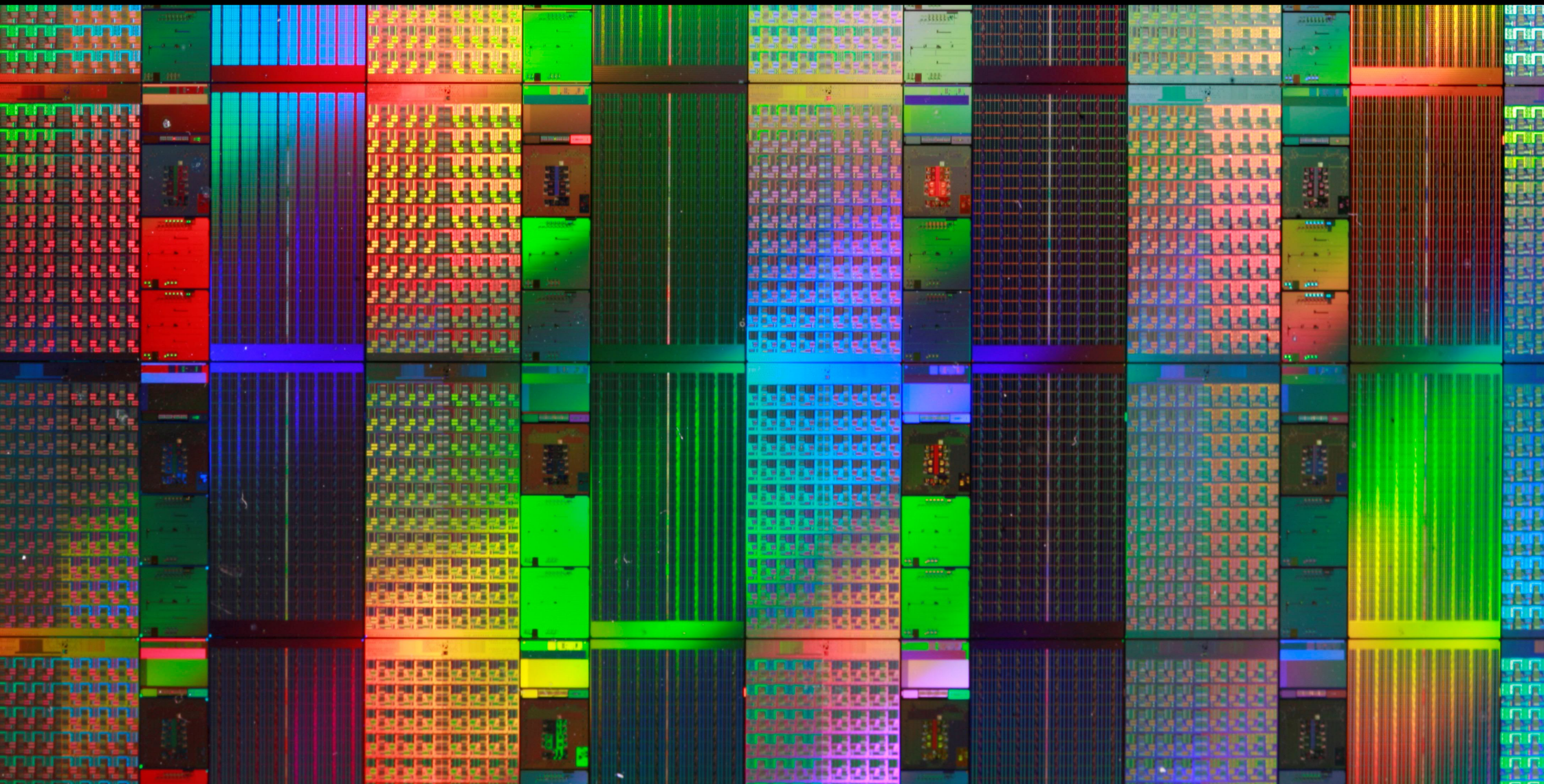
Memory optimizations on the sequential hardware-in-the-loop simulator

Mark Westmijze



Memory optimizations on the sequential hardware-in-the-loop simulator

Master's Thesis
by

Mark Westmijze

Committee:
dr. ir. P.T. Wolkotte
dr. ir. A.B.J. Kokkeler
ir. E. Molenkamp

University of Twente, Enschede, The Netherlands
23rd March 2009

Abstract

Simulating hardware designs with the assistance of a Field-programmable Gate Array (FPGA) can greatly increase the simulation speed. Especially since new hardware designs often encompass a complete System-on-Chip (SoC). Due to the limited resources of a single FPGA these designs may be too large to instantiate them into an FPGA. Wolkotte et al, presented a simulation approach that can simulate those designs [1]. The approach uses time multiplexing to simulate only a small part of the hardware designs in a single clock cycle. This technique works well with hardware designs that contain a lot of nearly identical components, e.g. a Multi Processor System-on-Chip (MPSoC). Such a MPSoC may consist of a 2D-mesh Network-on-chip (NoC). Each router in the NoC could be connected to a small processing element, i.e. the Montium tile processor [2]. One of the transformations that is performed for time multiplexing the simulation is state extraction. The current approach by Rutgers [3, 4] is only able to extract flip-flops from the hardware designs. This thesis introduces some new algorithms that extract large memories. These algorithms make it possible to also simulate the network with the processing element attached. This was not possible in the approach of Rutgers due to the bandwidth limitations within an FPGA.

Contents

List of Acronyms

ASIC	Application-Specific Integrated Circuit
CE	Clock Enable
CLB	Common Logic Block
CPU	Central Processing Unit
DCM	Digital Clock Manager
EDIF	Electronic Design Interchange Format
FF	Flip-flop
FPGA	Field-programmable Gate Array
GPU	Graphical Processing Unit
HC	Hyper Cell
HCI	Hyper Cell Input
HCO	Hyper Cell Output
HILS	Hardware in the Loop Simulator
HDL	Hardware Description Language
IAR	Input Address Read
IO	Input Output
KB	Kilo (2^{10}) Byte
Kb	Kilo (2^{10}) bit
LUT	Look-up-table
MPRAM	Multiple Port RAM
MPSoC	Multi Processor System-on-Chip
NA	Not Available
NoC	Network-on-chip
NPRAM	n-port RAM
OA	Output Address
PnR	Place and Route
RAM	Random Access Memory
RB	Register Bank
RTL	Register Transfer Level
SE	State Element
SoC	System-on-Chip

SOP	Sum of Products
VLIW	Very Long Instruction Word
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

Only 2300 transistors were needed to construct the first commercially available microprocessor. They were used to build the Intel 4004, the first Central Processing Unit (CPU), built by the Intel Corporation in 1971 [5].

Several years earlier, in 1965, Gorden E. Moore observed, that the number of transistors that could be placed on an integrated circuit, doubled almost every year [6]. For about ten years Moore's law held. After that Moore adjusted it to double every couple of years [7, 8].

The latter trend continued, which means that, since the early days of integrated circuits, the number of transistors on chips surpassed the billion mark in 2008. An example of one of these billion transistor chips is the GT200 Graphical Processing Unit (GPU) from Nvidia [9], featuring 240 cores. Furthermore one of Intel's own processors, the Quad-core Itanium Tukwila, has even broken the two billion transistor mark [10].

The last several years there seems to be an increase in the number of large multi-core chips. Examples of these are: The Cell architecture [11], which features nine cores; An Intel research project named the Tera-scale Computing Research provided an 80 core chip, but never went into commercial production [12, 13].

But not only the large high performance chips went from single to multi-core. For embedded solutions, MPSoC are developed, which use a homogenous or heterogenous architecture of embedded processors, which are connected through a NoCs [13]. An example of such an architecture is the Annabelle chip [14].

Developing these new architectures requires knowledge of the required performance of applications and algorithms on the new architectures. Therefore, the new architectures will have to be simulated. However, these simulations can take a long time [1, 15, 4]. The Hardware in the Loop Simulator (HILS) introduces a simulation technique that uses relatively inexpensive equipment to reduce the simulation time of new hardware architectures, it will be elaborated in section 2.

1.1 Related Work

A hardware design has to be thoroughly simulated before it can be put in production. There are many methods how a hardware design can be simulated. These methods range from behavioral simulations to timing-accurate post synthesis simulations. They will be discussed starting at the highest level of abstraction, slowly working to a lower level of abstraction.

A method that can simulate designs from the behavioral level to Register Transfer Level (RTL) is SystemC [16]. SystemC consists of a set of class libraries for C++, these libraries can be used to model the hardware design. Hardware designs that are implemented in a Hardware Description Language (HDL) can be simulated in simulators such as ModelSim [17]. When the simulation speed of the software simulators start to become a bottleneck, hardware assisted simulations are necessary. A simple method is to use an FPGA to simulate the hardware [18], but the size FPGA can limit the amount of hardware designs that can be simulated. Several FPGAs together can simulate larger hardware designs. Techniques such as virtual wires [19], can be used to efficiently use those FPGAs.

Several commercial products are also available such as Veloce from Mentor Graphics [20], and the Hardware Embedded Simulation accelerator from Aldec use multiple FPGAs to simulate those designs. But the complete design does not have to be instantiated completely. The design can be programmed partially in an FPGA, and during runtime be reconfigured to simulate the complete design [21]. Cadambi et al. present a method that uses an FPGA to program a Very Long Instruction Word (VLIW) processor, called SimPLE, which is able to efficiently execute parts of a hardware design [22].

Another method, which is more suitable for hardware designs that largely consists of the same components, is proposed by Wolkotte et al [1]. All state elements are removed from the design, which make is possible to time multiplex the simulation of a single clock cycle. This makes is possible to simulate large hardware designs in a single FPGA. A automated flow has been developed by Rutgers [4].

Chapter 2

Hardware in the Loop Simulator

2.1 Overview

This section gives an overview of the simulator design. Detailed information can be found in [1, 3, 4]

The simulation speed of a hardware design can be increased by using one or more FPGAs to execute the simulation [23]. This is because an FPGA can be tailored to the design, such that each clock cycle of the FPGA corresponds to a clock cycle in the simulation. A software simulation does need many clock cycles on the CPU to simulate a single clock cycle of the simulation [4]. FPGAs use reconfigurable logic to instantiate the hardware design, but the size of the designs that can be instantiated within a single FPGA is limited. A simple, yet expensive, solution would be to use multiple FPGAs to instantiate the complete hardware design. But for very large hardware designs the required number of FPGAs might be too high or the connections between the FPGAs might result in an IO bottleneck. For both cases another solution is necessary.

A hardware design consists of several instances of *components*, which will be called *entities*. For example in a Quad-core microprocessor there are four instances of a processor core, which is the component. The instances themselves represent the entities.

As mentioned before, many of the new and large hardware designs run multiple entities of the same component in parallel. Instead of simulating all those entities in parallel, it is possible to evaluate them sequentially for simulation purposes. Moreover, because all entities from the same component can now run on a single instance of that component, less hardware is necessary to simulate this transformed design. In which case it fits in a single FPGA. The instance of the component that will run the entities from the hardware design is called the *hyper cell*. See [4] on how these hyper cells are generated.

Figure 2.1 depicts the current simulator. The basics of this simulator are elaborated in the following sections.

2.1.1 Multiplexing Internal State

Essentially, all instantiated entities of a specific component in the parallel architecture are run on one instance of that component. However, all these entities have an internal state in the form of memory elements. It is therefore not possible to use an unmodified version of the component to simulate all the entities, because only a single instance of such an component cannot be used to simulate more of them sequentially.

All state elements in the hardware design are replaced with some logic, which emulates the behavior of the extracted state elements. The state itself is then stored outside the transformed component, but through the replacement logic it is still available to the component.

The *internal state* of each entity is represented by an entity's *state vector*. The state vector for each entity is stored in a memory designated *state storage*. The component that replaces all the entities is called the *hyper cell*.

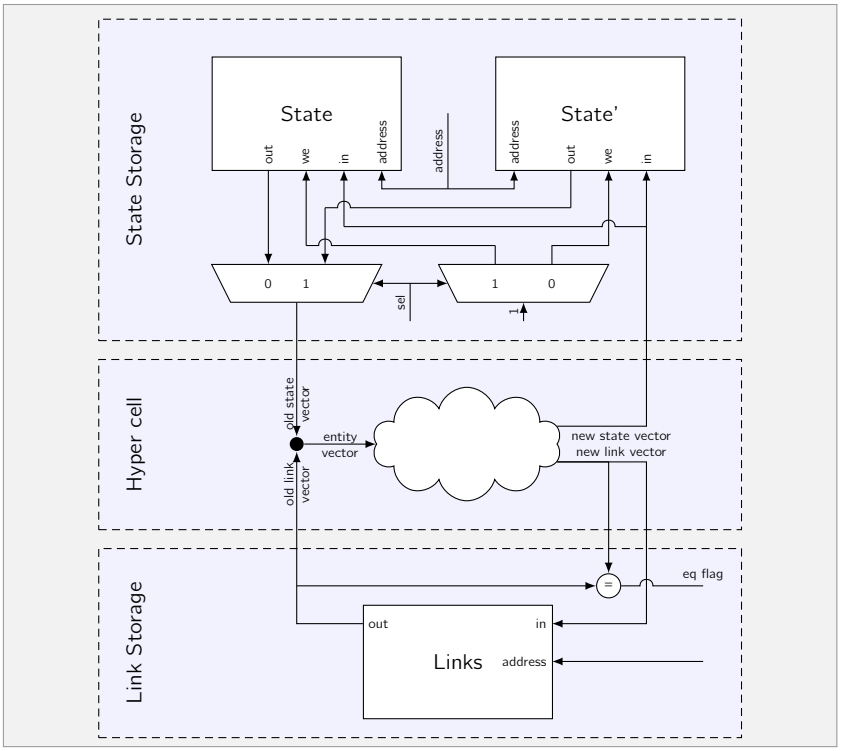


Figure 2.1: State storage configuration

2.1.2 Port Connections

The new state vector, and the output of an entity are also influenced by its input ports. The input ports are connected to the output ports of other entities. Therefore, it is necessary to correctly supply each entity with the output of the connected entities in the parallel architecture. The output of each entity is represented by a *link vector* and is stored in a memory designated *link storage*.

The link vector and state vector together represent the *entity vector*.

2.1.3 Evaluating all entities

A clock cycle in the original hardware design is called a *system clock cycle*. A system clock cycle consists of the evaluation of all separate entities, these evaluations are called *delta cycles*. In each delta cycle the input ports of an entity are supplied by information from connected link vectors. However, before any of the entities are evaluated the link vectors are not yet known. Only after a delta cycle the link vectors for that entity are known. The connections between the entities there may have circular dependencies, when such dependencies are present it is not possible to evaluate an entity with a correct link vector. In this case one of the entities within the dependency cycle has to be evaluated, once its dependencies are evaluated it has to be evaluated again. When an entity is evaluated again, it is possible that its link vectors have changed. In that case the entities that are connected to those link vectors also have to be evaluated again. As long as there are link vectors that have been changed due to an extra evaluation this process continues. However, if there are no combinatorial loops within the hardware, it can be proofed that eventually all link vectors can be derived correctly (See appendix A for the formal proof). When all the link

vectors on which an entity depends are stable, a complete system clock cycle has been evaluated.

