and_meta_address** address index is incremented by '1' and the process is repeated by returning to state **STATE_TOUR_READ_NBIT_AND_BTB_AND_META**.

27. **STATE_TOUR_READ_SR**: The 2nd step is to export a state for the Tournament Predictor's SR mirror memory. In this state, the State FSM issues a read request to the Tournament's SR mirror memory, using a dedicated **tournament_state_sr_address** index and advances to state **STATE_TOUR_WAIT_SR**.

28. **STATE_TOUR_WAIT_SR**: The data requested in state **STATE_TOUR_READ_SR** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_TOUR_PROCESS_SR**.

29. **STATE_TOUR_PROCESS_SR**: In this state the data from the Tournament Predictor's SR mirror memory are received and are used to synthesize the 512-bits of the **state_line_to_write** register. Each line from the SR represents a single history register whose data are exported using a single 32 bit integer. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_TOUR_WRITE_SR**. The format of a **state_line_to_write** is depicted in figure 7.13.

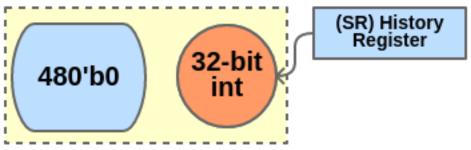


Figure 7.13 – The format of a **state_line_to_write** containing a history register for the Tournament Predictor's state data.

30. **STATE_TOUR_WRITE_SR**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire Tournament's mirror SR mirror memory (**N # of words**). If this is the case, the **tournament_state_sr_address** address index is set to zero and the FSM proceeds to state **STATE_TOUR _READ_PHT**. Otherwise, the **tournament_state_sr_address** address index is incremented by '1' and the process is repeated by returning to state **STATE _TOUR_READ_SR**.

31. **STATE_TOUR_READ_PHT**: The third and final step is to read the Tournament Predictor's PHT mirror memory using a dedicated **tournament_state_pht_address** address index. In this state, the FSM issues a read request to the PHT mirror memory and advances to state **STATE_TOUR _WAIT_PHT**.

32. **STATE_TOUR_WAIT_PHT**: The data requested in state **STATE_TOUR_READ_PHT** will be available after 1CC, so, using this state the FSM just waits for 1CC before moving to state **STATE_TOUR_PROCESS_PHT**.

33. **STATE_TOUR_PROCESS_PHT**: In this state the data from the Tournament's PHT mirror memory are received and are used to synthesize the 512-bits of the **state_line_to_write** register. The k-bit predictor contained inside a single line from the PHT mirror memory is exported using a single 32 bit integer. After formatting the **state_line_to_write**, the FSM proceeds to state **STATE_TOUR_WRITE_PHT**. The format of a **state_line_to _write** is depicted in figure 7.14.

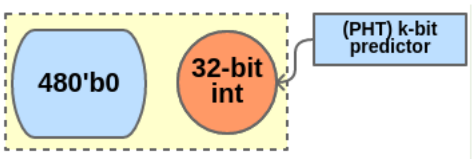


Figure 7.14 – The format of a **state_line_to_write** containing the PHT's k-bit predictor for the Tournament Predictor's state data.

34. **STATE_TOUR_WRITE_PHT**: In this state, the FSM issues a DDR write request to write the **state_line_to_write** register's 512-bits to the result buffer and the printing index **state _index _for_the_ddr_writes** is incremented by '1'. Now, the FSM has to check if it has traversed the entire Tournament's mirror PHT mirror memory (**PHT_MEM_DEPTH # of words**). If this is the case, the **tournament_state_pht_address** address index is set to zero and the FSM proceeds to state **STATE_END**. Otherwise, the **tournament_state_pht_address** address index is incremented by '1' and the process is repeated by returning to state **STATE _TOUR_READ_PHT**.

35. **STATE_END**: This is the final state of the State FSM that concludes the state export by informing the SW API that the AFU is finished via the **'write_finished'** signal. The FSM will remain in this state unless a new **'write_start'** signal or **reset** is received.

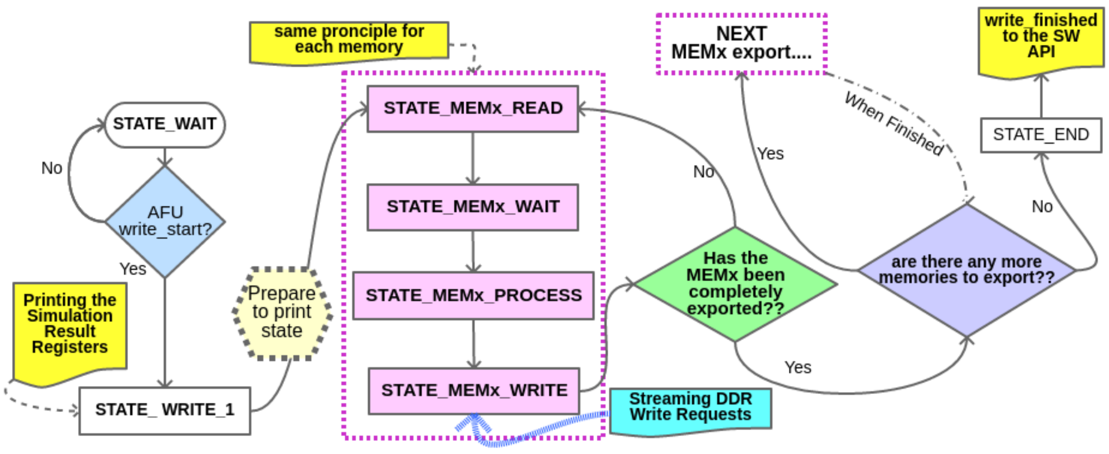


Figure 7.15 – The State FSM Exporting Procedure.

7.4 ASE Simulation

The ASE simulation has already been mentioned in section 4.3, however, all the details will be discussed in this section. With the project's working directory set up as described in section 7.1, the ASE simulation is possible with a few discrete steps.

The first step is to configure a build environment. If the OPAE is installed on a private directory, set the environment variable **OPAE_ INSTALL_PATH** to point to the OPAE installation directory, otherwise do nothing. Also, the developer selects the FPGA class out of 4 classes described in [42]. The FPGA class-of-interest is the **fpga-bdx-opae** which is the one used for simulating the **Broadwell Xeon CPUs (E5-2600v4) with an integrated in-package Arria 10 GX1150 FPGA (10AX115U3F45E2SGE3)**. This configuration is done by issuing:

```
$ source /export/fpga/bin/setup-fpga-env  fpga-bdx-opae
```

Afterward, the developer issues the **qsub-sim** which is used to start an **interactive session** on generic batch machines. When inside qsub-sim, navigate to the project's **hw** directory. OPAE has a dedicated script to initiate the creation of a simulation build directory (build_sim). This is done by executing:

```
$ afu_sim_setup -s rtl/sources.txt build_sim
```

As mentioned already, ASE simulation requires 2 processes. **Inside qsub-sim, the use of tmux is essential in order to split the terminal into 2 discreet screens, one for SW and one for the RTL.** For more information about tmux visit [43]. For the RTL process navigate into the build_sim directory and issue:

```
$ make && make sim
```

This will initiate the 'RTL environment'. While this process is running, switch to the SW pane, navigate to the **sw** directory and prepare the SW simulation. First, issue 'make' for the SW API. Then export the **ASE_WORKDIR** environment variable that points to the /build_sim/work directory and initiate the SW simulation. To do so, issue:

```
$ export ASE_WORKDIR = <path to project/directory/hw/build_sim/work>
$ with_ase ./<sw_api_name> <sw_api_parameters>
```

This step concludes the ASE simulation for the simulator platform. For more detailed information visit [42] and Appendix A. The SW parameters are depicted in figure 7.6.

7.5 Bitstream Generation and FPGA Execution

The Bitstream Generation and RTL synthesis have already been discussed in section 4.4, however, all the details will be discussed in this section. The synthesis starts just like before, by configuring a build environment. After this configuration, navigate to the project's hw directory and initiate the creation of a synthesis build directory (build_fpga). This is done by executing:

```
$ afu_synth_setup -s rtl/sources.txt build_fpga
```

Now, enter the build_fpga directory and initiate the synthesis of the green bitstream. This is done by issuing:

```
$ qsub-synth
```

The blue bitstream that contains the FIU is pre-configured and pre-loaded to the FPGA. The green bitstream will be the one used for partial reconfiguration of the FPGA containing the accelerator. The developer can monitor the green bitstream's synthesis process by issuing:

```
$ tail -f build.log
```

The synthesis process can take multiple minutes or hours. After its completion, the user opens a shell on the FPGA system of the selected class by issuing **qsub-fpga**. While in this shell, the developer needs to export the **PBS_O_WORKDIR** environment variable that points to the /build_fpga directory:

$ export **PBS_O_WORKDIR** = <path to project/directory/hw/build_fpga>

Now, enter the build_fpga directory and load the green bitstream to the FPGA. This is possible by issuing:

$ fpgaconf **<synthesis_name>.gbs**

After this is completed, the only thing left to do is to navigate to the **sw** directory, '**build/make**' the SW API and run the program by issuing:

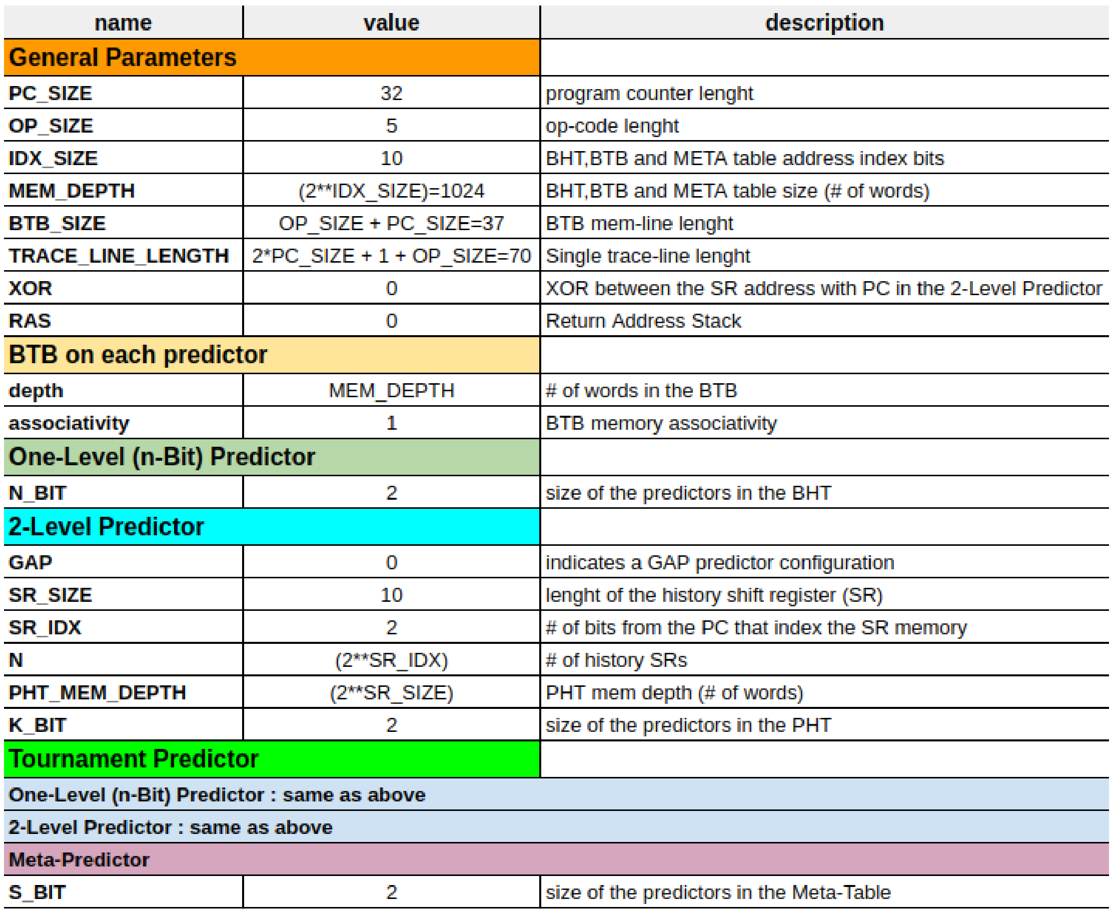
$ **./<sw_api_name> <sw_api_parameters>**

This concludes the bitstream generation and the whole simulation process running on the FPGA. For more detailed information visit [42] and Appendix B. The SW parameters are depicted in figure 7.6.

Chapter 8
Results and Data Analysis

8.1 The Branch Predictor (BP) Simulator Results

The first series of experiments were conducted in order to test all of the Branch Predictor models on the HARP Platform. The experiments provided a great deal of information as regards to timing, efficiency, resource utilization and also the ideal size of an input trace buffer. The configuration parameters for all BP models can be seen in figure 8.1. All of the parameter names and their explanations have already been described in section 6.8.2. Once again, as in section 6.8.2, for the 2-Level Predictor, the PAg was chosen as it represents the more generic version of a 2-Level Adaptive Predictor. Also, no model has a Return Address Stack (RAS=0).



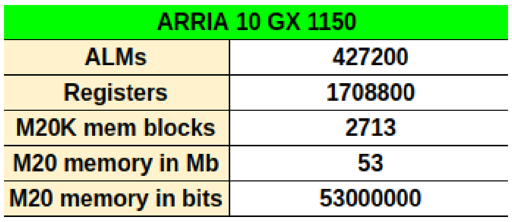
name	value	description
General Parameters		
PC_SIZE	32	program counter lenght
OP_SIZE	5	op-code lenght
IDX_SIZE	10	BHT,BTB and META table address index bits
MEM_DEPTH	(2**IDX_SIZE)=1024	BHT,BTB and META table size (# of words)
BTB_SIZE	OP_SIZE + PC_SIZE=37	BTB mem-line lenght
TRACE_LINE_LENGTH	2*PC_SIZE + 1 + OP_SIZE=70	Single trace-line lenght
XOR	0	XOR between the SR address with PC in the 2-Level Predictor
RAS	0	Return Address Stack
BTB on each predictor		
depth	MEM_DEPTH	# of words in the BTB
associativity	1	BTB memory associativity
One-Level (n-Bit) Predictor		
N_BIT	2	size of the predictors in the BHT
2-Level Predictor		
GAP	0	indicates a GAP predictor configuration
SR_SIZE	10	lenght of the history shift register (SR)
SR_IDX	2	# of bits from the PC that index the SR memory
N	(2**SR_IDX)	# of history SRs
PHT_MEM_DEPTH	(2**SR_SIZE)	PHT mem depth (# of words)
K_BIT	2	size of the predictors in the PHT
Tournament Predictor		
One-Level (n-Bit) Predictor : same as above		
2-Level Predictor : same as above		
Meta-Predictor		
S_BIT	2	size of the predictors in the Meta-Table

Figure 8.1 – Branch Predictor parameters for each model.

A very useful mathematical equation to calculate the Simulation Time in msec, given the number of Clock Cycles (CCs) and the clock frequency in MHz, is given below.

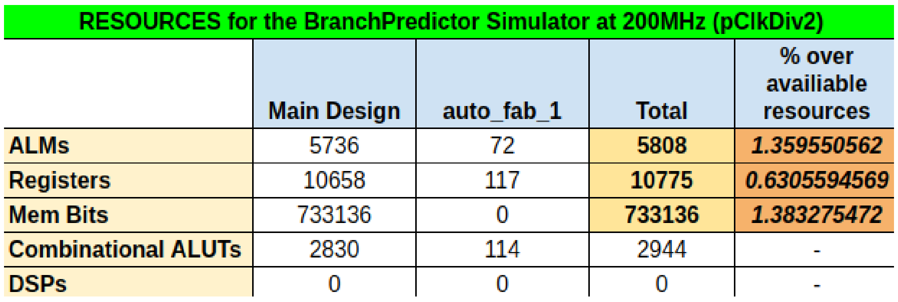
$$SimulationTime = \frac{\frac{Number\_of\_CCs}{Clock\_Frequency\_in\_MHz}}{1000}(msec)$$

The HW integrated design includes the AFU and a single **16-slot FIFO** along with all the predictor models described in chapter 6. The overall resource utilization of the Branch Predictor Simulator platform described in the **bdw_503_pr_afu_synth.syn.rpt** output file, along with the total available resources of the Arria 10 GX 1150 FPGA is shown in figures 8.2 and 8.3.



ARRIA 10 GX 1150	
ALMs	427200
Registers	1708800
M20K mem blocks	2713
M20 memory in Mb	53
M20 memory in bits	53000000

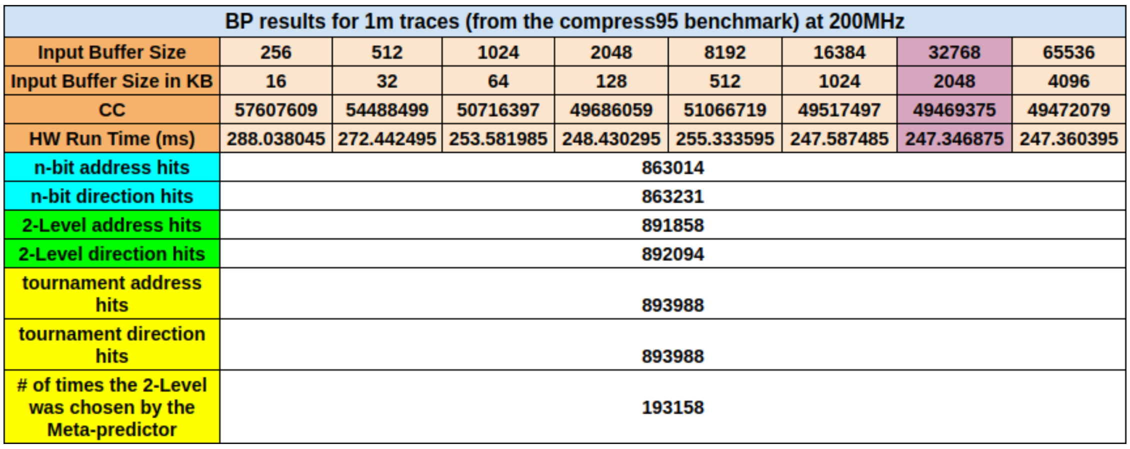
Figure 8.2 – Intel Arria 10 GX 1150 Available Resources [16].