2.1.4 State Storage

Because an entity may be evaluated multiple times, it is necessary to store the old state vector for each entity until the end of a complete clock cycle. The new state vectors are also stored during the system clock cycle. Therefore, the state storage stores both the old, and new state for the entire system in separate memories. At the end of a system clock cycle the roles of these memories switch, because the old state in the next system clock cycle was the new state in the current. Hence these memories display ping-pong behavior [3]. This behavior is implemented in figure 2.1 by the muxes in the state storage.

2.2 Current Status

Wolkotte et al, demonstrate that it is feasible to use an FPGA to do fast simulations of large parallel designs [1, 15]. Rutgers made an effort to begin the automation of this simulation flow [3]. This tool was the foundation of the 'Sequential hardware in the loop simulator'. Currently, only a limited number of features are implemented in this simulator.

First, only some memory components are extracted. These are mainly flip-flops, and latches. Hence larger memories such as Random Access Memory (RAM) are not yet supported.

Second, the automated simulator has mainly been simulated, and not run on the actual FPGA itself.

2.3 Research Definition

In order to define the scope of this thesis the actual problem needs to be defined properly. The following sections define some of the current problems that need to be addressed in order to implement a more efficient simulator.

2.3.1 State Access

As explained in section 2.1.1, the state vector represents the state of a specific entity that is simulated. Hence the size of this vector is directly related to the amount of memory present in such an entity. In a naïve solution this complete state vector has to be supplied to the hyper cell of the entity, when it is evaluated. This leads to some severe problems, namely:

Bandwidth The complete state of an entity is represented by the state vector. The problem occurs, when we do have an entity with a large state vector, because for each bit in this state vector a dedicated input line is needed. Before an entity can be evaluated on a hyper cell, the state vector has to be loaded from state storage as fast as possible. Ideally, in one clock cycle, otherwise the pipeline of the simulator will stall, which results in performance penalties. The width of the state vector depends on the hardware design being simulated. But it can range from a few thousand in the case of a small hardware design such as a NoC-router [24, 25] to a few hundred thousand in case of a larger design, such as the Montium tile processor [2].

The only way an FPGA can directly read a large amount of memory in a single clock cycle is when the data is stored on the chip itself. For this there are two techniques.

The first technique is to use the storage capacity of the basic components of a FPGA, the Look-up-table (LUT). In a typical FPGA these LUTs can store 16 bits, depending on the number of input ports of the LUT. These LUTs are glued together by multiplexing logic to create a structure, which behaves as a memory. The main disadvantage of this technique is that it consumes a large amount of basic components. In Xilinx FPGAs this ram structure is called *distributed RAM*. The FPGA used within this thesis supports distributed RAM upto 1056 Kb.

The second technique uses dedicated memory components in an FPGA. These memory components do have a maximum input- and data port width. However, it is possible to use several in parallel to increase the amount of data, which can be accessed or stored in a clock cycle. A disadvantage is that typically these memory components do have a synchronous read port, whereas distributed ram does have an asynchronous read port, which imposes a overhead for the evaluation when a hyper cell has asynchronous memories. However, since this technique uses dedicated memory components, it doesn't consume any other basic component. In Xilinx FPGAs these memory components are called *Block RAM*. Each Block RAM stores 2304 bytes of information, and the FPGA that is used as target within this thesis houses 288 Block RAMs, which results in a total storage capacity of 648 KB. This limited amount of memory can lead to another problem. For a hardware design that contains a lot of memory it is possible that the amount of memory in an FPGA is not large enough to accommodate the state and link storage. In that case the state, and link storage has to be (partially) offloaded to memory outside the FPGA. This creates another set of problems.

State partitioning When both the state, and link storage are placed in external memory, because it did not fit in the FPGA, the entity vector is copied to the FPGA before evaluating the entity. The bandwidth available between the FPGA, and external memory is limited. Therefore, if the state vector becomes too large, the simulator will stall until the complete entity vector is available. This will reduce the speed of the simulator.

However, in most hardware designs only a small portion of the current state vector changes or influences the new entity vector. The current state vector will still represent the complete current state of an entity, and the reduced current state vector will be a subset of this vector, which represents the part of the state, which influences the new entity vector. In the current naïve implementation this distinction is not made, and hence the complete state vector is loaded. The new state vector can be reduced in the same manner, because not all bits in the new state vector will change. Only the bits that might change, which depends on the current entity vector, do have to be updated.

Hence the first problem is:

How can we reduce the size of the old and new state vector?

2.3.2 State Storage

All the state vectors from all the entities represent the *total system state*. When an entity is evaluated the state vector for that entity does have to be fetched from state storage. The width of this state vector depends on the hardware

design, but for performance reasons should be fetched in as few clock cycles as possible and ideally with a fixed latency in order to reduce the complexity of the simulator pipeline.

When using FPGAs, several storage containers are available, these include: Distributed ram, Block RAM and external memory. The state storage can be divided and spread over these containers. Due to time constraints for this thesis the focus will be how to use the Block RAMs as efficiently as possible. Because how the state storage is mapped onto the Block RAMs of the FPGA determines how much resources are needed and also influences clock frequency.

Hence the second problem is:

How can we efficiently store the state storage in the FPGA?

2.4 Outline Thesis

Chapter 3 describes the solution for the state vector reduction. It covers the type of memories that can be extracted, and how they can be extracted. It concludes with some results on the vector reduction.

How the extracted state can be stored as efficiently as possible is elaborated in chapter 4. First, a mathematical model is introduced, which can find the optimal mapping with respect to the number of memories necessary for the mapping. Second, some heuristics are introduced, which can also be used to find mappings, but may not find the optimal mapping. The chapter concludes with some results on the state storage.

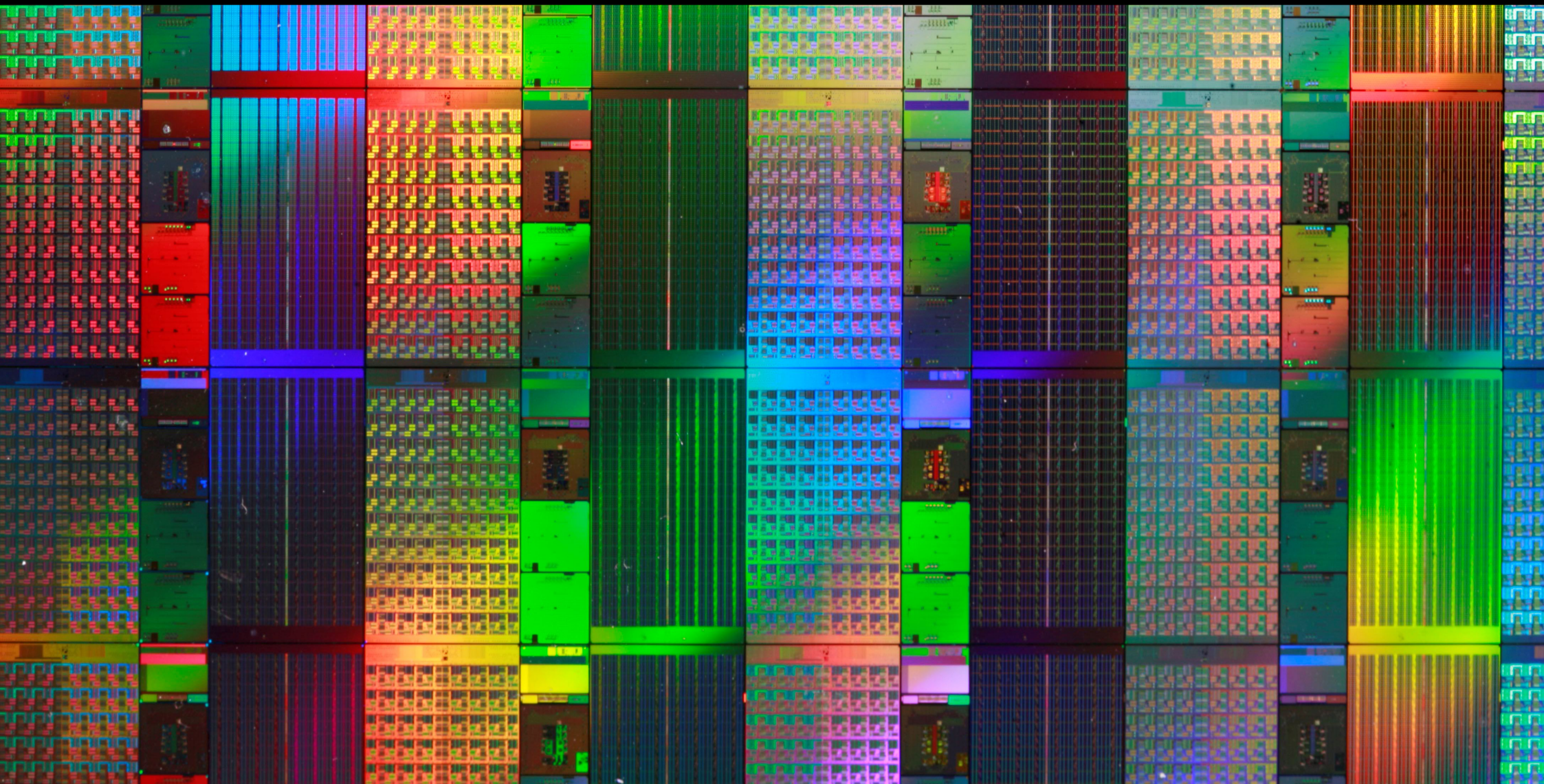
When both the state vectors are reduced, and an efficient way of storing all the state vectors is known, the storage pipeline which is responsible for loading, and saving the state vector can be implemented. This is elaborated in chapter 5.

A small case study is performed on a NoC-router, the results can be found in chapter 6.

Finally, some the conclusions are presented in chapter 7, some further topics of research are also elaborated here.

Chapter 3

State Access



Abstract

In this chapter two techniques for reducing the entity vectors are presented, in order to efficiently simulate large hardware designs. The first technique detects, and extracts large memories, which are represented by clocked_ram primitives in the hardware graph. The second technique detects register banks. Each register bank is removed from the hardware graph, subsequently replaced by a clocked_ram primitive, and some supporting logic. This replacement behaves exactly as the original register bank, but because the replacement uses the clock_ram primitive to store the state it can be extracted by the first technique.

Outline

In the approach of Rutgers [3, 4] the tool uses the synthesized netlist as input, some alternatives to this format are presented, and elaborated in section 3.1. Subsequently, the hardware design from this level is converted into a graph representation, which will be used to perform the analysis, and state extraction on, that is elaborated in section 3.2. Before the actual analysis, a short introduction on which types of memories are present in the hardware can be found in section 3.3. The two analysis and extraction techniques will be presented respectively in section 3.4 and 3.5.

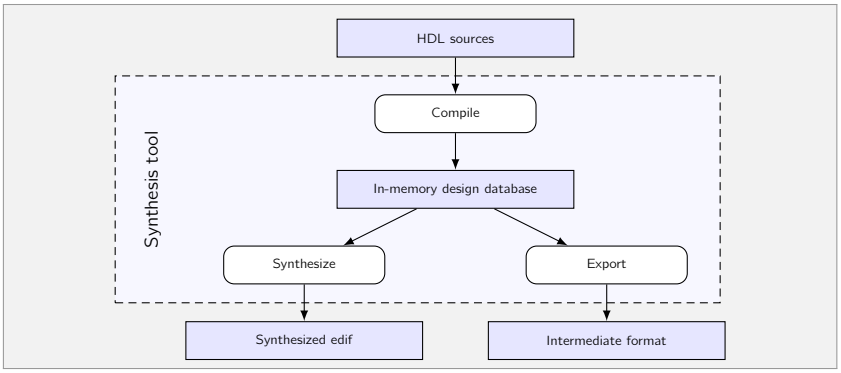


Figure 3.1: Input levels

Currently, the extracted state is exported as a bit vector, which implies that during the evaluation of an entity the complete vector has to be loaded and saved. At this moment the state vectors are stored in Block RAMs on the FPGA itself. For now an FPGA, the Xilinx `XC4VLX160`, with 288 Block RAMs, is used as target platform. Each Block RAM is a dual port RAM that both support read- and write actions, and has a maximum data port size of 36 bits, and is capable of storing 2304 bytes. Since the new state vector has to be saved in the same clock cycle as the old state vector has to be loaded, only half of the total number of ports are available for either action. This results in a maximum bandwidth of 10368 bits per clock cycle per action, when all the Block RAMs are used for state storage. However, since the Block RAMs are also used for the link memories and other parts of the simulator, the actual bandwidth is smaller. For a small hardware design the required bandwidth suffices, but when the hardware design requires more bandwidth than available, the pipeline has to stall, and this will reduce performance of the simulator.

3.1 Analysis Level

All hardware designs, that eventually will result in an ASIC, will have to be processed in a few steps. The hardware design is initially represented in a HDL, such as VHDL or Verilog. In the first phase of the synthesis flow a HDL design is read, and compiled into technology independent components. The second phase consists of mapping these technology independent components into technology dependent components. The result of the synthesis is a netlist that can be saved in the Electronic Design Interchange Format (EDIF), a netlist is a file that describes all components of a hardware design, and how they are connected [26]. Compilation, and synthesis is usually done within a single tool, but some tools can also export the netlist between these two phases [27, 28]. A graphical overview of the levels, and how they are related is depicted in figure 3.1. These different levels, of which one will be chosen to be used within this thesis, are elaborated in the following paragraphs.

Synthesized EDIF The EDIF format is a standard format for the interchange of electronic designs between different tools [29]. Within this project there are two synthesis tools available, Synopsis [27] and Precision [28], that deliver netlists. But since these use the same technology library, both tools are elaborated as one option. The major disadvantage is that the used technology library does not support RAM components with asynchronous read ports. Hence a RAM, with asynchronous read, is implemented as distributed RAM, and these structures

are more complex to extract. An advantage is that, when the synthesis target is a Xilinx FPGA, the components that are used are thoroughly documented by Xilinx.

Precision's intermediate format As mentioned before most synthesis tools do have two distinct phases. Between these two phases the hardware is described by a netlist that uses technology independent components. An example of such a component in the Precision tool is the clocked_ram component. The clocked_ram component is used as memory component for all types of memory configurations. The component itself functions as a memory with synchronous write ports, and asynchronous read ports.

The major advantage is that, because this component is used as basis for all large memories, when this component is extracted, all large memories are extracted. The major disadvantage is that Precision is not able to import the exported files directly before the second phase. However, these shortcomings have been circumvented by exporting the netlist to VHDL, which made it possible to compile. Another disadvantage is that the technology independent components are not documented. Since these components are relatively simple (e.g. asynchronous memory, muxes, selects, incrementors, multipliers, etc), it is not infeasible to determine the functionality of these components, after they can be simulated correctly.

Synopsys intermediate format Synopsys also uses an intermediate format before the actual technology mapping. While the documentation suggests that it can infer asynchronous memories, this resulted in several internal errors within the compiler of Synopsys, and was therefore not usable. Because Synopsys did not infer these asynchronous memories correct, no effort was put into determining the source of these internal errors.