RESOURCES for the BranchPredictor Simulator at 200MHz (pClkDiv2)				
	Main Design	auto_fab_1	Total	% over availiable resources
ALMs	5736	72	5808	*1.359550562*
Registers	10658	117	10775	*0.6305594569*
Mem Bits	733136	0	733136	*1.383275472*
Combinational ALUTs	2830	114	2944	-
DSPs	0	0	0	-

Figure 8.3 – Resource Utilization of the Branch Predictor Simulator.

At this point, the simulator includes only the Branch Predictor Models. The results from the HARP hybrid platform execution can be seen in figures 8.4, 8.5 and 8.6. To guarantee that the AFU will write the correct results back to the DDR, a **10ms** delay was integrated to the SW API between the AFU's '**write_finished**' response and the API's log file generation. The simulations were executed with different input buffer sizes for **1000000**, **14361067** (entire trace from the SPEC95 compress95 benchmark) and **80274735** (entire trace from the SPEC95 go benchmark) traces. The design successfully meets timing at **200MHz**, so the overall simulated time is calculated accordingly. Address and direction hits are depicted to verify the correct functionality of the accelerator when selecting different input buffer sizes. On all figures, the best run is highlighted in **magenta**.



BP results for 1m traces (from the compress95 benchmark) at 200MHz								
Input Buffer Size	256	512	1024	2048	8192	16384	32768	65536
Input Buffer Size in KB	16	32	64	128	512	1024	2048	4096
CC	57607609	54488499	50716397	49686059	51066719	49517497	49469375	49472079
HW Run Time (ms)	288.038045	272.442495	253.581985	248.430295	255.333595	247.587485	247.346875	247.360395
n-bit address hits	863014							
n-bit direction hits	863231							
2-Level address hits	891858							
2-Level direction hits	892094							
tournament address hits	893988							
tournament direction hits	893988							
# of times the 2-Level was chosen by the Meta-predictor	193158							

Figure 8.4 – Execution Results for 1000000 traces from the SPEC95 compress95 benchmark for all Branch Predictor Models.

BP results for 14361067 traces (total compress95 benchmark) at 200MHz								
Input Buffer Size	512	16384	32768	131072	262144	524288	1048576	2097152
Input Buffer Size in KB	32	1024	2048	8192	16384	32768	65536	131072
CC	601827231	516244011	515993935	518163295	514275777	580420201	581276287	585833351
HW Run Time (ms)	3009.136155	2581.220055	2579.969675	2590.816475	2571.378885	2902.101005	2906.381435	2929.166755
n-bit address hits	12909842							
n-bit direction hits	12912759							
2-level address hits	13210170							
2-level direction hits	13213251							
tournament address hits	13332674							
tournament direction hits	13335692							
# of times the 2-Level was chosen by the Meta-predictor	2704575							

Figure 8.5 – Execution Results for 14361067 traces from the SPEC95 compress95 benchmark for all Branch Predictor Models. . The address/direction hit results are a perfect match with the BP models results when tested as bare-metal on the ModelSim Simulator as shown in figure 6.8

BP results for 80274735 traces (total go benchmark) at 200MHz								
Input Buffer Size	8192	262144	524288	1048576	2097152	4194304	8388608	16777216
Input Buffer Size in MB	0.5	16	32	64	128	256	512	1024
CC	3003416449	2939307559	2764422775	2771785611	2784436685	2792177461	2786633655	2790514383
HW Run Time (sec)	15.017	14.697	13.822	13.859	13.922	13.961	13.933	13.953
n-bit address hits	56518495							
n-bit direction hits	63548011							
2-level address hits	49063567							
2-level direction hits	55861740							
tournament address hits	56710123							
tournament direction hits	63780350							
# of times the 2-Level was chosen by the Meta-predictor	21740113							

Figure 8.6 – Execution Results for 80274735 traces from the SPEC95 go benchmark for all Branch Predictor Models.

The final line of simulation data on figures 8.4, 8.5 and 8.6, represents the number of times the Meta-predictor would have selected the 2-Level's prediction over the n-bit's. Of course, there are many cases where the 2-Level prediction was the same as the n-bit's. **However, this value only represents how many times the Meta-predictor would have selected the 2-Level's prediction over the n-bit's.**

This concludes the first part of the experiments regarding only the Branch Predictor (BP) Simulator loaded onto the HARP hybrid platform. **This system is provided as a standalone BP HW simulator, alongside the Full State Simulator described in this dissertation**.

8.2 The Combined BR and Cache Simulator Results

The second series of experiments involved the **integration of the Cache Model**, inside the already functional BP simulator. After conducting the first set of experiments, the optimal input buffer sizes for the BP simulator has already been determined. A FIFO identical to the one used for the BP simulator, is added to the design for storing the cache-lines fetched from the Cache's Read FSM. The parameters used to configure the Cache Model along with their values, selected for the second execution phase, are depicted in figure 8.7 and described in section 5.3.

Cache Model Parameters		
name	**value**	**description**
MRU_OR_LRU	1	Replacement policy index: LRU=1 and MRU==0
WRB_OR_WRT	0	Update policy index: WRT=1 and WRB==0
WAYS	4	set associativity
DTAG_W	22	tag size
DIDX_W	5	index size
ADDR_W	DTAG_W+DIDX_W	Cache request width
SIZE	2**DIDX_W	cache memory depth (# of CLs)
RPL_IDX_SIZE	$clog2(WAYS)	size of the replacement indexes

Figure 8.7 – Cache Model Parameters and their Values for the 2nd HW Execution Phase.

All the parameters for the BPs are the same as in phase 1. The overall resource utilization, described in the **bdw_503_pr_afu_synth.syn.rpt** output file, for the Combined BP and Cache simulator on the HARP platform, is given in figure 8.8. These numbers include a 5-FSMs AFU (without the State FSM), all of the BP models, the Cache model and the 2 FIFOs.

RESOURCES for the BP and Cache Combined Simulator at 200MHz (pClkDiv2)				
	Main Design	**auto_fab_1**	**Total**	**% over availiable resources**
ALMs	8028	93	**8121**	***1.900983146***
Registers	14380	140	**14520**	***0.8497191011***
Mem Bits	744784	0	**744784**	***1.40525283***
Combinational ALUTs	4728	133	4861	-
DSPs	0	0	0	-

Figure 8.8 – Overall resource utilization of the Combined BP and Cache simulator.

As in phase 1, the BPs are executed for the same **14361067** trace input file from the SPEC95 compress95 benchmark. The Cache model is executed using the **10000000** trace file generated for the bare metal simulation as described in section 5.5. The simulation results after the HW execution are identical to the ones exported from the bare metal simulation of each individual model and can be seen in figure 8.9. **However, the HW Run Time is not the best time, but the average execution time that was exported after multiple runs**.

BP and Cache results for 14361067 traces (total compress95 benchmark) and 10000000 traces (total cache trace) at 200MHz	
Input Buffer Size	**262144**
Input Buffer Size in KB	**16384**
Additional API delay for cache model (msec)	**500**
CC (avg.)	**836425051**
HW Run Time (ms)	**4182.125255**
n-bit address hits	**12909842**
n-bit direction hits	**12912759**
2-level address hits	**13210170**
2-level direction hits	**13213251**
tournament address hits	**13332674**
tournament direction hits	**13335692**
# of times the 2-Level was chosen by the Meta-predictor	**2704575**
Cache Read Hits	**6**
Cache Write Hits	**5**

Figure 8.9 – Execution Results for 14361067 traces from the SPEC95 compress95 benchmark for all Branch Predictor Models and 10000000 traces from the trace file reported in section 5.5 for the Cache Model.

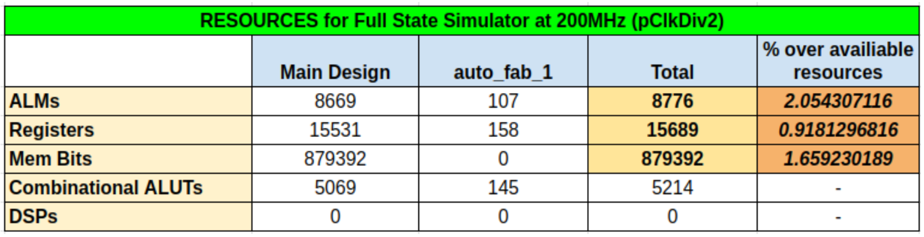
This concludes the second part of the experiments regarding the combined Branch Predictor (BP) and Cache Simulator loaded onto the HARP hybrid platform. **This system is provided as a standalone BP+Cache HW simulator, alongside the Full State Simulator described in this dissertation**.

8.3 The Full State Simulator Results

The final set of results contain all the information from executing the Full State Simulator on the HARP hybrid platform. **The Full State Simulator is an extension of the BP+Cache simulator** described is section 8.2. It has the same capabilities and functionality, but, it is also **capable of printing the system's state inside the result log file, along with the simulation results**. The necessary logic and hardware, that were added to complete this simulator, are describe in detail in section 7.3.5.