VHDL All the techniques mentioned above extract the ram after phases in the synthesis flow. Because all these synthesis tools have the ability to detect memory components, it should also be possible to write a VHDL analyzer. This VHDL analyzer should then be able to detect memory components, and extract them at the VHDL level. For the analysis there is a parser available [30]. However, even using this parser the analysis of VHDL proved to be too complex, and is not further examined within this project. Another disadvantage of VHDL is that it does not cover all hardware designs, i.e. designs written in other HDLs cannot be analyzed.

Evaluation Precision's intermediate format was chosen for the level to perform the analysis, and transformations. Detailed information on which file formats are used for exporting, and importing this intermediate format can be found in appendix B.1. The most important advantage of this level is that it is technology independent.

3.2 Netlist Representation

The netlist describes the functionality of the design by an interconnection of primitive, and operator components. In the intermediate stage there are two libraries, PRIMITIVES, and OPERATORS. These two libraries are used to determine the behavior of the component (See page 18).

We represent each primitive by a small graph in which the primitives function and each port is represented by a vertex. Each vertex of a primitive has a unique label. The set P contains all primitives of the technology libraries.

Graphs representing such instances, are defined as:

Let $p \in P$ be a component with input ports $I[p]$ and output ports $O[p]$. A *primitive graph* is defined as $g_p = (V_p, E_p, L_p, B_p)$, where g_p is a directed graph that represents p and

- $V_p = I[p] \cup O[p] \cup \{p\}$ is the set of vertices (also called *nodes*) in the graph, where p is a vertex that represents the primitive p;
- $E_p = \{i \in I[p] \mid \langle i, p \rangle\} \cup \{o \in O[p] \mid \langle p, o \rangle\}$ is the set of edges in the graph;
- $L_p(v \in V_p)$ is a function that assigns a label to every vertex, such that $L_p(p)$ is a label based on the primitive of p and $L_p(v \in I[p] \cup O[p])$ is based on both the primitive of p, and the label of the corresponding port.
- $B_p(v \in V_p)$ is a function that assigns a library to every vertex.

The netlist is the hierarchical description of cells interconnected by wires. The instance of a component in a netlist is an example of a cell. Larger cells consist of multiple instantiated primitives and other cells. Each cell in the netlist is represented by a netlist graph in which vertices are uniquely described by their label and their hierarchical position (denoted with a sequence of numbers) in the corresponding netlist.

Thus, the graph of an instance of a primitive extends its primitives graph with such a hierarchy annotation on every vertex. The first number of this annotation corresponds to the cell number on the top level of the netlist, the second corresponds to the cell number within the top level cell, etc.

This netlist can be described as a graph:

Let c be a cell, built of instances of components P_c and other cells C_c, with input ports $I[c]$ and output ports $O[c]$. A *netlist graph* is defined as $g_c = (V_c, E_c, L_c, H_c)$, where g_c is a directed graph that represents c and

- $V_c = I[c] \cup O[c] \cup \bigcup_{i \in P_c \cup C_c} \{\langle L_i(v), i : H_i(v) \rangle \mid v \in V_i\}$ is the set of vertices in the graph;
- $E_c = W_c \cup \bigcup_{i \in P_c \cup C_c} E_i$ is the set of edges representing the wires in the netlist, where $W_c \subseteq V_c \setminus O_c \times V_c \setminus I_c$;
- $L_c(v \in V_c)$ is a function that assigns a label to every vertex in the graph, such that $L_c(v) = L_p(v)$ when $v \in V_p$ for a given $p \in P_c$, otherwise $L_c(v)$ gives a label based on the port $v \in I[c] \cup O[c]$ it represents.
- $H_c(v \in V_c)$ is a function that maps a vertex onto the hierarchical annotation.

The combination of the $L_c(v)$ and $H_c(v)$ distinguishes every vertex in the graph.

Graphical representation Figure 3.2 shows how a single primitive is depicted graphically. The cell node is represented by the rectangle. The circular nodes represent the input-, and output ports.

3.3 Memory Types

Within Precision's intermediate format there are two distinct ways how memory structures are represented. A memory structure is defined as a set of primitives, which together store an array of values, and using an address-, data-, and write enable port, values can be stored, and retrieved from this structure. The

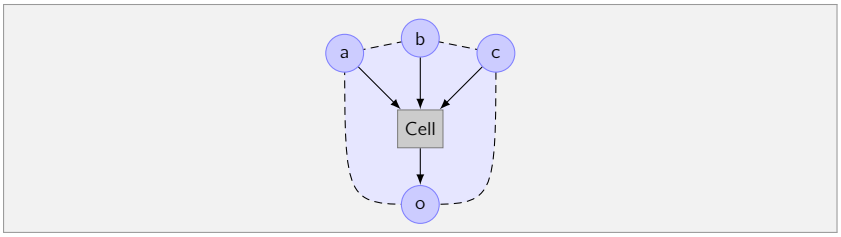


Figure 3.2: Primitive example

behavior of both memory structures is the same, but within the netlist they are represented in completely different ways.

- The first memory type, which is defined, is the RAM structure. It is represented by a single primitive within the netlist. How this structure is detected and extracted can be found in section 3.4.
- The second memory type is the register bank, as the name suggests, it is an array of registers. The registers themselves are an array of flip-flops. This also means that this memory structure is not represented by a single primitive. Each flip-flop is a primitive, and other primitives are necessary to control the write enable signals, and the necessary logic to select values from the flip-flops. Section 3.5 elaborated the analysis, and extraction of register banks.

Before memory structures can be extracted they must be detected. This detection must not only detect primitives that together represent some memory structure, but also how this memory structure behaves, in order to simulate it correctly.

3.4 RAM Structure

In this section the analysis of the RAM structure will be elaborated. First, the behavior of such structures is presented. When the behavior has been analyzed, the structure can be detected in the hardware graph. In the last step the detected structure is replaced in the graph.

3.4.1 Behavior

As mentioned before, a memory element represents an array of values in which each value is separately addressable for read- and write actions. Typically, a memory element supports only synchronous writes. Therefore it will at least have one *clock* port. It is possible that multiple clock domains use the same memory element. In that case multiple clock ports are present. Data can be retrieved and stored using address-, data- and write enable ports. An *address* port is responsible for addressing the location for the read, and or write action. An *input* port is available for writing new data to the memory, but it is only written when the *write enable* port is active. The data that is retrieved is available at the *data* port, or sometimes this port is referred to as the q port, if the *read enable* is active. Ports do not necessarily support both the read- and write action. Input, and data ports may be shared. Additional ports may be present, which enable the memory, set the output port in case of synchronous read, and so on. Furthermore, the behavior of reading and writing on a single location in a single clock cycle can be described by either write-after-read or read-after-write.

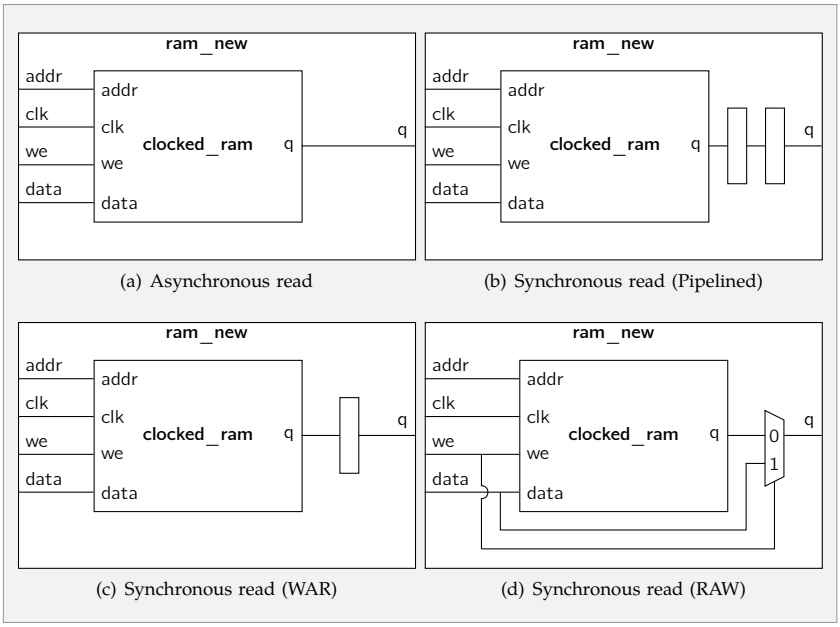


Figure 3.3: Examples of ram operator

Precision's RAM implementation In Precision's intermediate format all RAM structures are represented by the *ram_new* operator. However, this operator is not a black box, it is further specified in the internal library OPERATORS, which is available within the netlist. The main component in this ram_new operator is the *clocked_ram* operator. This operator implements the core function of the RAM structure: Storing the data in a memory structure. Additional components such as flip-flops, and muxes implement other parts of the functionality of the RAM, such as a synchronous read port, read-after-write behavior , write-after-read behavior, etc. See figure 3.3 for several RAM structures in the intermediate format.

Because of all the logic within the ram_new component it is not necessary to extract the complete ram component. It suffices to extract all the state components within it. The only state components in the ram component are flip-flops to store the output of the read ports, and the clocked_ram component to store all the data. How the flip-flops can be extracted can be found in [3].

Details of the clocked_ram operator As mentioned in the previous section the clocked_ram component is a black box. In all configurations it behaves as a synchronous write, and asynchronous read memory component. It is used as the basic component for all memory storage. The behavior of the clocked_ram component depends on a few properties. Most of these properties are also available in the component's identification string. An example of such an identification string is:

```
clocked_ram_16_6_64_F_F_F_F_F_F_F
```

The numbers respectively represent data width, address width, and the total number of locations. After the numbers are eight boolean options, see table 3.1 for the explanation of these options. The boolean values give information on which port this clocked_ram possesses, hence it is also possible to ignore the boolean option, and check the interface of the specific component for which ports it possesses. The only field, which is not available as a boolean, is *ram type*.

clocked_ram_16_6_64_A_B_C_D_E_F_G_H		
Option	**Generic**	**Explanation**
A	dual clocks	There are two clock inputs
B	dual addresses	Read and write addresses are separated
C	n addresses	There are more than two address ports
D	n addresses	There are more than three address ports
E	dual data ports	There are (at least) two data ports
F	dual out ports	There are (at least) two out ports
G	n out ports	There are more than two out ports
H	n out ports	There are more than three out ports

Table 3.1: Explanation of option booleans for clocked ram

Port	**Explanation**
clk	Clock
[clk2]	Second clock domain
we	Write enable for port 1
[we2]	Write enable for port 2
address	Address for port 1
[addr{2, 3, 4}]	Address for ports 2, 3 and 4
q	Data output for port 1
[q{2, 3, 4}]	Data output for port 2, 3 and 4
data	Data input for port 1
[data2]	Data input for port 2

Table 3.2: Clocked ram ports

It describes which of the ports are used for reading and writing, where the latter also can be deduced on the total number of write enable ports. However, also this behavior can be deduced based of the ports of the component. One last behavior to mention is that the clocked_ram component only triggers on the rising edge of the clock, and does not support dual edge triggering.

3.4.2 Detection

Clocked_ram components are represented by a single primitive. The label of the cell node for that primitive starts with 'clocked_ram_'. The detection of clocked_ram components is therefore trivial.

3.4.3 Extraction

For the extraction of the clocked_ram only its input-, and output ports have to be connected to the outside. Replacement logic outside the hyper cell should fetch the required data for the input ports, and save data from the output ports to state storage.

In the intermediate graph format the implementation of this extraction is trivial, and not elaborated here.

Extracted RAM behavior Because the clocked_ram component uses an asynchronous read, the data for that clocked_ram has to be available within the clock cycle that the entity is evaluated. Since the state is stored in Block RAMs that only support synchronous read actions, this is not possible Two techniques to circumvented this problem are multiple evaluations, and pipelined evaluation. In the following two paragraphs these techniques are elaborated.

Multiple evaluations The easiest solution to the asynchronous read problem is to evaluate the hyper cell multiple times, each time the entity is evaluated another 'fraction' has become stable. For instance, if we examine figure 3.4(a), a

chain through two asynchronous memory elements can be detected. After the first evaluation the correct read address for 'ram 1' is available. Hence in the second evaluation the logic between 'ram 1' and 'ram 2', shown in the figure as cloud G, derives the correct address for 'ram 2'. In the third evaluation the correct data is available for the output ports of both memories, and the state has stabilized.

This example shows that the number of evaluations necessary depends on the longest chain of asynchronous elements in the hardware design. The first technique used in the example determines the longest chain of asynchronous elements, and the scheduler has to evaluate the entity that amount of times after the last input change. The disadvantage of this solution is that some of its evaluations may be redundant, because the correct addresses already are available.

Another technique could compare the complete output of the hyper cell, and when the results of two consecutive evaluations are the same, and the input ports were the same, the state has stabilized. A disadvantage of this technique is that it requires one evaluation more in comparison with the first solution in the worst case, but when some of the addresses in the chain are already correct, the best case uses less evaluations than the first technique. Another disadvantage is that the comparison between the output of the current and last state may consume a lot of resources resources.

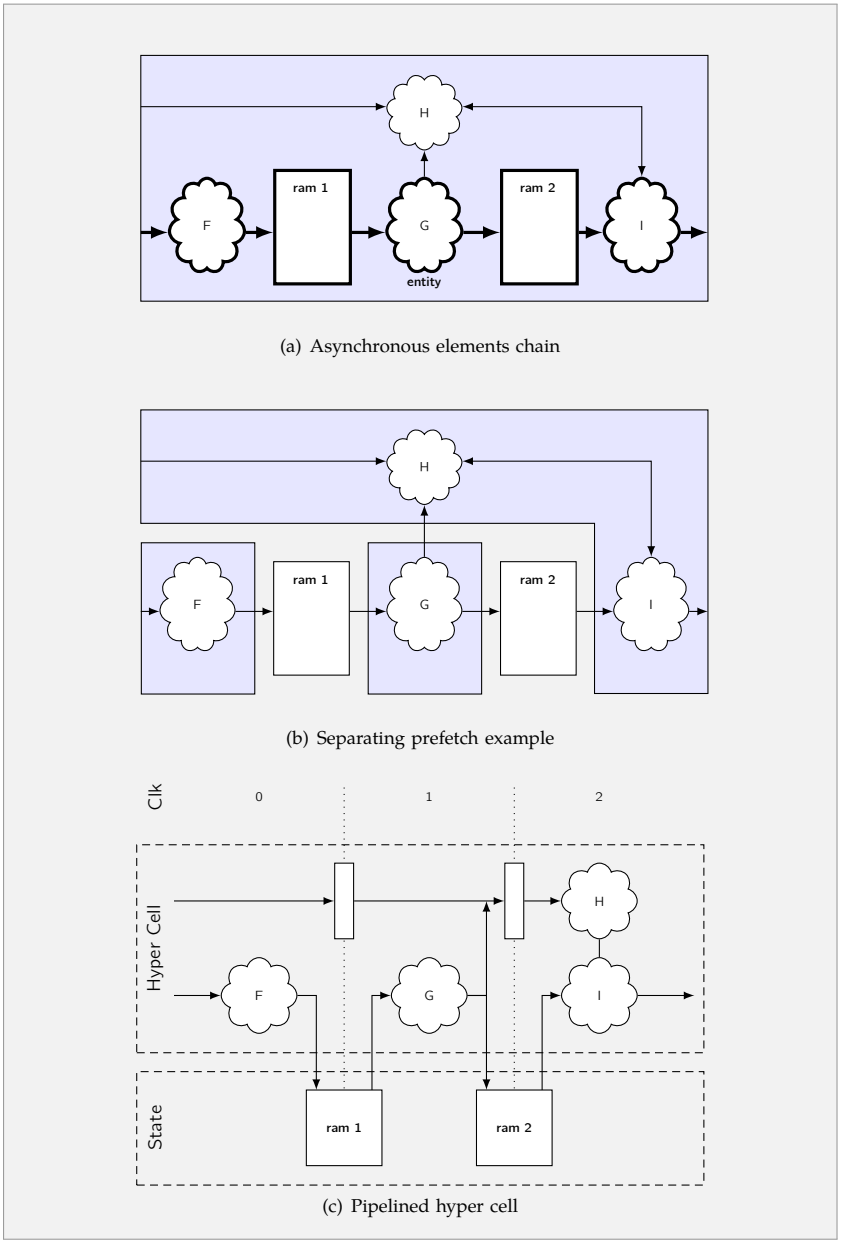
Therefore, a hybrid solution, which does count the evaluations, and also checks if the last two evaluations are the same may results in the least number of evaluations.