The Full State Simulator executed in this step, contains the BPs and Cache Models whose parameters are the same as the ones used to execute the **BP Simulator** and the **BP+Cache Simulator** described in sections 8.1 and 8.2 respectively. This was intentional, because a correct execution of a **full state simulation** can be verified when the simulation results are a perfect match with the simulation data delivered in the previous sections.

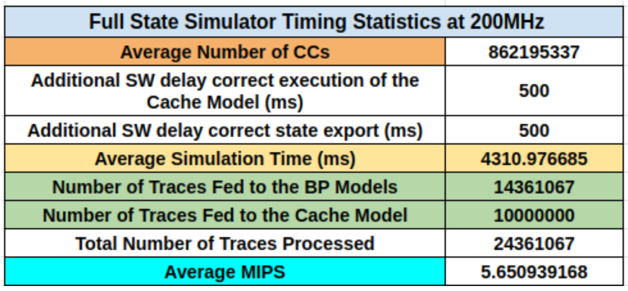
The added logic, obviously means an increase in the HW needed to deploy this extended simulator platform. The overall resource utilization of the Full State Simulator is depicted in figure 8.10.



RESOURCES for Full State Simulator at 200MHz (pClkDiv2)				
	Main Design	auto_fab_1	Total	% over availiable resources
ALMs	8669	107	**8776**	***2.054307116***
Registers	15531	158	**15689**	***0.9181296816***
Mem Bits	879392	0	**879392**	***1.659230189***
Combinational ALUTs	5069	145	5214	-
DSPs	0	0	0	-

Figure 8.10 – Overall resource utilization for the Full State Simulator.

After **~100** executions on the Full State Simulator, average timing result estimations were exported from the simulation data. These data include the **'average number of clocks after reset'** and the **'average simulation time'**. Depending on the data being processed by the simulator, the **'average MIPS'** parameter was added, stating the number of instructions that the Full State simulator can handle in 1 second. **The 'average MIPS' was calculated by including the AFU processing time as well as the state export, so, it is an key result parameter for the entire system behavior**. These timing results are depicted in figure 8.11.



Full State Simulator Timing Statistics at 200MHz	
Average Number of CCs	**862195337**
Additional SW delay correct execution of the Cache Model (ms)	**500**
Additional SW delay correct state export (ms)	**500**
Average Simulation Time (ms)	**4310.976685**
Number of Traces Fed to the BP Models	**14361067**
Number of Traces Fed to the Cache Model	**10000000**
Total Number of Traces Processed	**24361067**
Average MIPS	**5.650939168**

Figure 8.11 – Timing Statistics for the Full State Simulator.

After the AFU is finished, the SW API is responsible for printing the system state in a readable and user-friendly format. The simulation results regarding the hits and misses of the BPs and Cache are the same as in sections 8.1 and 8.2 and are printed at the top of the simulation log file. The following section contains the data from the cache memory module. As mentioned in section 7.3.5, an 8-way cache memory is printed each time. Depending on the Cache Model's configuration, all unused ways will be printed as zeros. The following sections are the contents of the memory modules used by the One-Level, 2-Level and Tournament Predictors. The format of the data exported from the AFU is described in section 7.3.5, however, the format of the Full State Simulator's log file that includes the system's state is also depicted in figure 8.12.

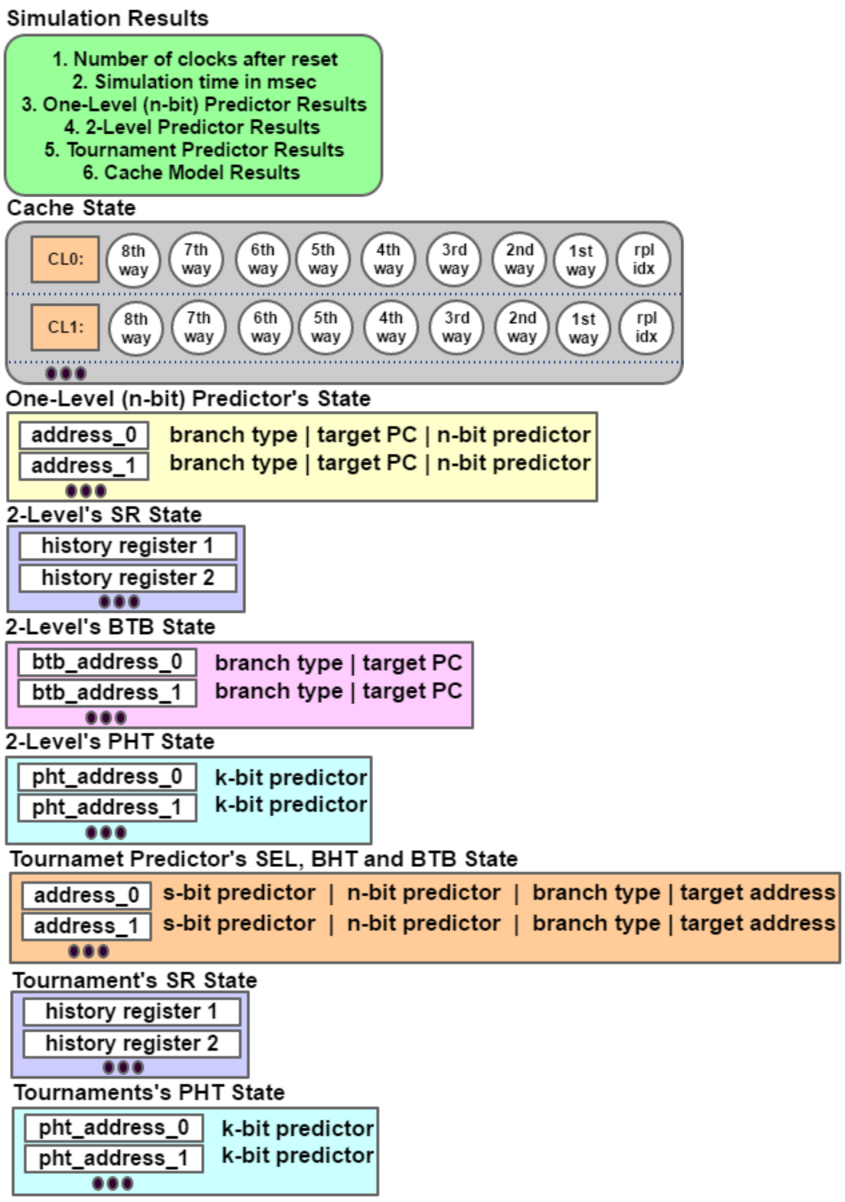


Figure 8.12 – Format of the data included in the Full State Simulator's log file.

The execution of the Full State Simulator concludes the final part of the experiments regarding the integration of the Branch Predictor (BP) and the Cache Simulator onto the HARP hybrid platform, that are also capable of exporting the system's state. **This system is provided as the third and final standalone Full_State_Simulator (FSS) HW simulator, alongside the other two simulators described in this dissertation**.

8.4 Analysing the Data

Analyzing the execution and simulation results is arguably one of the most important sections in every research dissertation. The results exported from the experimental process provide information about the performance of the Hardware Models but also about the capabilities and behavior of the HARP target platform. By monitoring the data exported from the Simulator and the overall system operation, many useful results and conclusions can be reported. This data analysis is presented in the form of a series of bullets for better clarity of each observation.

- **Execution-Simulation Comparison**: The first thing to highlight is that all bare-metal simulation results, regarding the BP and Cache models, matched perfectly with the execution results of these models when integrated on the HARP HW platform. Each individual model has been tested and verified as a bare-metal standalone HW module using the SimpleScalar 6.7 and the DineroIV 5.7 'golden models'. Therefore, **this identicality between simulation and execution results verifies the correct functionality of every single module of this simulator platforms**.

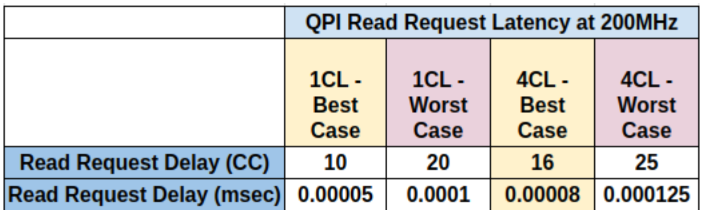
- **Sub-optimal, Yet Fast**: The BP models, as well as the Cache model, are all **prototypes** and can receive multiple upgrades and optimizations like pipelining and prefetching. However, the current performance of this simulator is already very promising, with multiple simulation results and system-state data being exported at the fraction of the time as from similar simulator platforms. Furthermore, given an accurately build SW API, this particular simulators are designed to run in coordination with a fast functional full system simulator on specified timestamps, meaning that they can export heavy-duty simulation results while the fast SW simulator continues unobstructed.