Pipelined evaluation Instead of multiple evaluations to fetch the correct data, it is also possible to pipeline the hyper cell, so that a memory only fetches data at the moment that an address port has stabilized. The technique, which counts the number of evaluations that are necessary to stabilize a hyper cell, can be used to determine for each primitive when it has stabilized. All the primitives, which stabilize in the same clock cycle, are grouped together. See figure 3.4(b) for how the hyper cell from figure 3.4(a) would be divided. These groups can be used to implement a pipelined hyper cell. See figure 3.4(c) for the pipeline, which used the groups from figure 3.4(b); The advantage of this technique is that only one evaluation of the hyper cell is necessary. This reduces the bandwidth necessary to evaluate hyper cell. However, it does introduce a latency in the hyper cell, the new state is not available within one clock cycle.

Evaluation Because of the simplicity of the first technique, counting the evaluations of the hyper cell, it has been chosen for implementation. However, implementing the pipelined technique could drastically improve performance, but this is left as future work.

Further optimizations The following paragraphs present some optimizations that could further increase the simulation speed. These optimizations have not been implemented.

Synchronous read optimization Pipelining the hyper cell for asynchronous reads imposes an overhead. We have to implement a larger pipelined datapath, and controlling this datapath makes the design of a scheduler more complex, and possibly slower. Ideally, the number of pipeline stages is minimized. This is possible if an originally replaced ram does have synchronous read ports. In that case the data has to be available before the next evaluation in the next system clock cycle. Furthermore, it means that in case of multiple



(a) Asynchronous elements chain

(b) Separating prefetch example

(c) Pipelined hyper cell

Figure 3.4: Asynchronous elements

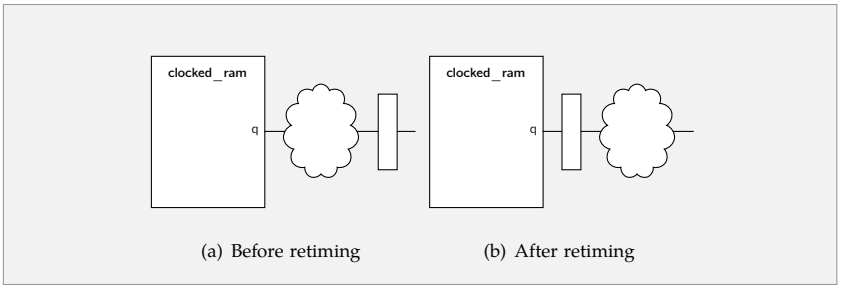


Figure 3.5: Ram retiming

evaluations no data is fetched from state storage. Only at the end of the system clock cycle the data has to be fetched from state storage. As we can see in figure 3.3(b), 3.3(c) and 3.3(d) synchronous reads are implemented by placing a register after the output port of the clocked_ram. Hence if the output of the ram is only connected to flip-flops, the data does not have to be prefetched, because it is not actually used within the same clock cycle. So instead of an extra evaluation or pipeline stage, the address of the read location can be forwarded to the next clock cycle.

Synchronous read optimization through retiming When a ram is originally implemented with asynchronous read ports, the hyper cell has to be pipelined or multiple evaluations are necessary. However, it might be possible to retime memory elements such that the output of the data port is directly connected to a synchronous write port, basically creating a synchronous read. See figure 3.5 for an example.

3.5 Register Bank Structure

Where the clocked_ram component represents a complete memory structure in a single primitive, register banks are made using multiple primitives. To be more specific the data is stored in flip-flops, the clock enables of those flip-flops are driven by addressing logic, and the correct data from the flip-flops is selected by muxing logic. All the primitives, which implement the flip-flops and the logic, which creates the register bank behavior together, represent the register bank. Normally, when memories are described in the HDL, the synthesis tool will correctly identify them as RAM structures. But sometimes they are not identified as memory, because for example clock gating is used, and in this case the memory structure is instantiated as register bank.

See figure 3.6 for a graphical example of how a register could look like. The register bank consists of several primitives, which can be divided into three groups that implement parts of the behavior of the register bank.

The core functionality of the register bank is data storage. This is done by the 'state' group, which consists of a group of flip-flop that are divided into registers. The next group, the 'CE-logic' group, is responsible for which of the flip-flops are enabled for writing, based on the address-, and we port. The last group, the 'read logic' group, selects the flip-flops that are being read by the read address port.

3.5.1 Detection

There are several techniques how a register bank can be detected. In this section several techniques are elaborated, and evaluated.

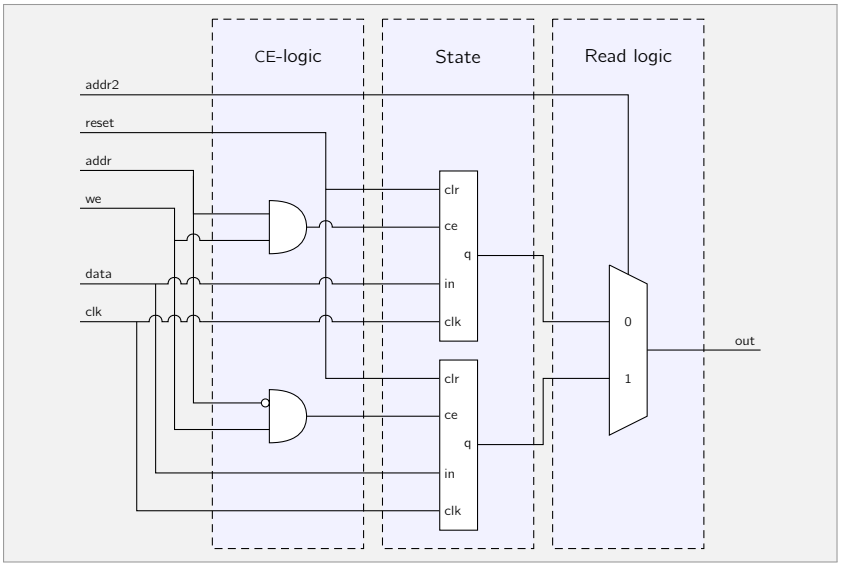


Figure 3.6: Register bank with two locations

Predefined search patterns For each known configuration of primitives that behave as register bank, a search pattern can be defined. A search pattern consist of a group primitives, and how they are connected. An advantage of this technique is that it is simple to implement. However, this also has a downside, because only the predefined search patterns are extracted there can be no guarantee that all register banks are found. Furthermore, the configuration of known register bank configurations will already be enormous, matching all those search patterns on the graph will be time consuming.

Graph matching In essence the internal representation of the hardware is a graph structure. This graph could be converted to a format, which can be imported in a graph matching, and transformation tool, such as Groove [31]. The idea is that the patterns specified in Groove are more general than the predefined patterns from the previous technique, and thus could recognize more register banks, with less predefined patterns. However, a small test with Groove showed that it is not powerful enough to express patterns that describe register bank behavior. Detecting register banks with multiple patterns is still feasible, but is nothing more than another way to express predefined search patterns.

CoSy pattern matching CoSy [32] is a compiler generation suite. Internally it uses a graph like representation based on predefined building blocks. It could be possible to represent the hardware description in this format. Larger patterns, such as muxed/demuxed flip-flops, might be detected and extracted using the included pattern matcher. The major disadvantage is that the pattern matcher also expects predetermined search patters, and thus is yet another method to implement predefined search patterns. Another disadvantage is that the CoSy compiler has a closed source license.

Behavioral search Starting with some basic behavior of register and register banks it should be possible to detect these structures without predefining specific patterns. At the moment, the only primitives which are subject of any analysis are the state elements, all combinatorial logic is seen as one

Table 3.3: Behavior of flip-flop

clk	in	ce	set	clr	old	q / new
↑	0	1	0	0	-	0
↑	1	1	0	0	-	1
↑	-	0	0	0	0	no change
-	-	-	1	0	-	1
-	-	-	0	1	-	0
-	-	-	1	1	-	-

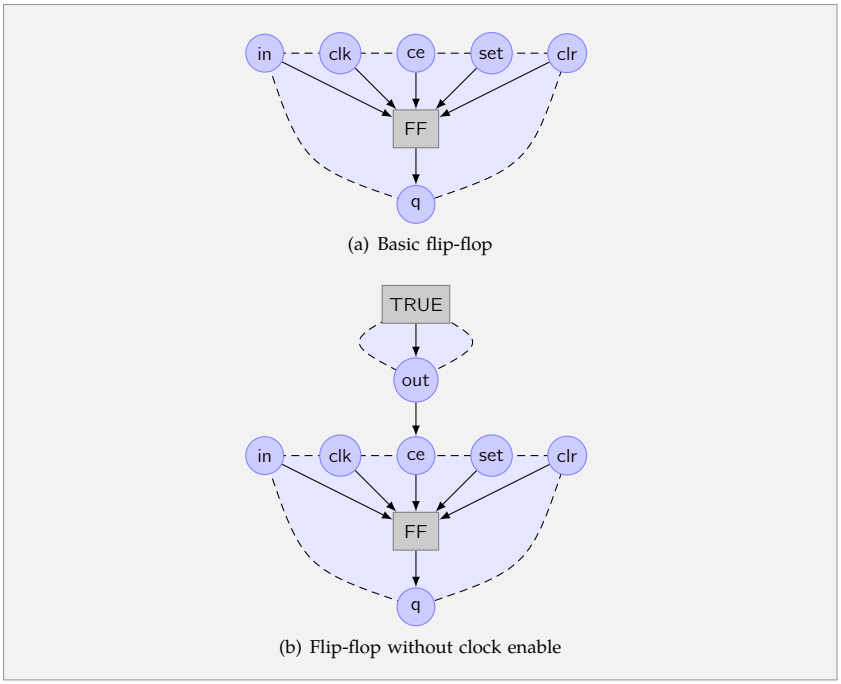
large function, which operates on the entity vector. But this function basically describes when certain flip-flops should be written to, which flip-flops have influence on the output entity vector, and so on. Based on the behavior of this function, register bank behavior might be detected, and this behavior can be described mathematically. This is the major advantage of this technique; no search patterns need be defined, which could result in the detection of register banks, which were not known in advance. However, some small tests quickly showed that this technique uses too much processing time, when it has to take into account all of the combinatorial logic.

Evaluation The graph matching, and behavioral search techniques both have some promising advantages. Both techniques try to detect register banks without defining the exact structure of what they are looking for. At this moment a hybrid solution of these techniques seems likely to give the best result. The small test with graph matching resulted in an easy way to detect groups of flip-flops that might belong to the same register bank, but much more than that was difficult to achieve without predefined patterns. But the combinatorial logic that was connected to these groups of flip-flops was a small subset of the graph, analyzing this subset with the behavioral search might be feasible without requiring much processing time. Therefore, a hybrid approach of graph matching and behavioral search is further elaborated in this section.

Register bank detection steps As mentioned before, the register bank detection will be separated into several steps. A register bank consists of an array of registers, which themselves are arrays of flip-flops. First groups of flip-flops are detected by graph matching, these groups represent the registers in the hardware design. The registers are grouped again using another graph matching rule. These groups represent possible register banks. These groups will than be used to analyze a subset of the graph with a behavioral search to determine if the group of registers really does behave as a register bank. After this analysis can the register banks be removed, and replaced in the graph. These steps will be elaborated in the next paragraphs.

Flip-flop grouping First step is to identify the registers. Figure 3.7(a) shows the graph representation of a flip-flop. This configuration is defined as the basic flip-flop. The behavior of other types of flip-flops can be emulated using this flip-flop and some additional logic. For example, a flip-flop without a clock enable can be seen as a basic flip-flop where the clock enable port is connected to logic '1', as seen in figure 3.7(b) The same applies for flip-flops without a set or reset port. The behavior of the flip-flop can be found in table 3.3.

A register is represented by a group of flip-flops within the graph, which act the same. For example, when a register stores the information from the in port, all flip-flops within the same register must do so, the same holds for the set and reset ports.