- **HARP Capabilities**: The QPI is the only channel utilized, with the maximum 4CL (2048-bit-batch) DDR read requests and streaming 512-bit DDR write requests. However, **the maximum throughput of the QPI has not been reached** (this will be discussed later on). Furthermore, **the HARP platform is heavily underutilized**, with less than **3%** of the of its resources being used for the BP and Cache simulators. The integration of the ARM modules, that complete the Full System Simulator (2.8) on the HARP hybrid platform, will definitely increase the system requirements. However, there will still be a great amount of unutilized HW meaning that multiple instances of the Full State Simulator or the Full System Simulator can be integrated, to execute multiple system simulations, at different timestamps, in parallel. This will translate into delivering great amounts of speedup and overall platform efficiency (**parallel simulations => x2, x3, etc.**). Also, **the HW clock can be pushed close to 400MHz** by using different development strategies and the provided 'uClk_usr' OPAE parameter that allows the user to specify a system clock different from the pre-specified 100, 200 and 400MHz.

- **The Input Buffer's Size**: The experimental procedure highlighted the fact that **selecting the correct size of the input buffer for the BP models can lead to increased performance and simulator efficiency**. At first, selecting a large input buffer seemed like the best choice, given that the SW API would need to 'refill' the input buffer and recall the AFU for fewer times. However, the execution results printed a different pattern, that indicated that the best size was actually 16MB for the 109MB trace and 32MB for 612MB the trace file. The conclusion is that selecting **the best size will eventually come from experimentation** and the choice is not so obvious. **One of the reason might be the AFU's capability to access**

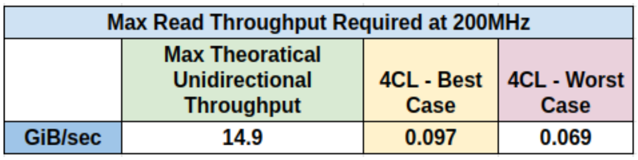
certain DDR address ranges (closer to a page size) better. The cache model's input buffer size cannot be configured for the current simulator integrations and might be one of the reasons why this model is much slower than the BPs. It can be developed to boost the cache model's efficiency and is being encouraged as future work.

- **Regarding the DDR Read Requests**: The execution phase, highlighted **the importance of multiple CL-read requests**. A **single CL** (512-bit) read request is handled after **10-20 CCs** for a 200MHz HW design, whereas, **quadruple CL** (4*512 = 2048-bit) read requests are handled after **16-25 CCs**, meaning an estimated **3x increase** in bits read per second. This is a major improvement in the simulator model's efficiency that requires an input FIFO to store all the data on arrival, as well as a reorder buffer, because the data requested are received **out-of-order**.



	QPI Read Request Latency at 200MHz			
	1CL - Best Case	1CL - Worst Case	4CL - Best Case	4CL - Worst Case
Read Request Delay (CC)	10	20	16	25
Read Request Delay (msec)	0.00005	0.0001	0.00008	0.000125

Figure 8.13 – Best and Worst QPI Latency achieved with single (512-bit) and quadruple (2048-bit) read requests to the DDR.



	Max Read Throughput Required at 200MHz		
	Max Theoratical Unidirectional Throughput	4CL - Best Case	4CL - Worst Case
GiB/sec	14.9	0.097	0.069

Figure 8.14 – Current QPI Utilization achieved for Reads, compared to the maximum theoretical unidirectional throughput

- **Regarding the DDR Write Requests**: The CCI-P is capable of handling streaming DDR write requests, delivering a valid common signal whenever a request is handled. **Note that a write-valid signal (sRx.c1.rspValid) is received by the AFU when a DDR write request is handled, not when the data has been successfully written to the DDR**. That is why the developer should add a SW delay between the printing of the simulation results and the completion of the Write/State FSMs.

- **No FPGA Memory Reset**: On this system, the SW API does not issue a reset signal to the AFU. This means that between runs on the same HW loaded onto the HARP platform, the Arria 10 Altera syncram memory modules are not reset. So, each new simulation starts by having all of its memory modules filled with data from the previous execution. In order to export accurate simulation results and an actual system state, the user has to execute only one time after loading the HW platform on HARP. In general, **for every new execution, for example with different SW API parameters, the FPGA needs to be reprogrammed**. Luckily, this is nothing but an inconvenience that can be easily solved as future work.

- **The 500ms SW Delays**: The Cache simulator model runs a little slower than the BP models. For that reason, the SW API needs to issue a '**usleep**' for at least **400-500ms** before issuing the 'write-start' signal to the AFU, in order for the Write or State FSM to print the correct simulation results. This 500 ms delay can be avoided by developing a faster Cache simulator model that utilizes memory prefetching and pipelining. This work is proposed as future work. Also, the State FSM requires **another ~500msec** in order to print the system's state, thus increasing the execution time even more.

- **Duplicate State Memories**: In order to export the system state, **mirror memory modules** were used to store the system's internal memory contents. The simulator can be upgraded by using interfaces to access the internal memory modules. This will result in reduced resource utilization and the removal of the AFU's control that is responsible for updating the duplicate memories. This work is proposed as future work.

- **The Importance of a System State**: One of the biggest advantages of the Full State simulator is the fact that it can export system states in an efficient manner. **The state is exported as an easy-to-read log file alongside all the simulation data**. **Each memory module has its own private section, with discrete contents and explanation about the format of each section.** This means that a user/developer can manage these data to **warm-up** a new timing or functional simulation from either of the timestamps that the states were exported from. They can also be used to **monitor**, the contents of the memories to validate the functionality of a system or even manipulate specific data in order to view certain system behavior. Of course, the system states can also be used **as backups** for very large simulation procedures that take multiple hours, days or even months to finish. Having these states means that **a process can be restarted from various checkpoints in time, rather than from the start**, which results in massive time reduction of the execution phase.

- **Comparing the results with Similar Platforms**: The data comparison is performed by using the results exported from the three execution phases described in sections 8.1, 8.2 and 8.3 along with the results from the PROTOFLEX [26] paper regarding the **Simics-fast** and the **Simics-trace** simulators. These results regarding the MIPS capabilities of each system is depicted in figure 8.15.

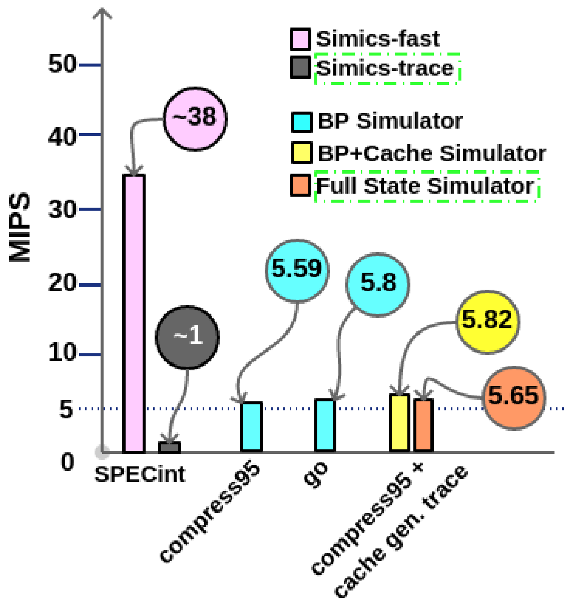


Figure 8.15 – MIPS capabilities of the **BP**, the **BP+Cache** and the **Full State** simulators compared to the **Simics-fast** and the **Simics-trace** simulators.

The results of interest are the ones regarding the **Simics-trace** and the **Full State Simulator**. The Full State Simulator appears to have a **~5.6x** speedup compared to the Simics-trace Simulator. The fact that **the state export does not affect the performance of the Full State Simulator, compared to the BP and the BP+Cache Simulators**, indicates that, by integrating the optimizations proposed in this section and chapter 9, the system may have more potential than the one presented so far. Estimated results after some key optimizations, can be found in the following bullets.

- **System Estimation No1**: The suggestion for optimal use of the Full State Simulator is that it should be **executed in parallel** with a fast functional simulation as depicted in figure 9.1. That way, if not all, the majority of the processing time will be 'hidden' by the SW execution. A state of the art SW API will be responsible of exporting trace data from the functional simulator at different timestamps throughout a simulation process, possibly even at run time. These data can then be 'fed' to multiple HW modules (simulator engines) loaded onto the FPGA and various system checkpoints can be exported in parallel, maximizing the potential of the HARP platform.

- **System Estimation No2**: As in the majority of the accelerator architectures and platforms, a bottleneck may be presented in the I/O transactions between the FPGA and the SW or DDR. From figure 8.13 the current system's maximum **achieved** throughput is *approximately* **0,097GiB/sec => 99,32 MiB/sec**. The maximum QPI throughput is registered at **8 GT/s**, meaning an approximate **14,9 GiB/s**.

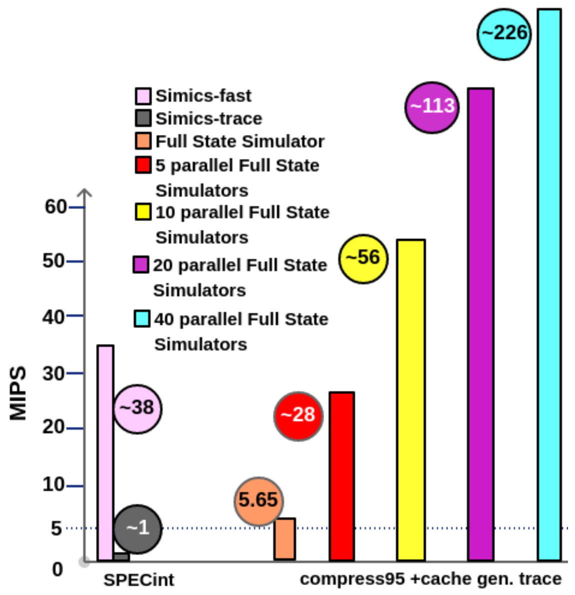


Figure 8.16 – The Estimated MIPS capabilities of the **5**, **10**, **20** and **40** simultaneously executed **Full State Simulators** compared to the **Simics-fast** and the **Simics-trace** simulators.