(a) Basic flip-flop

(b) Flip-flop without clock enable

Figure 3.7: Flip-flop configurations

This can be described formally. First let S be the set of all possible states of the hardware described by the graph, which is the Cartesian product of the possible states of all state primitives. Let P be the set with all primitives of the graph. Two flip-flops act the same, if for all possible states the input on their control (clk-, ce-, set- and preset) ports is the same. The following function determines what the value is for a specific port on a primitive in a state:

$$V_{i,p,s} = \text{The value for port } i \text{ from primitive } p \text{ in state } s \tag{3.1}$$

Furthermore, we need to express when a flip-flop belongs to a specific register:

$$R_{f,r} = \text{Flip-flop } f \text{ belongs to register } r \tag{3.2}$$

Finally only the flip-flops in a graph need to be checked; Hence the ability to retrieve the type of a primitive is defined via:

$$T_{p,t} = \text{The primitive } p \text{ is of type } t \tag{3.3}$$

Lets first define the formula, which checks that two flip-flops, $f_i and f_j$, act the same way for a specific state s. In the following formula the first two arguments are the two flip-flops followed by a specific state. Basically the formula checks that in a specific state the value of the control ports of the flip-flop have the same values. When they all have the same value, they will act the same.

$$\begin{aligned} F_{f_i,f_j,s} = (&V_{\text{ce},f_i,s} = V_{\text{ce},f_j,s} \\ &\wedge V_{\text{set},f_i,s} = V_{\text{set},f_j,s} \\ &\wedge V_{\text{clr},f_i,s} = V_{\text{clr},f_j,s} \\ &\wedge V_{\text{clk},f_i,s} = V_{\text{clk},f_j,s}) \end{aligned} \tag{3.4}$$

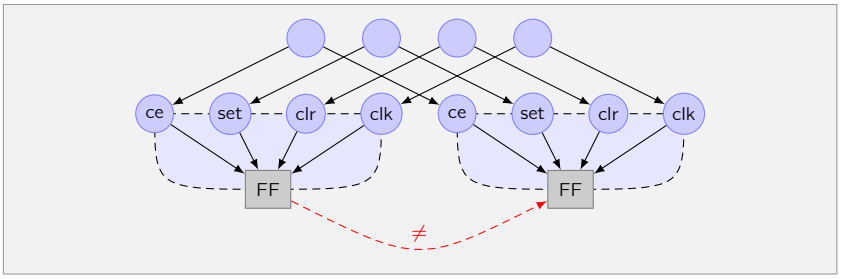


Figure 3.8: Two flip-flops belong to the same register rule

We use this formula, so that it does the check all possible states:

$$G_{f_i,f_j} = \forall s \in S \bullet F_{f_i,f_j,s} \tag{3.5}$$

This will result in groups of registers where flip-flops will all reset or set together. A relaxation of equation 3.4 could also determine that when one flip-flop does a set, and the other a reset it does exhibit the same behavior. The register that will be found will have initiation pattern other than only zeros or ones.

$$\begin{aligned} F'_{f_i,f_j,s} = (&V_{\text{ce},f_i,s} = V_{\text{ce},f_j,s} \\ &\wedge V_{\text{clk},f_i,s} = V_{\text{clk},f_j,s} \\ &\wedge ((V_{\text{clr},f_i,s} = V_{\text{clr},f_j,s} \wedge V_{\text{set},f_i,s} = V_{\text{set},f_j,s}) \\ &\vee (V_{\text{clr},f_i,s} = V_{\text{set},f_j,s} \wedge V_{\text{set},f_i,s} = V_{\text{clr},f_j,s})) \end{aligned} \tag{3.6}$$

However, at the moment the equation 3.4 is used within this thesis. Now we can express that, when two primitives are both flip-flops, and act the same way in all states, they belong to the same register.

$$\forall p_i, p_j \in P | T_{p_i,\text{FF}} \wedge T_{p_j,\text{FF}} \implies (G_{p_i,p_j} \implies R_{p_i,r} \wedge R_{p_j,r}) \tag{3.7}$$

The former expression can be translated into a graph matching rule, which can check all combinations of flip-flops, in the same way as the last expression. The rule expresses the same as the expression, only in a graph notation. The complete rule can be found in figure 3.8, but will be elaborated step by step: First, the two flip-flops, which do have to be checked against each other do have to be matched. This is done by matching two cell nodes with the type flip-flop. Two ports in the graph have the same value when they they are connected to a common source. While it is also possible that they might have the same value, if they are not connected to a common source, the assumption is that the synthesis tool will not instantiate more logic than necessary, which implies a common source for values that are the same. To check if all the control ports of the two flip-flops are the same, the control ports themselves, and the common source are added to the rule. Now this rule will match only two flip-flops, which do have a common source for all their control ports. Furthermore, an embargo edge between the two cell nodes is added. Such an edge forbids that the two cell nodes could match the same flip-flop. This is actually not forbidden, because a flip-flop will always be in the same register with itself, but it is unnecessary. Using this graph rule the set with registers can now be filled.

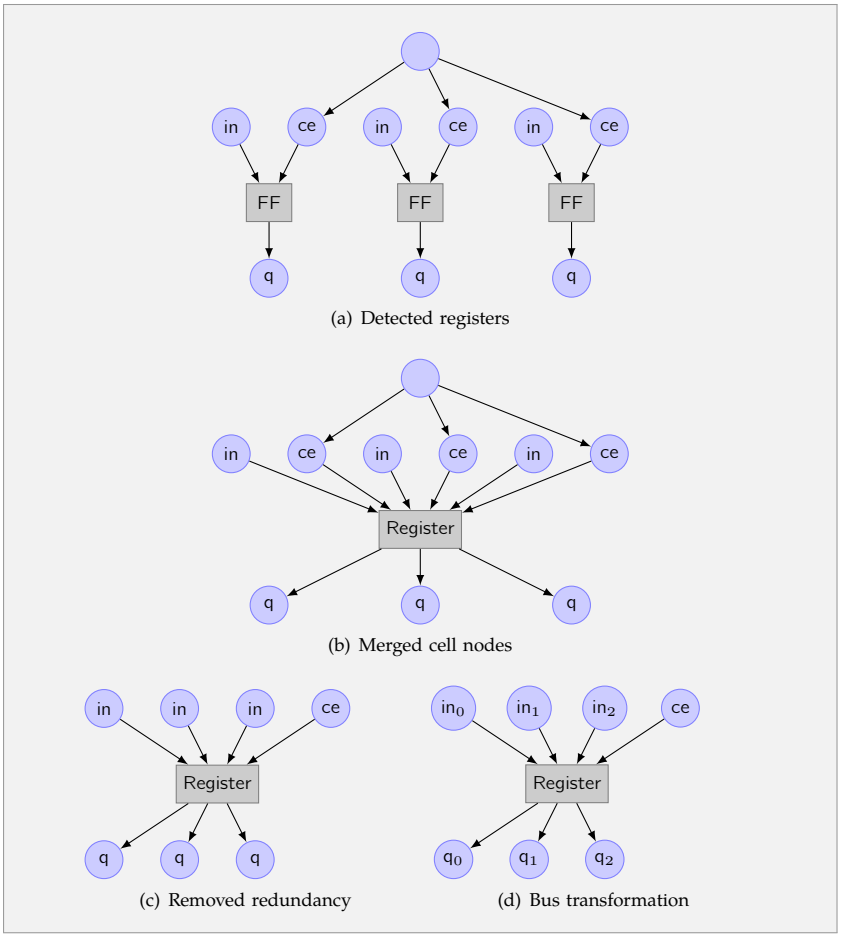


Figure 3.9: Flip-flops to register transformation

Flip-flop merging All detected register groups can now be merged together into registers. Figure 3.9(a) shows an example of a detected group of flip-flops. The set-, clk-, and clear ports have not been shown for simplicity. The cell nodes have been merged into the custom primitive 'Register' in figure 3.9(b).

For each group of flip-flops all cell nodes are merged into one new cell node, this cell node now represents a register, a custom primitive. Many of the ports connected to the merged cell node are redundant, hence ports with redundant behavior are removed. The redundant 'ce-ports' are still present at this step, and these are removed in figure 3.9(c). The last step involves changing all in-, and q ports to vector based ports, because they are now part of a bus. This final register is shown in figure 3.9(d).

During the testing of the implementation of flip-flop grouping, there were some registers, which were wider than expected. The problem was that this analysis was performed on the intermediate format, as provided by Precision, on which certain optimizations are not yet performed. One of these optimizations is removing redundant, and unnecessary primitives. It is possible to describe several different registers, which each hold the same information, but are read by different parts of the hardware, see for example graph 3.10(a). Formally we can describe this as if two flip-flops that are not the same, within a register group share a common source node on the in port, one of them is redundant.

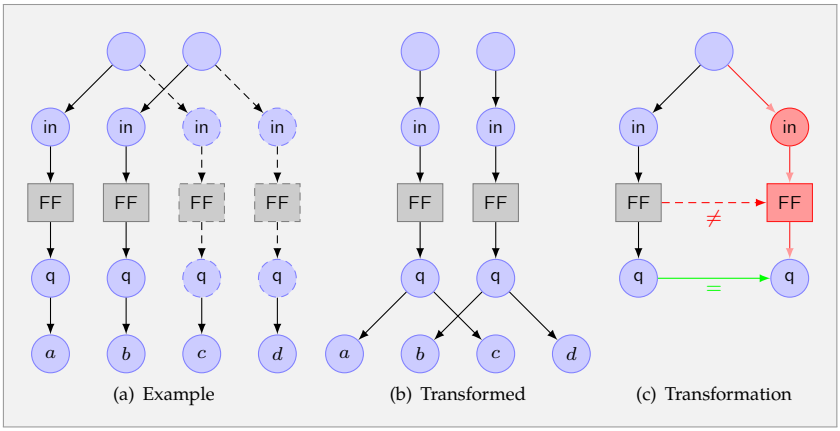


Figure 3.10: Redundant flip-flops

In that case, the redundant flip-flop can be removed. The following function will check if one of the two flip-flops is redundant:

$$U_{f_i,f_j} = (f_i \neq f_j \wedge (\forall s : S | V_{\text{in},f_i,s} = V_{\text{in},f_j,s})) \tag{3.8}$$

The graph transformation rule, which does this is depicted in graph 3.10(c). This rule matches flip-flops of which one is redundant, the embargo node forbids them to be the same cell node. One of the flip-flops, including its in port is removed. Furthermore, the merge edge, annotated with the '=', will merge both q ports, so that both original q ports will have the same value.

Register grouping At this moment the graph contains a set of registers, a subset of these registers can form a register bank. The next step into the detection of register banks is to determine which of them can be grouped together. An important requirement for a register bank is that all registers receive the same data. At this moment the register bank detection is limited to register banks with a single write port. Hence all data in ports of the register banks will share an array of common nodes. Formally we can describe this in a few steps. Let $W_{i,p}$ be a function, which returns the set of ports for primitive p, with label i. When the specified input port is an array, all of its input ports are present in this set. Furthermore, let X_n be a function, which will return a set which has all the predecessors of all the nodes in set n in it. Finally, let $M_{r,b}$ be a function which specifies that register r belongs to register bank b. The following expression checks if two registers belong to the same register bank:

$$\forall p_i, p_j \in P | T_{\text{REG},p_i} \wedge T_{\text{REG},p_j} \wedge X_{W_{\text{in},p_i}} = X_{W_{\text{in},p_j}} \implies M_{p_i,b} \wedge M_{p_j,b} \tag{3.9}$$

The graph matching rule, which does the same thing is shown in figure 3.11. First it selects two registers, which cannot be the same register due to the embargo edge. Then it uses the universal quantifier to ensure that for all in ports there exist a common source node. When this graph rule matches on two registers, they belong to the same register bank. This graph will be used to group registers together that will form a register bank.

Register bank behavior analysis After the moment when several registers are grouped together, additional analysis has to determine if they can be merged

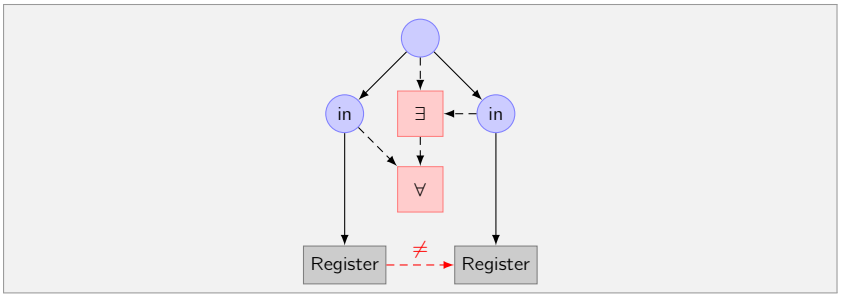


Figure 3.11: Data port matching

into a register bank. The analysis steps, which will be performed on the groups of registers are: Address logic detection, reset- and set detection, and read ports detection.

Address logic detection The clock enable inputs of a group of registers are analyzed to see if they behave as a single write port. A group of registers behaves as a single write port if at most one of the registers is updated per rising edge of the clock in all states. Therefore a *Clock Enable (CE)-graph* for the group of registers is analyzed. The CE-graph is defined as all the cells, which influence the clock enables of the registers. This means that each node in the CE-graph has a combinatorial path to one of the clock enables. First we define a formula, which will determine if a specific node can influence the clock enable ports of the register bank. The function $T_{\text{state},s}$ is used to match all primitives that have an internal state. In the following formula, let e be the set of clock enables in a register bank, c be the specific node and g be the graph:

$$H_{e,c,g} = \exists \langle s,t \rangle \in E_g | s = c \wedge \neg T_{\text{state},s} \wedge (t \in e \vee H_{e,t,g}) \tag{3.10}$$

Using this equation all nodes can be checked to determine if they belong in the CE-graph.

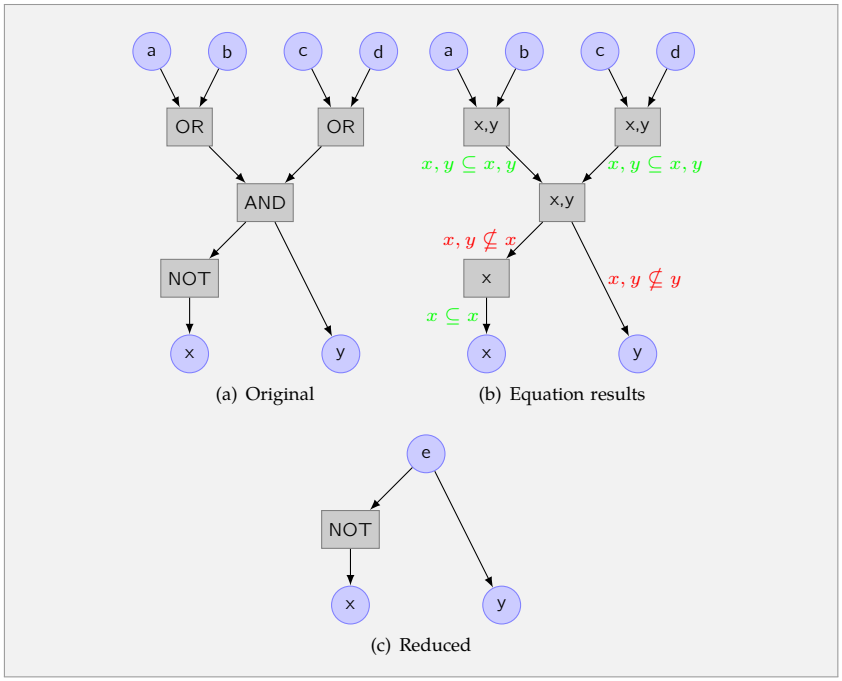
$$\forall n \in V_g | H_{e,n,g} \tag{3.11}$$

This graph has to be analyzed in order to determine where the address, and write enable signals are instantiated. The size of this graph might cover a large portion of the complete hardware graph. Hence the size of the entity vector, which has influence on the clock enables can be large as well. Determining for which input patterns a specific clock enable port is active can be solved using the *boolean satisfiability problem*, which is NP-complete [33]. Hence many of the clock enable graphs for the register banks, which were analyzed are too large to be analyzed directly. However, much of the information in the clock enable graph is redundant, and can be removed to simplify the detection of the addressing logic, which reduces processing time. The only primitives that need to be in this influence graph are cells which 'directly' influence the CEs.

Nodes which do have more influence than at least one of their predecessors are identified as the new input ports. All of their predecessors are removed from the graph. First a new function is defined, which takes a node, all the clock enables, and a graph as arguments and returns the set of clock enable ports of that graph, which are connected to that node.

$$O_{v,e,g} = \{\forall e' \in e | H_{\{e'\},v,g}\} \tag{3.12}$$

The new input ports are identified by the following equation:



(a) Original (b) Equation results (c) Reduced

Figure 3.12: Influence graph reduction example

$$\forall v \in V_g | \exists \langle s, t \rangle \in E_g \bullet v = s \wedge O_{s,e,g} \not\subseteq O_{t,e,g} \tag{3.13}$$

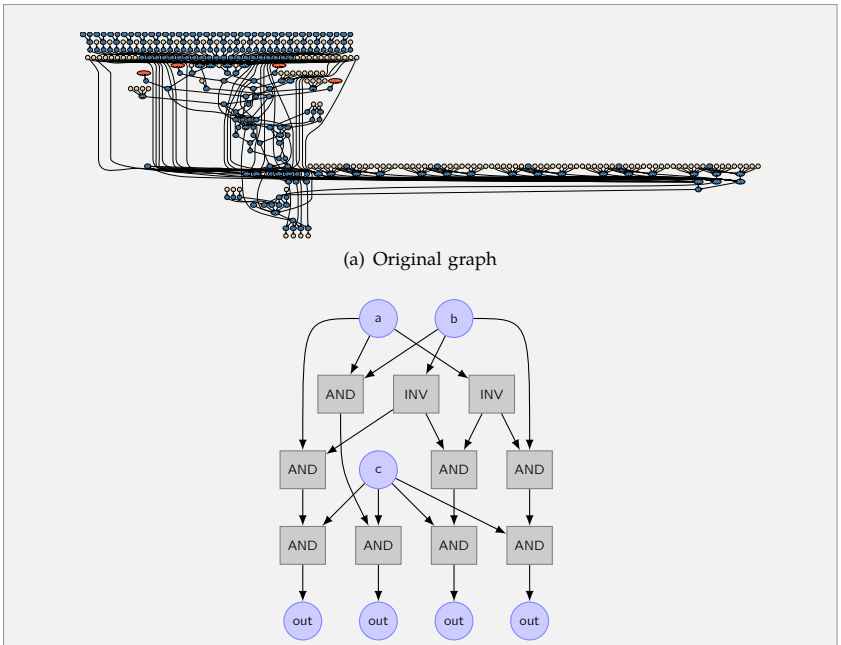
We illustrate this technique with a small example. See figure 3.12(a) for a small hardware graph with four inputs ports (a, b, c, and d), and two output ports (x, and y). For the set of inputs there are sixteen possible patterns. However, many of these patterns result in the same output values at the output ports.

See figure 3.12(b) for a graphical representation of how the last equation is applied to the graph. In each cell node the result of the function $O_{v,e,g}$ is shown. At all the edges, the comparison of the influence of the two nodes is shown. In this graph it can be seen that the output port of the AND-cell is identified as a new input port, because it has more influence on the output ports than any of its successors. Hence the output of this port becomes a new input port, reducing the number of input patterns from sixteen to two.

Figure 3.13(b) shows a reduced influence graph. The original graph, of which a portion is depicted in figure 3.13(a), has 40 inputs ports. After the reduction only three input ports remained. From the logic can be derived that input ports a, and b generate the address, and input port c acts as a write enable port.

Resolving mutual exclusiveness The resulting influence graph is converted to a *primitive influence graph*. This graph should only contain the binary operators AND, OR and the invertor as unary operator. Other operators such as a n-port AND, OR, etc operators are converted to binary operators.

For each output port its relation with the input ports is described in an equation. To ease the comparison of mutual exclusiveness the equations are reduced to their minimal *Sum of Products (SOP)* equivalent. The 'Boolean Expression Reducer' [34] software was used to minimize the equations. These reduced equations are tested against each other to see if more than one can be



(a) Original graph

(b) Reduced graph

Figure 3.13: Clock enable input reduction

Register	Clock enable formula
1	$I_1 I_2 I_5 \overline{I_3}\,\overline{I_4}\,\overline{I_6}$
2	$I_{10} I_{11} I_2 I_4 \overline{I_{12}}\,\overline{I_3}$
3	$I_{13} I_{14} I_2 I_3 I_4 \overline{I_{15}}$
4	$I_1 I_2 \overline{I_3}\,\overline{I_4}\,\overline{I_5}\,\overline{I_6}$
5	$I_{10} I_2 I_4 \overline{I_{11}}\,\overline{I_{12}}\,\overline{I_3}$
6	$I_{13} I_2 I_3 I_4 \overline{I_{14}}\,\overline{I_{15}}$
7	$I_1 I_2 I_5 I_6 \overline{I_3}\,\overline{I_4}$
8	$I_{10} I_{11} I_{12} I_2 I_4 \overline{I_3}$
9	$I_2 I_3 I_7 I_8 \overline{I_4}\,\overline{I_9}$
10	$I_{13} I_{15} I_2 I_3 I_4 \overline{I_{14}}$
11	$I_{13} I_{14} I_{15} I_2 I_3 I_4$
12	$I_2 I_3 I_7 I_8 I_9 \overline{I_4}$
13	$I_1 I_2 I_6 \overline{I_3}\,\overline{I_4}\,\overline{I_5}$
14	$I_2 I_3 I_7 \overline{I_4}\,\overline{I_8}\,\overline{I_9}$
15	$I_2 I_3 I_9 I_7 \overline{I_4}\,\overline{I_8}$
16	$I_{10} I_{12} I_2 I_4 \overline{I_{11}}\,\overline{I_3}$

Table 3.4: Clock enable equations

active at the same time. See table 3.4 for an example of these equations, the equations are based a register bank found in the NoC-router.

The software package uses the Quine McCluskey algorithm [35] to reduce the equations, which has a complexity of NP. Because of the reduction of the influence graph, it did not present a problem during the tests performed during the implementation of this algorithm, because the processing time required for this step did not have a significant impact on the total processing time. When the processing time required increases, there are several promising options to reduce the processing time for this step, such as work presented by Coudert, and Arevalo and Bredeson [36, 37].

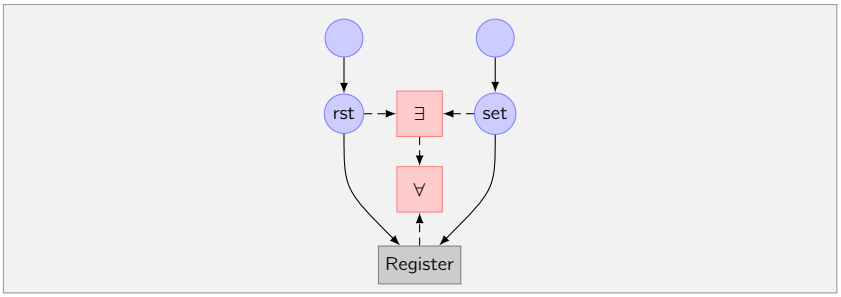


Figure 3.14: Reset- and set detection

Reset- and set detection The registers within a register bank do have reset-, and set ports. All the hardware designs that were analyzed during the development of the algorithm did either not reset their registers, or all in the same clock cycle. This led to the assumption that most register banks do reset or set all registers or none at all. It would be possible to extract register banks with selective reset or sets, but at the moment they will be discarded as register banks (See equation 3.4).

Therefore, a distinction must be made between these two types of register banks. As before with the detection of register banks, and data matching, we will determine if all reset- and set ports do have the same value in all states. The following expression must remain true, otherwise the register bank has to be discarded. Let the set R contain all register primitives in a register bank.

$$\forall r \in R, s \in S | \exists a \in \{0,1\}, b \in \{0,1\} | V_{\text{set},r,s} = a \wedge V_{\text{reset},r,s} = b \tag{3.14}$$

This expression can be transformed to a graph matching rule; See figure 3.14 for the rule. The rule matches the registers that do have a common source for their reset- and set port. This rule should match all the registers in a register bank. If it does not match all registers there is not a common source for all registers, which implies that this register bank should be discarded.

Read ports detection On the read side of the registers we must identify which registers are actually being read in a clock cycle. First the influence graph algorithm from the previous section was adapted to reduce the outgoing graph. Equation 3.10 was rewritten to check if a node from the set q, has a path to the current node c, in the graph g:

$$G_{q,c,g} = \exists \langle s,t \rangle \in E_g | t = c \wedge \neg T_{\text{state},s} \wedge (s \in q \vee G_{q,s,g}) \tag{3.15}$$

Using this equation the outgoing graph can be defined as the graph with the nodes from the following set, where r is the set of register cell nodes:

$$\forall n \in V_g | G_{r,n,g} \tag{3.16}$$

We determine which of the registers can have influence on the node v:

$$Q_{v,r,g} = \{\forall r' \in r | G_{\{r',v,g\}}\} \tag{3.17}$$

Now we adapt equation 3.13, so that only nodes, which do have equal or less influence than all their successors, are included.

$$\forall v \in V_G | \exists \langle s,t \rangle \in E_g \bullet v = t \wedge Q_{t,r,g} \not\subseteq Q_{s,r,g} \tag{3.18}$$

The outgoing graph is the graph with all nodes between the set of registers, r, and all the nodes from equation 3.18. This did not result in the expected

reduction, which was observed in the CE-graph. The outgoing graphs were, in almost every case, too large to be processed by the Boolean Expression Reducer. However in all outgoing graphs the outputs of all the registers were muxes. Because only this structure was found, the predefined search pattern technique, as mentioned in section 3.5.1, was used.

The structure must meet several requirements, namely:

Only muxes All registers only drive muxes. When a register does not drive a mux, it is possible that it always used. Further analysis, with for example a depth limited behavioral search, could still find muxing behavior in these components, but these structures are disregarded at this point.

First we define the following function that will return all successors for node v at depth d:

$$Y_{v,d,g} = \begin{cases} v & d = 0 \\ \forall \langle s,t \rangle \in E_g | s = v | Y_{t,d-1,g} & d > 0 \end{cases} \tag{3.19}$$

Using this equation we can check whether there are cell nodes directly connected to the output of the registers that are not muxes. Because there is always an output- and an input port between two adjacent cell nodes, we look at all successors at depth 3 from the register cell nodes:

$$\nexists r \in R | \exists v \in Y_{r,3,g} | \neg T_{\text{mux},v} \tag{3.20}$$

Mux inputs When the input of the muxes only consists of the outputs of the registers within the register bank, they are easy to extract. All muxes, found during the development of the register bank detection algorithm, complied to this. Hence the assumption is that most muxes, which are used by a register bank, will only mux data from the register bank itself. Let M be the set of cell nodes for the muxes as defined by $Y_{r,3,g}$. First we define a function, which will return all predecessors at a certain depth, similar to the function in 3.19:

$$Z_{v,d,g} = \begin{cases} v & d = 0 \\ \forall \langle s,t \rangle \in E_g | t = v | Z_{s,d-1,g} & d > 0 \end{cases} \tag{3.21}$$

With this function we can check whether there are no inputs for the data port of the mux, the a port, that are not connected to a primitive within the register bank:

$$\nexists m \in M | \exists \langle s,t \rangle \in E_g | L_s = \mathrm{a} \wedge Y_{s,2,g} \not\subset R \tag{3.22}$$

Behavior analysis result When all analysis have confirmed the behavior of the registers as a register bank it can be extracted. The register bank can be replaced by a properly configured clocked_ram component. How this this done can be found in the next section.

3.5.2 Replacement

Because the register bank behaves in the same way as a CLOCKED_RAM primitive, it can be replaced by that. Then the RAM extraction, as explained in section 3.4, can be used to extract the CLOCKED_RAM primitive.

The register bank detection obtained the following information:

- The data input port for all registers receives the same values.

- Of all CEs, at most one can be active at the same time.
- The reset-, clk- and set ports for all registers share common sources.
- The muxes connected to the data ports only mux data from the register bank.

This information is used to implement the logic that is connected to the ports of the CLOCKED_RAM primitive.

clocked_ram instantiation A new cell node, with the label CLOCKED_RAM, is added to the graph. Now all ports for the clocked_ram primitive need to be added, and connected to the graph. The following paragraphs describe how each of the ports is added, and what other logic is instantiated to correctly drive the ports.

clk port Because the clk ports of all registers share a common source, it can be connected to a new port node with the label CLK.

we port Only when one of the CEs is active the CLOCK_RAM is written to. Hence a new OR primitive is added to the graph, the inputs of that primitives are connected to the CEs.

addr[{ess,2,3, ... ports}] These are the read ports of the CLOCK_RAM primitive. They are connected to the address port of the muxes. Each register is annotated with the address of the input port of the mux that it is connected to. This information will later be used to correctly implement the write address port.

address port This is the write port of the memory. In the last step, when the read ports are connected, each register was annotated with a read address. This information is used to correctly identify which clock enable belongs to which address. Because the CEs signals, which form a sort of one hot encoded address, do have to be converted to a normal address, which can be read by the CLOCKED_RAM primitive. A SELECT-TO-ADDRESS primitive is added to the graph. This primitive performs that transformation.

data port The data port is directly connected to the data port of one of the registers.

q[{2,3,... port}] For each set of muxes that form a read port, the outputs of those muxes are connected to a new Q port.

Reset- and set replacement Each register has a reset-, and set port. During the register bank detection it was confirmed that those ports shared common sources. When they are connected to a FALSE primitive, the register bank is never set or reset. However, when they are connected to other primitives, the flip-flops within the register bank can be set and/or reset. The main reason for the memory and register bank extraction is that it reduces the memory bandwidth to the state storage. But when a whole register bank can be modified in a single clock cycle, all locations within that register bank do have to be updated, which implies that the bandwidth for this register bank cannot be reduced. There are two solutions to this problem. First, if we assume that the activation of the reset or set signal of the register bank rarely occurs, the

pipeline can be stalled and all locations can be updated sequentially. The disadvantage of this technique is that it imposes some timing overhead, how much depends on the frequency of (re)sets, and the depth of the register bank. Furthermore, it requires changes to the pipeline, in order to facilitate the sequential update.

The second solution keeps an additional reset vector, which buffers the reset and set signals. On the read side of the register bank this vector is accessed to see if the data can be read from memory or that a vector containing zeros or ones must be returned. When new data is written to the register bank the specific entry for that location is reset in the vector. The advantages of this solution are that no changes are necessary within the pipeline of the simulator (the required vectors can be instantiated within the logic of the hyper cell), and that it does not impose any timing overhead (on the assumption that the clock frequency does not increase). The disadvantage of this solution is that it imposes a memory overhead, because for every register in the register bank one or two flip-flops are necessary to store the reset-, and set signals.

The second solution was chosen, mainly because the implementation of it is simpler than the first solution.

Flip-flop insertion To store the set- or reset signal a flip-flop is added to the graph for each location in the register bank. Each flip-flop reset port is connected to one of the CE-signals, because when a specific location is written to, that register it is no longer in its initial state. The set port for all the flip-flops are connected to either the reset- or set port from one of the registers.

Flip-flop selection When a location is being read from the register bank, the flip-flop for that location has to confirm that the location is available for reading or that its initial state should be returned. One mux is inserted into the graph for each of the read ports, then the data ports for that mux are connected to the flip-flops, and the address port to the same signals that are used for the address read port.

Value or initialization selection Depending on the value of the flip-flop selection mux the value from memory or an initialization vector should be returned. Therefore, a mux is inserted after the output port bits. These muxes selects either the data from the CLOCK_RAM primitive or the initialization vector, which at the moment is a vector of zeros or ones.