From the data exported and calculated so far (8.10), **more than 40 Full State Simulators can run simultaneously on the HARP hybrid platform, without congesting the QPI or exceeding the FPGA HW resources**. All simulators can work independently, processing traces from the same or different input trace files. The input trace files could also be from various benchmarks. This means that **multiple checkpoints** and simulation statistics can be exported simultaneously **for different timestamps** and **for different programs** and system configurations.

From the experimentation phase, the speedup from using multiple Full State Simulators, has been calculated to be linear. So, by combining all these data, figure 8.16 depicts the estimated MIPS processing capabilities of 5, 10, 20 and 40 Full State simulator engines, running in parallel on the HARP platform. The implementation of 5 parallel simulators, is the one that is currently under development, because the full system simulator will require enough HW resources to host multiple ARM models and other peripherals for the full system architectural simulator 2.8. All these analysis is done for 2 reasons:

1. **The first one** is that the use of Full State simulators on HARP is a highly efficient and robust implementation of an accurate Cache and Branch Predictor simulator, that also highlights the capabilities of the HARP hybrid chip.
2. **The second one** is that this system does not require much HW resources and can run independently and efficiently in coordination with similar other HW components. This means that it **will not be a bottleneck** or hindrance when integrated to the full system architectural simulator.

- **System Estimation No3**: From the Architectural and experimental evaluation process, it is concluded that the current system can be parallelized between (worst) **74%** and (best) **83%**. Based on Amdahl's Law the maximum speedup for the optimized system can be:
  - **best**: **5.96x** => **33.67 MIPS**
  - **worst**: **3.83x** => **21.64 MIPS**

Of course, these optimized modules can also work independently and in parallel with other modules, loaded simultaneously on the Harp platform. The estimated performance of these systems is depicted in figure 8.17. These theoretical estimations can be used to determine that this system's efficiency has much room for improvement.

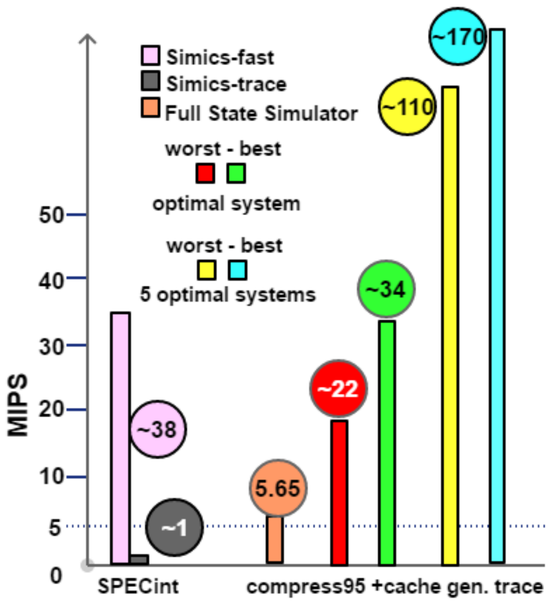


Figure 8.17 – The Estimated MIPS of one or multiple optimized and parallelized **Full State** simulators compared to the **Simics-fast** and the **Simics-trace** simulators.

Chapter 9
Conclusion and Future Work

This dissertation was all about utilizing high-performance hardware that can simulate a system's Cache and Branch Prediction accurately and efficiently. Arguably it is a great approach towards scalable and accelerated full system simulators using FPGAs. **3 individual simulators** are delivered capable of simulating 3 key Branch Predictor Models, the Cache memory, and also, exporting the system's state. The system simulator is **accurate**, **validated** by golden models and **efficient** in terms of HW resource utilization and time consumption.

Being one of the first works on the HARP XEON-FPGA Hybrid Chip and the first project developed by the Technical University of Crete, compared to previous platforms, it performs great and has room for improvement. It is but the first step towards a true, full system simulator that will be able to run multiple simulations in parallel, thus providing simulation data and system states faster than any other previous simulator platform. It is also a work that provided great insight on Intel's Hybrid Chip and highlighted all of its strengths and behaviours. This knowledge is going to be the key for developing powerful HW on this particular platform in the future.

As mentioned in the Introduction, a HW system is much faster if it is designed specifically for a target hardware to maximize speed and resource utilization. This simulator is designed to exploit the benefits of the HARP CPU-FPGA hybrid platform and the QPI interconnect to its max. However, due to it being a prototype, there are **a number of improvements** that can push this HW design to even greater efficiency and speed:

- All the modules developed for this work can be directly integrated onto the new Skylake Xeon SP platform [15] to validate the benefits of a faster Xeon processor and differences in performance between the UPI over the QPI.

- The Full State Simulator described in this dissertation will be **combined with the ARM processor simulator** described in section 2.8, to become a complete full system simulator platform. That system is currently under development at the Technical University of Crete and it is being created in coordination with this simulator, so **the 'matching' will be possible with no major setbacks or incompatibility issues**. The system will then be able to handle full-system simulations and provide complete timing reports and system states.

- The module that controls the n-bit predictor is developed as a 3-state FSM. This results at a prediction from the n-bit every 3 CCs. It is possible that the n-bit predictor model's efficiency can be enhanced by **prefetching** the next branch instruction thus reducing the states by 1. Also, the system can be easily **pipelined** and the memories can be replaced with **dual-port syncrams** for maximum efficiency.

- The same upgrades described for the n-bit predictor can also be applied on the 2-Level and the Tournament predictor. This will result in predictions being exported every 3CCs instead of 5. **In case of a pipeline design, a prediction will be available every CC after the initiation interval that is equal to 3.**

- A configurable input buffer size is used for the BPs, but not for the Cache Model. The SW API and the AFU can be configured in such a way so that they can handle a **configurable**

input buffer for the cache model. This might possibly increase the performance of the Cache model, just like in the BP models, depending on the input trace file's size as well.

- SystemVerilog provides the capability of using interfaces to access HW modules. They can be utilized in order to **eliminate the mirror memories** that are used to export the system's state. The AFU will be able to access the internal memory modules directly, using these interfaces to export the final system state, thus decreasing the resource utilization.

- As observed from the execution and simulation results, the HARP hybrid platform's resources are heavily underutilized. This means that **multiple Full State Simulators** can be loaded onto the FPGA, creating a system that can handle **multiple simulations in parallel**. **With every new instantiation, the performance of the simulator platform is multiplied along with its efficiency and time consumption**. The huge size of the Arria 10 GX 1150 FPGA provides a suitable candidate for this type of a simulator engine. The only bottleneck might be the number of DDR Read Requests issued from these multiple simulator engines. A well developed HW control module is necessary, in order to benefit from the maximum capabilities of the HARP's QPI as well as experimentation to find the best possible **balance between data fetching efficiency and the number of Full State Simulators that the HARP platform should execute simultaneously**.
  This idea will be more effective if the Full State Simulators are executed alongside a fast functional simulator. At different checkpoints, the simulator will be 'called' and multiple log files will be exported, thus 'hiding' much of the HW execution time inside the SW execution. In detail, this idea is featured in figure

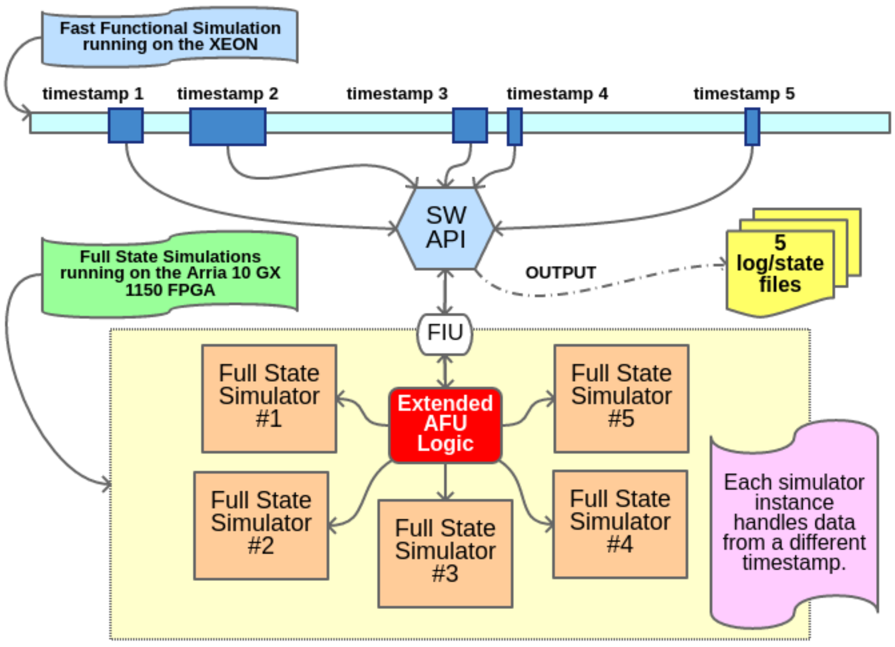


Figure 9.1 – Parallel Full State Simulator execution.