Primitive removal After all new primitives are added to the graph, the primitives, which represented the registers, and the muxes can be removed from the graph.

Example In figure 3.15 a small example is shown of a register bank with four locations that has been replaced with a CLOCK_RAM primitive. Originally the hardware design was simular to the register bank introduced as an example at the beginning of this section (See figure 3.6).

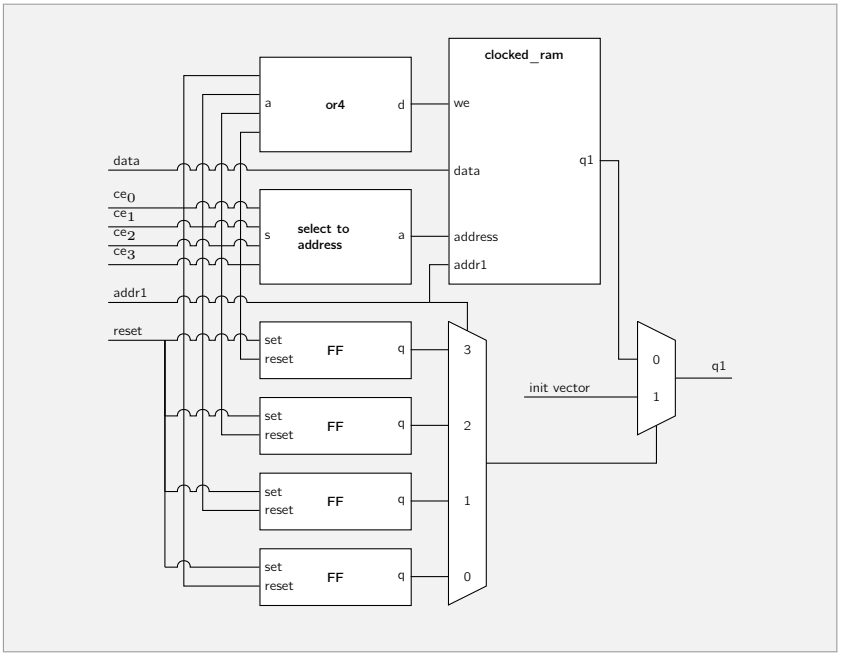
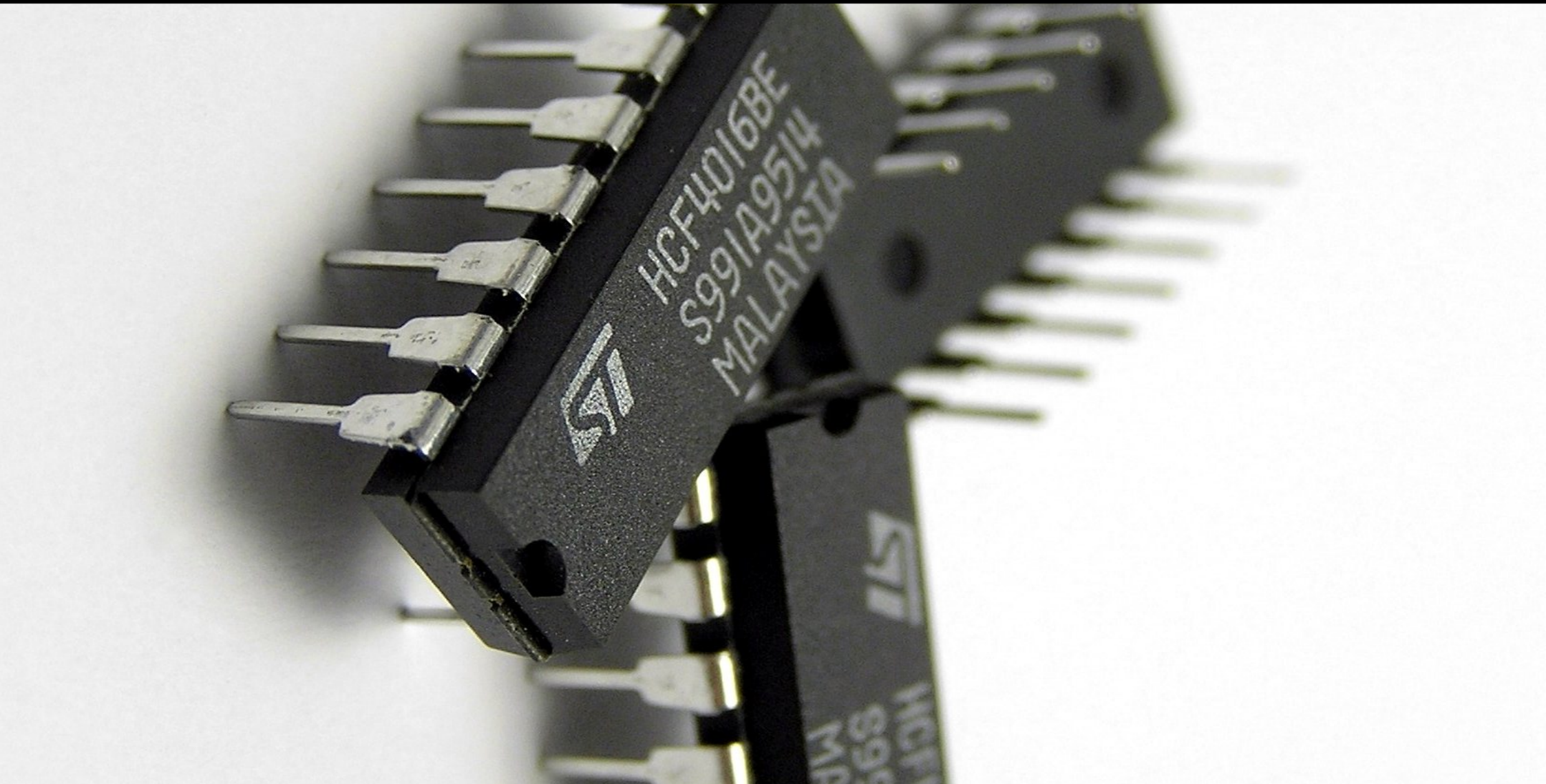


Figure 3.15: Register extraction with global reset

Chapter 4

State Storage



Abstract

How the memories of a hardware design are mapped onto the FPGA influences the performance of the simulator. This chapter introduces a mathematical model, which can be used to find mappings for the memories. For the simulator generator the mathematical model proved to be unusable, therefore some light weight heuristics have been developed. These heuristics are used to find good mappings within reasonable time.

Outline

This chapter will elaborate the mapping of the memories of the original hardware design to the memories of the FPGA. First, in section 4.1 the state storage hierarchy is elaborated. A small introduction to the dedicated memories on an FPGA is given in section 4.2. The model, which is used as basis for the mapping, is introduced in section 4.3, and is formally described in section 4.4. How the model is used will be explained in section 4.5, the results of this tool can be found in section 4.6. The model is slightly modified in section 4.7. Before some heuristics are introduced in section 4.9, a simple method to calculate a minimum bound for the mapping is introduced in section 4.8. Finally, the heuristics are evaluated in section 4.10.

4.1 State Storage Hierarchy

Before elaborating how the state structure looks like, the hierarchy of the state storage is described. The memory hierarchy can be divided into four different levels; For a graphical overview see figure 4.1.

Entity state's input In order to reduce the critical path length, the input of the instantiated entity is registered. Before the evaluation of an entity, the input state vector is stored in this register. Hence this is the lowest level of hierarchy where only a subset of the state information for a specific entity is available.

On chip state storage The dedicated memories of an FPGA are used to store as much of the state as possible. Ideally, all state information is stored on chip, but it may be necessary to use the on chip memories as a large cache for some off chip state storage.

Off chip state storage When the on chip state storage is not large enough to store the complete state, it is necessary to store the state in external memory. For example, one or more DDR chips may be connected directly to the FPGA in order to store the complete state, at a latency, and bandwidth penalty.

Host computer state storage The computer that is controlling the FPGA can store portions of the state from each clock cycle for analysis, debugging, logging, and other purposes.

This chapter will elaborate the second level of state hierarchy: the on chip state storage.

4.2 On chip State Storage

For now we assume that the state storage can be stored completely in the FPGA. An FPGA typically uses dedicated memory structures for the storage of 'large' amounts of data. The number of these dedicated memories range from just a few to more than a thousand [38]. The FPGA which will be used as development board in this project houses 288 dedicated memories.

In the Xilinx technology library these memories are called Block RAMs [38]. Each Block RAM can be configured independently. Configuration options include height and width, write behavior, etc. Because the state storage is not the only component in the simulator that stores large amounts of data, it cannot use all of the available Block RAMs. The other main contender for Block RAMs are the link memories. However, initial tests show that the link memories only need a fraction of the amount that the state storage requires [4]. This depends on the ratio of links, and memories in a hardware design, but the assumption is this observation holds for most hardware designs.

Each Block RAM has two independent ports, which both support reading, and writing. Using a time multiplexing technique it is possible to increase the number of ports, see section 5.1.1 for the details on the Block RAM multiplexing.

4.3 Model

Having multiple ports on a single Block RAM allows for the mapping of several memories of the original hardware design onto the same Block RAM. For the state storage, a mapping must be found such that all memories of the original

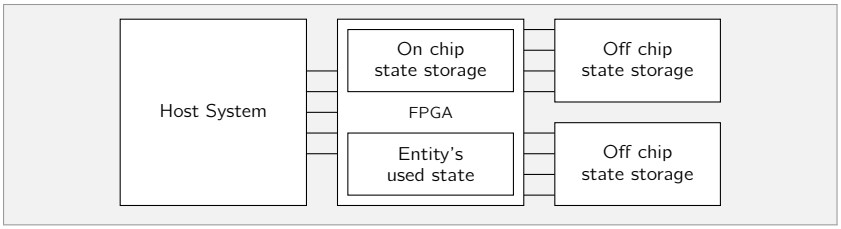


Figure 4.1: State storage in the simulator

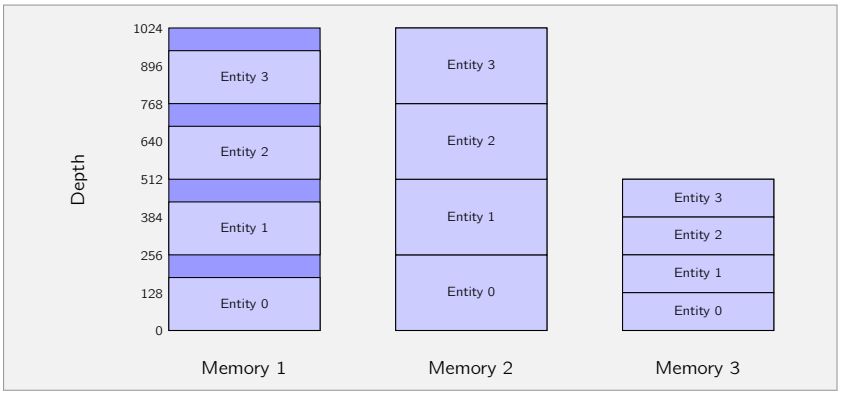


Figure 4.2: Entity multiplexing

hardware design are completely mapped onto the available set of Block RAMs. In the next section a mathematical model is introduced, which consists of several constraints a correct mapping must satisfy. This model could also be used to find an optimal mapping.

Entity multiplexing Depending on how many hyper cells there are instantiated, only a small number of entities will be evaluated in each clock cycle. Only the entities that are evaluated, need to retrieve their state vector. A simple solution to multiplex the entities, is to increase the depth of each memory by the number of entities that are simulated. For efficient storage and retrieval, all memories depths, which are not a power of two, are increased to the next power of two, so that addressing entities can be done by setting the most significant bits of the address port. The drawback of a depth increase is that not all locations of a memory are used, which imposes an overhead. In the worst case, when all memories have a depth which is one more than a power of two, the overhead will almost be 50%.

See figure 4.2 for some small examples on how entities are multiplexed. In the example there are four entities. Memory one does not have a depth on a power of two. The base memory was increased to a depth of 256. Memory two, and three do not need any increase in their depth.

Memory Normalization Depending on the hardware design, each memory in a hardware design will have its own arbitrary width, and depth. The Block RAMs onto which these memories will be mapped only support a small number of depth, and width configurations (See table 4.1 for the supported configurations). The reason that a Block RAM uses widths for the data ports of 9, 18 or 36, is that there is an extra parity bit for each 8 bits in the memory. But this parity bit can only be used to store additional data, if the width of the data port is set to 9, 18 or 36.

Furthermore, the Block RAM supports byte wide enable signals for the write

Table 4.1: Block RAM width configurations

Width	Depth	Total size (B)
36	512	2304
18	1024	2304
9	2048	2304
4	4096	2048
2	8192	2048
1	16384	2048

ports. These could be used to selectively update parts of a single location if the width of the data ports are set to 18 or 36. Bit wide enable signals could be instantiated using additional logic, but is not explored within this thesis. This means that we can set the width of the data ports on the Block RAM to 36, and not lose memory space, because we can normalize the memories. The implementation of the component that is responsible for generating the byte wide enable signals, and correctly retrieving the information from the location will be elaborated in section 5.1.3.

When a memory has a width less than 18, its depth can be halved and its width doubled, this can be done several times, until the width is larger than 18. This technique allows a single location in the Block RAM to contain multiple locations from the original memory. All memories with a width larger than the width of the Block RAM can be divided into several memories of 36 bits. This normalization process reduces the search space, because the mapping does not have to take the width of the memory into account, but the number of bins that are necessary for this simplified model remains the same. However, it must be noted that this might increase the amount of logic required to instantiate the state storage, because some logic is required to correctly generate the byte wide enable signals.

4.4 Mathematical Model

The basis for the model is the bin packing problem [39]. The bin packing problem defines how several items of variable sizes, weight, etc can be stored in a set of bins, and how to minimize the amount of bins that are used. In the normal bin packing problem the items themselves cannot be divided. However, since we can divide a memory in smaller pieces, and store each piece on a different Block RAM, our problem has more solutions. In this section a mathematical model is introduced, which formally describes which mappings are valid.

4.4.1 Input

The input of the model is the set of memories of the hardware design, memories are represented by items (See equation 4.2). These memories can have arbitrary widths, and depths. To reduce the complexity of the mapping they are normalized to the maximum width of a Block RAM. A specific item consists of its depth, and access. Depth represents the number of addresses in the memory after normalization, and the accesses represent the number of ports used for the memory (See equation 4.5). Another input for the mapping is the set of Block RAMs. Block RAMs are represented as bins (See equation 4.1), onto which the items should be mapped. How the Block RAMs are configured is described by the $depth$, and $accesses$ parameters, as seen in respectively equation 4.3, and 4.4.

$$b(ins) \quad 1,\ldots \tag{4.1}$$
$$i(tems) \quad 1,\ldots \tag{4.2}$$
$$d(epth) \quad n \tag{4.3}$$
$$a(ccesses) \quad o \tag{4.4}$$

$$it(em)_i \in i = \langle d, a \rangle \tag{4.5}$$

The depth, and accesses of a specific item can be retrieved by respectively $it_{i,1}$ and $it_{i,2}$.