These proposals conclude this MSc Thesis regarding the Simulator Modules developed on the HARP Hybrid CPU-FPGA platform. When applied, the complete system will be able to handle full system simulations with **accuracy** and **performance** at a fraction of the time compared to similar platforms. As for the current systems, they provide an excellent baseline for future development with the 3 standalone simulators capable of handling any system simulation regarding the Branch Predictors and the Cache to monitor performance and to export system states.

Bibliography

[1] Mentor. ModelSim Logo;. https://www.digikey.ca/product-detail/en/intel/SW-MODELSIM-AE/544-2590-ND/2021454.

[2] Intel Logo;. https://rtos.com/platform/intel-fpga/.

[3] CPU-FPGA Logo;. https://www.elprocus.com/wp-content/uploads/2014/09/9-17-2014-5-05-59-PM.png.

[4] TUC Logo;. https://www.tuc.gr/index.php?id=4992.

[5] OPAE;. https://01.org/OPAE.

[6] Quartus Prime;. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[7] Intel Labs;. https://www.intel.com/content/www/us/en/research/overview.html.

[8] FORTH;. https://www.forth.gr/.

[9] Pancretan Endowment Fund;. https://www.pancretanpef.org/.

[10] Imperas and OVP;. http://www.ovpworld.org/.

[11] DineroIV;. http://pages.cs.wisc.edu/~markhill/DineroIV/.

[12] gem5;. http://gem5.org/Main_Page.

[13] Austin T. The SimpleScalar system software infrastructure tool set; 2004. [currently developed and supported by SimpleScalar LLC]. http://www.simplescalar.com/.

[14] Intel HARP;. https://software.intel.com/en-us/hardware-accelerator-research-program.

[15] Integrated Xeon-FPGA Hybrid Chip;. https://www.nextplatform.com/2018/05/24/a-peek-inside-that-intel-xeon-fpga-hybrid-chip/.

[16] Arria 10 FPGA Family;. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf.

[17] Edler J. Dinero IV: Trace-driven uniprocessor cache simulator. http://www cs wisc edu/~ markhill/DineroIV. 1994;.

[18] Todd Austin;. http://web.eecs.umich.edu/~taustin/.

[19] K.Kyriakidis, D.Theodoropoulos, A.Dollas, D.Pnevmatikatos. A full system simulator for disaggregated computing platforms and cloud data centers;. http://purl.tuc.gr/dl/dias/BF0CE2BD-47BD-48E9-B3C0-5B49CD508F61.

[20] dReDBox;. http://www.dredbox.eu/.

[21] ACACES;. http://acaces.hipeac.net/2018/index.php?page=home.

[22] FireSim for Amazon EC2 F1;. https://fires.im/.

[23] Berkeley Architecture Research Group;. https://bar.eecs.berkeley.edu/.

[24] Chisel - Constructing Hardware in a Scala Embedded Language;. https://chisel.eecs.berkeley.edu/.

[25] Chung ES, Papamichael MK, Nurvitadhi E, Hoe JC, Mai K, Falsafi B. ProtoFlex: Towards scalable, full-system multiprocessor simulations using FPGAs. ACM Transactions on Reconfigurable Technology and Systems (TRETS). 2009;2(2):15.

[26] ProtoFlex Image Archive;. https://www.archive.ece.cmu.edu/~protoflex/doku.php.

[27] Xilinx FPGAs;. https://www.xilinx.com/products/boards-and-kits.html.

[28] Wind River Simics;. https://www.windriver.com/products/simics/simics_pn_0520.pdf.

[29] COSSIM;. https://github.com/H2020-COSSIM.

[30] OMNeT++;. https://omnetpp.org/.

[31] McPAT;. https://www.hpl.hp.com/research/mcpat/.

[32] Xeon E5-2628L v4;. http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2628L%20v4.html.

[33] CCI-P;. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf.

[34] AFU and ASE;. https://opae.github.io/1.1.0/docs/ase_userguide/ase_userguide.html.

[35] OPAE;. https://github.com/OPAE.

[36] Intel's FPGA BBB;. https://github.com/OPAE/intel-fpga-bbb.

[37] Synopsys VCS;. https://www.synopsys.com/verification/simulation/vcs.html.

[38] ModelSim;. https://www.mentor.com/products/fv/modelsim/.

[39] QuestaSim;. https://www.mentor.com/products/fv/questa/.

[40] Burger D, Austin TM. The SimpleScalar tool set, version 2.0. ACM SIGARCH computer architecture news. 1997;25(3):13–25.

[41] Techtitude. SPEC95 benchmarks (little endian); 2008. [Design by Free CSS Templates]. http://techtitude.blogspot.com/2011/12/how-to-execute-spec95-benchmarks-in.html.

[42] ASE and Synthesis on vLabs;. https://wiki.intel-research.net/FPGA.html.

[43] tmux shortcuts & cheatsheet;. https://gist.github.com/MohamedAlaa/2961058.

Appendix A

ASE Simulation on vLabs

```
# Configure a build environment
$ source /export/fpga/bin/setup-fpga-env fpga-bdx-opae
# Run ASE in the vLab batch queue
$ qsub-sim
# Navigate to the hw directory
$ cd <path to project directory>/hw
# Construct a simulation build directory
$ rm -rf build_sim/
$ afu_sim_setup -s rtl/sources.txt build_sim
# Split the screen
$ tmux
$ Ctrl+b%
# Switch between screens
$ Ctrl+bo
# Compile and run the RTL simulator
$ cd build_sim
$ make
$ make sim
# Switch to the SW pane and go to the sw dir
$ Ctrl+bo
$ cd ../sw
# Copy the export ASE_WORKDIR=<path> from the RTL simulator pane and in-
voke it here
$ export ASE_WORKDIR = <path to project directory>/hw/build_sim/work
# make and run
$ make
$ with_ase ./<sw_api_name> <api_parameters>
```

Appendix B

RTL Synthesis and FPGA Execution on vLabs

```
# Configure a build environment
$ source /export/fpga/bin/setup-fpga-env fpga-bdx-opae
# Navigate to the hw directory
$ cd <path to project directory>/hw
# Construct a synthesis build directory
$ rm -rf build_fpga
$ afu_synth_setup -s rtl/sources.txt build_fpga
$ cd build_fpga
# Run Quartus in the vLab batch queue
$ qsub-synth
# Monitor the build (the file is created after the job starts)
$ tail -f build.log
```

```
# When synthesis finishes, re-configure a build environment
$ source /export/fpga/bin/setup-fpga-env fpga-bdx-opae
# Open a shell on an FPGA system of the configured class
$ qsub-fpga
# export the PBS_O_WORKDIR environment variable
$ export PBS_O_WORKDIR = <path to project directory>/hw/build_fpga
$ cd $PBS_O_WORKDIR
# Load the green bitstream onto the FPGA
$ fpgaconf <green_bitstream_name>.gbs
# Compile the matching software
$ cd ../../sw
$ make
# Run the SW API
$ ./<sw_api_name> <api_parameters>
```

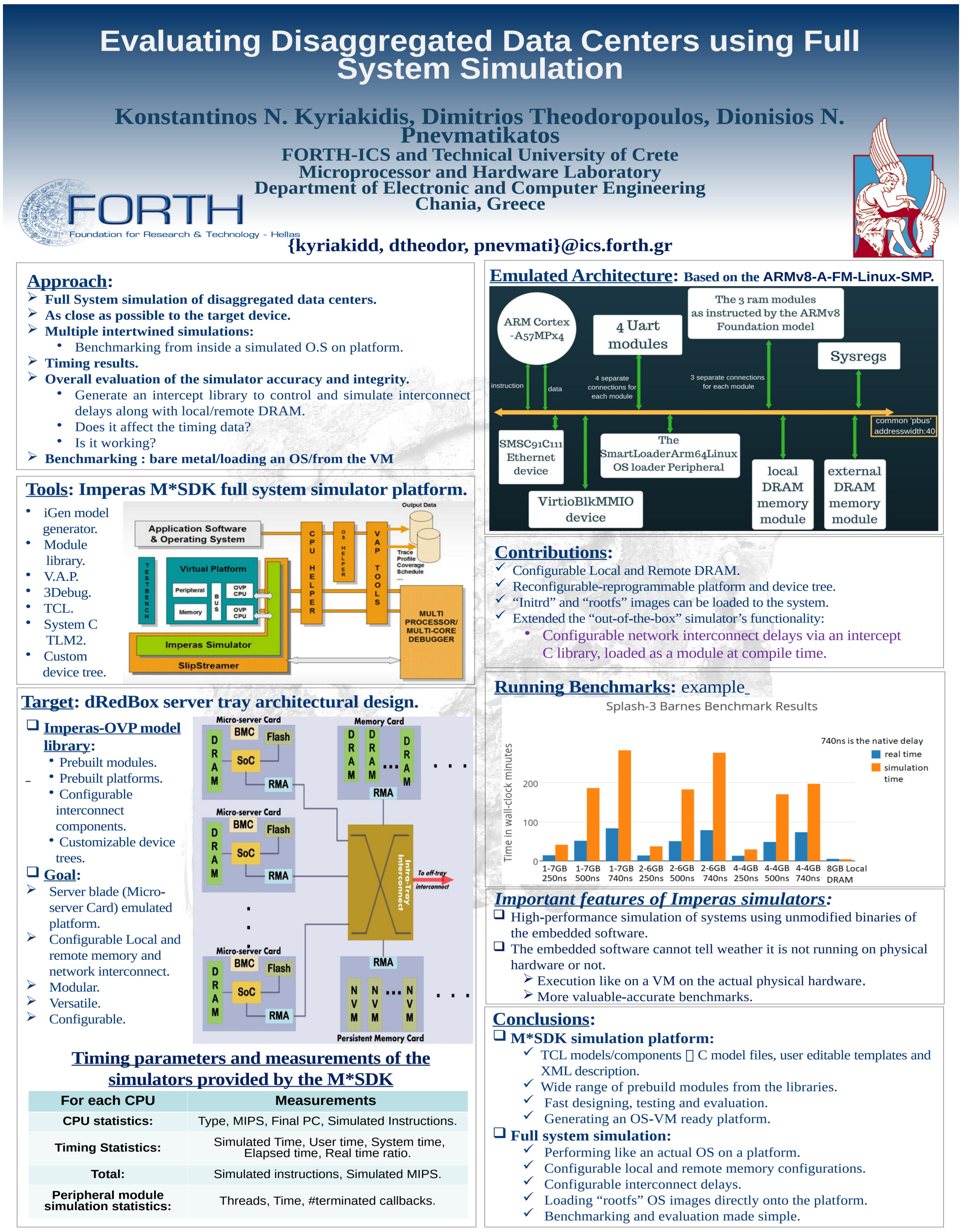
Appendix C

Glossary

- **HARP** : Intel's Hardware Accelerator Research Program.
- **Xeon SP** : Intel's XEON CPU that has an integrated Arria 10 GX 1150 FPGA on the same dye.
- **Hybrid Simulator** : As regards to this dissertation, it is a simulator that runs both on SW and on the FPGA.
- **AFU** : Accelerator Functional Unit.
- **ASE** : AFU Simulation Environment.
- **OPAE** : Open Programmable Acceleration Engine.
- **BBB** : Basic Building Blocks.
- **FIU** : FPGA Interface Unit.
- **UPI** : Ultra-Path Interconnect.
- **QPI** : Quick-Path Interconnect.
- **CCI-P** : Core Cache Interface.
- **CC** : Clock Cycle.
- **BPs** : Branch Predictors
- **BP Simulator** : A standalone simulator for 3 highly configurable Branch Predictor Models.
- **BP+Cache Simulator** : A standalone simulator which is the extension of the BP Simulator, that can also simulate multiple Cache Configurations.
- **Full State Simulator (or FSS)** : A standalone simulator which is the extension of the BP+Cache Simulator, that can also export the system's state.
- **MIPS** : Million Instructions Per Second.
- **SW API** : Application Programming Interface that is used to interface between an SW application and the AFU.
- **MMIO** : Memory Mapped I/O.
- **CSR** : Control and Status Registers that facilitate HW-SW communication and coordination.
- **ALMs** : Adaptive Logic Modules, similar to Xilinx's LUTs.
- **LUTs** : Look-Up Tables.
- **HDL** : Hardware Description Language.
- **PRR** : Partially Re-configurable Region on an FPGA
- **FIM** : FPGA Interface Manager.
- **ISA** : Instruction Set Architecture.

Appendix D

Related Poster



Evaluating Disaggregated Data Centers using Full System Simulation

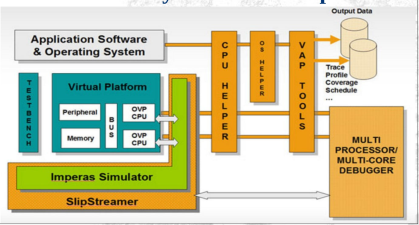
Konstantinos N. Kyriakidis, Dimitrios Theodoropoulos, Dionisios N. Pnevmatikatos
FORTH-ICS and Technical University of Crete
Microprocessor and Hardware Laboratory
Department of Electronic and Computer Engineering
Chania, Greece
FORTH Foundation for Research & Technology - Hellas
{kyriakidd, dtheodor, pnevmati}@ics.forth.gr

Approach:

- **Full System simulation of disaggregated data centers.**
- **As close as possible to the target device.**
- **Multiple intertwined simulations:**
  - Benchmarking from inside a simulated O.S on platform.
- **Timing results.**
- **Overall evaluation of the simulator accuracy and integrity.**
  - Generate an intercept library to control and simulate interconnect delays along with local/remote DRAM.
  - Does it affect the timing data?
  - Is it working?
- **Benchmarking : bare metal/loading an OS/from the VM**

Tools: Imperas M*SDK full system simulator platform.

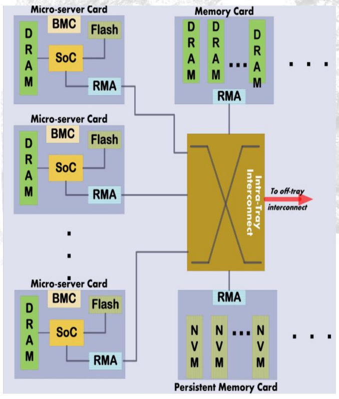
- iGen model generator.
- Module library.
- V.A.P.
- 3Debug.
- TCL.
- System C TLM2.
- Custom device tree.



Target: dRedBox server tray architectural design.

❑ **Imperas-OVP model library:**
- Prebuilt modules.
- Prebuilt platforms.
- Configurable interconnect components.
- Customizable device trees.

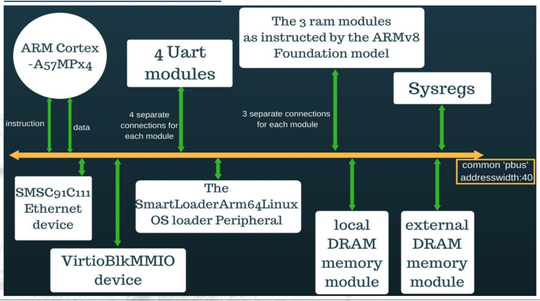
❑ **Goal:**
- Server blade (Micro-server Card) emulated platform.
- Configurable Local and remote memory and network interconnect.
- Modular.
- Versatile.
- Configurable.



Timing parameters and measurements of the simulators provided by the M*SDK

For each CPU	Measurements
CPU statistics:	Type, MIPS, Final PC, Simulated Instructions.
Timing Statistics:	Simulated Time, User time, System time, Elapsed time, Real time ratio.
Total:	Simulated instructions, Simulated MIPS.
Peripheral module simulation statistics:	Threads, Time, #terminated callbacks.

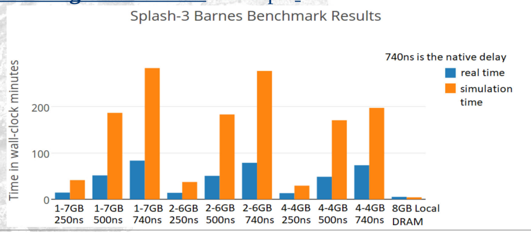
Emulated Architecture: Based on the ARMv8-A-FM-Linux-SMP.


ARM Cortex -A57MPx4
4 Uart modules
The 3 ram modules as instructed by the ARMv8 Foundation model
Sysregs
instruction
data
4 separate connections for each module
3 separate connections for each module
common 'pbus' addresswidth:40
SMSC91C111 Ethernet device
The SmartLoaderArm64Linux OS loader Peripheral
VirtioBlkMMIO device
local DRAM memory module
external DRAM memory module

Contributions:

- ✓ Configurable Local and Remote DRAM.
- ✓ Reconfigurable-reprogrammable platform and device tree.
- ✓ “Initrd” and “rootfs” images can be loaded to the system.
- ✓ Extended the “out-of-the-box” simulator’s functionality:
  - Configurable network interconnect delays via an intercept C library, loaded as a module at compile time.

Running Benchmarks: example_



Important features of Imperas simulators:

❑ High-performance simulation of systems using unmodified binaries of the embedded software.
❑ The embedded software cannot tell weather it is not running on physical hardware or not.
- Execution like on a VM on the actual physical hardware.
- More valuable-accurate benchmarks.

Conclusions:

❑ **M*SDK simulation platform:**
- ✓ TCL models/components C model files, user editable templates and XML description.
- ✓ Wide range of prebuild modules from the libraries.
- ✓ Fast designing, testing and evaluation.
- ✓ Generating an OS-VM ready platform.

❑ **Full system simulation:**
- ✓ Performing like an actual OS on a platform.
- ✓ Configurable local and remote memory configurations.
- ✓ Configurable interconnect delays.
- ✓ Loading “rootfs” OS images directly onto the platform.
- ✓ Benchmarking and evaluation made simple.