4.4.2 Output

The main output consists of the *'mappings matrix'* (Ma, See equation 4.6), which describes how much of an entity is mapped onto each bin. The other outputs are needed to construct some of the constraints.

$$Ma(ppings) \quad (1,\ldots;1,\ldots) \tag{4.6}$$
$$B(inarymappings) \quad (1,\ldots;1,\ldots) \tag{4.7}$$
$$(U)sed \quad 1,\ldots \tag{4.8}$$

As mentioned above the Mappings matrix describes the actual mapping. A specific $Ma_{i,b} \in Ma$ indicates how many addresses of a specific item i are mapped onto bin b. The matrix *'Binarymappings'* (B) is used to determine if a specific item maps onto a bin (See equation 4.9). Constraint 4.12 is used to correctly determine the values within this matrix.

$$B(inarymapping)_{i,b} \in \quad B = \begin{cases} 1 & \text{item } i \text{ might map onto bin } b \\ 0 & \text{item } i \text{ does not map onto bin } b \end{cases} \tag{4.9}$$

Furthermore the vector *'Used'* (U) determines if a bin might be used (See equation 4.10). Equation 4.13 is used to correctly set all the entries in vector U.

$$U(sed)_b \in \quad U = \begin{cases} 1 & \text{bin } b \text{ might be in use} \\ 0 & \text{bin } b \text{ is not used} \end{cases} \tag{4.10}$$

4.4.3 Constraints

Not all mappings are valid. Only when a mapping adheres to the following constraints it is valid.

An item cannot map a negative amount of content onto a bin:

$$Ma_{i,b} \geq 0 \quad \forall i, b \tag{4.11}$$

An entry in the matrix B is forced to one when an item i maps to bin b. In principle all entries in this matrix can be set to one to fulfill the constraint. However, when minimizing the cost function these values can be set to zero, which will indirectly decrease the cost function.

$$Ma_{i,b} \leq it_{i,1} \cdot B_{i,b} \quad \forall i, b \tag{4.12}$$

As with the constraint above, when an item maps onto a bin, the entry for that bin in vector *'Used'* will be forced to one.

$$\sum_i Ma_{i,b} \leq d \cdot U_b \quad \forall b \tag{4.13}$$

The amount of content in a bin is limited by its depth:

$$\sum_i Ma_{ib} \leq d \quad \forall b \tag{4.14}$$

The amount of access in a bin is limited by its accesses:

$$\sum_i it_{i,2} \cdot B_{i,b} \leq a \quad \forall b \tag{4.15}$$

Each item is completely mapped:

$$\sum_b Ma_{ib} = it_{i,1} \quad \forall i \tag{4.16}$$

4.4.4 Minimize

In the previous paragraphs, the input, output, and the constraints on them have been introduced. The first objective of the state storage mapping is to reduce the number of Block RAMs necessary. Therefore the vector U was introduced. Using this vector, the total number of used bins is calculated.

$$\textbf{Minimize: } \sum_b U_b \tag{4.17}$$

If a specific item is mapped onto many different bins, it costs more hardware to combine the different portions of that item. Therefore, when a solution with respect to the number of Block RAM is found, the spread of all items can be minimized using the following cost function.

$$\textbf{Minimize: } \sum_{ib} B_{i,b} \tag{4.18}$$

Previous minimizations did use a fixed number of access ports for the bins. But, since an access of more than two requires the Block RAMs to be clocked at a higher clock frequency, this can lead to slower clock frequencies for the simulator. Therefore, minimizing the number of accesses on all bins can lead to a faster simulation.

$$\textbf{Minimize: } a \tag{4.19}$$

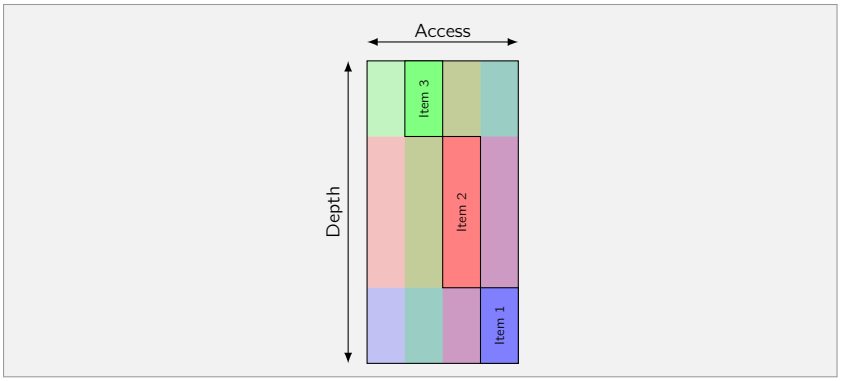


Figure 4.3: 2D Representation of a mapping

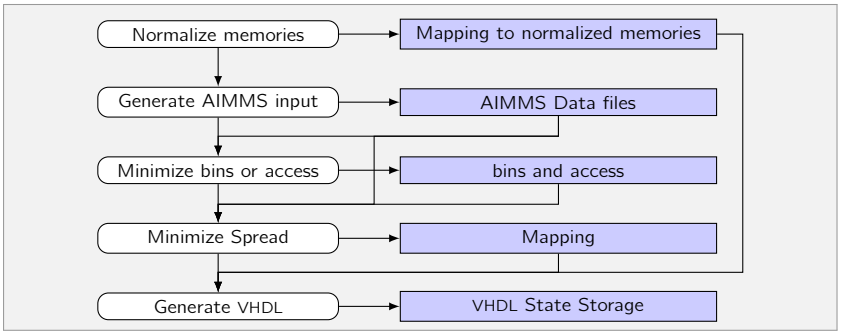


Figure 4.4: Tool flow for state storage VHDL generation

4.4.5 Example mapping

Figure 4.3 depicts a mapping of three items onto one bin. Each item does have its own depth, and access port. As can be seen in the figure, the items each have their own address, and their own ports, which cannot overlap. Not all accesses of the bin were used but since the depth is completely used, there is no space left for an additional memory.

4.5 Tool Flow

The mathematical model from the previous section can be modeled in an optimization tool, and solved using a constraints solver. The optimization tool used within this project is AIMMS. Figure 4.4 shows an overview on how AIMMS can be used within state storage generation flow. In the figure the white rounded boxes represent the states of the flow, and the dark rectangles the deliverables, which are used in other states of the flow.

4.6 Initial AIMMS Results

A large test case was setup using the memories from the Montium, see table 4.2 for a detailed view which memories were used. The input of the model consisted of the memories, and parameters for the depth and accesses. The depth parameter, which represents the depth of a bin, was set to the depth of a Block RAM, i.e. 512. The parameter 'accesses' was varied (2, 4, 6, and 8), to see what kind of impact this parameter has on the number of used bins. Furthermore, the number of entities for which the state should be stored was varied (1, 2, 4, 8, 16, and 32).

Table 4.2: AIMMS results - Bins used

			Bins used *Entities*					
			1	2	4	8	16	32
Optimal	*Access*	2						
		4				≥ 130		
		6				104	207	
		8	≤ 49	≤ 52	≤ 58	104	207	414

The AIMMS model was executed on the ILOG CPLEX 11.1 [40] solver with default settings, see table 4.2 for the results. Only a few of these executions terminated within reasonable time, i.e. less then a few hours.

Some executions did return a correct solution, but AIMMS could not prove that those solutions were optimal. The solver works towards the optimal solution from two sides. First, it tries to find a correct solution for the model, where the result of the cost function applied to this solution is the upperbound. The solver tries to change the solution in order to find a smaller upperbound. Second, the solver can prove that there are no solutions under a certain bound. This bound is known as the lower bound. The solver tries to increase this lower bound by proving that there are no solutions for which the result of the cost function are smaller. When the lower bound, and upperbound are equal, the solver knows that the current solution is an optimal solution, in the table these are the entries without additional operators. In some cases the solver could not let the bounds meet, hence it knows a correct solution, but cannot prove that there are no smaller solutions, which are represented in the table by the entries with a $\leq$ operator. In these cases the upper bound was shown in the results table.

One execution did find a lower bound, but had not found a feasible solution, which is represented in the table with a $\geq$ operator. One set of parameters, with accesses 4 and entities 8, was run for over 16 days without an actual solution.

4.7 AIMMS Model Modification

The AIMMS model uses indexed bins in order to correctly count which bins are used, and how many accesses there are on a specific bin. The result is that, when two bins swap their contents, it is another solution, but for the state storage these two solutions are the same. Therefore, the solver does a lot of redundant work.

One technique is to order the bins on their content. The same mappings are still possible, but since the bins themselves are ordered, there are less solutions which are the same. A simple ordering orders the bins on the amount of content. The following equation forces the amount of a bin with a lower index to have more content than a bin with a higher index:

$$\sum_i Ma_{ib} \geq \sum_i Ma_{(i+1)b} \quad \forall b \tag{4.20}$$

The executions still did not deliver additional results within a reasonable time, which was at least eight hours.

Table 4.3: Minimum bound - Bins used

			Bins used *Entities* 1	2	4	8	16	32
Access	*Access*	2	186	196	216	259	345	539
		4	93	98	108	130	173	270
		6	62	66	72	87	115	180
		8	47	49	54	65	87	135
	Depth		13	26	52	104	207	414
Max	*Access*	2	186	196	216	259	345	539
		4	93	98	108	130	207	414
		6	62	66	72	104	207	414
		8	47	49	54	104	207	414

4.8 Minimum Bound

Because of the run time, and lack of solutions in many test runs, the AIMMS model cannot be used to find an optimal mapping for the state storage. Since there are essentially two constraints, the depth and accesses on the bins that limit the mapping of items in a bin, it is possible to construct a minimum number of bins that are required.

In table 4.3, in the rows 'Access', and 'Depth', the minimum number of bins is shown, when the only parameters that constrains the mapping are respectively the accesses and depth. The maximum of these two results in a minimum bound. This is shown in the 'max' row.

4.9 Heuristics

Besides the implementation of the model in AIMMS it is also possible to map the items to bins using heuristics. Heuristics will find a feasible solution fast, but cannot guarantee that the mapping is optimal. Several heuristics were developed. In the following sections they will each be elaborated. The quality of the heuristics can be evaluated using the minimum bound as presented in table 4.3.

4.9.1 Naïve

Each memory will be mapped to a dedicated bin. Hence no sharing of bins is possible. When a memory is too large for a single bin, it will be divided into smaller portion, which each map to a dedicated bin. This heuristic will obviously find the largest solution in the number of bins that are used but will also have the least amount of spread. A drawback of the naïve heuristics is that it does not use additional access ports.

4.9.2 Merging

Several memories will be grouped together in a single bin. As with the previous heuristic, when a memory is too large for a single bin, it will be divided into smaller portions. Because several memories will share the same bin the solution found can be smaller, and because only large memories are divided it will still obtain optimal spread values. The order in which memories are merged is not specified, but should not violate any constraints.

4.9.3 Small First Merging with Padding

Instead of only merging small memories randomly together, it is also possible to use memories, which are larger than a single bin as padding. The padding should fill up the space of the bins, which are not used. Furthermore the list of memories can be sorted on depth, so that the padding is used efficiently. First, a large memory is selected as padding candidate. From the list of sorted memories, a number of memories are selected, from the beginning of the list, such that their depth do not exceed the maximum depth and leaves enough accesses for the padding memory. Now, the bin is completely filled with contents from the padding memory. As long as not all content is mapped this technique is applied.

For example, lets examine a case where there are many small memories and one large memory. E.g. memories with their depths from the following set: {64, 64, 64, ..., and 832}, the depth of each bin is set to 512, a bin has eight access ports, and all memories require two access ports. The largest memory is selected as padding, this is the memory with a depth of 832. Because this padding already consumes two access ports, six remain to be used for the smaller memories. The first three memories together use six access ports, their combined depth is 192, leaving 328 locations unused within the first bin. This padding memory is partly mapped onto these 328 locations, hence the first bin is complete full, leaving 512 of the 832 locations of the largest memory unmapped. The same procedure is repeated until all the memories are completely mapped onto the bins.

4.9.4 Large First Merging with Padding

The same as the previous heuristic with the exception that it tries to merge the largest memories, which do not exceed the depth of a single bin.

4.10 Heuristics Results

See table 4.4 for an overview of all results on bin usage. The naïve heuristics uses a constant number of bins, and presents the upperbound on bin usage.

Because this heuristic does not do any optimization, the bin usage remains constant, when the access on the bins increases, i.e. it does not use the additional ports. Note that because all memories within the Montium use two access ports, the other heuristics deliver the same output, when they can only use two access ports.

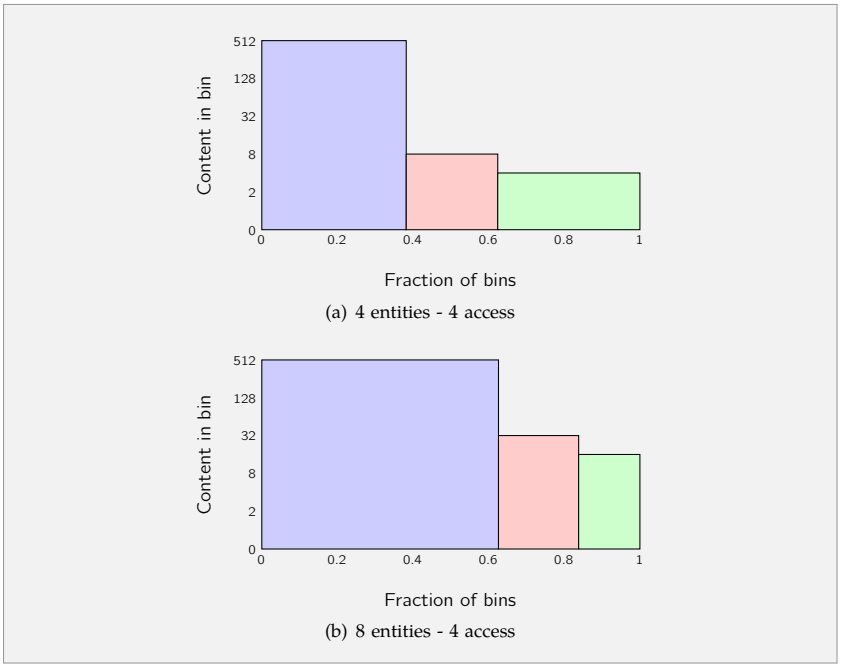
The merging heuristic presents a significant reduction in the numbers of bins used, when the number of access ports increases.

The last two heuristics also use merging but before the merging the list of memories is sorted. Hence larger reductions are feasible because the memories are combined more efficiently into the bins. The version, that first maps small memories together, seems to perform slightly better on some parameters, where on other there is no difference.

Because of the small increase in bin usage, when only a few entities are simulated, it can be concluded that in those cases most bins are sparsely used. Only when more entities are simulated, the bins are almost completely filled. For example, see the large, and small heuristics where at 16 and 32 entities the number of bins remains almost constant for 4,6 and 8 access ports. In figure 4.5 two histograms are shown which give an overview on how the contents is spread over the bins. The histogram in figure 4.5(b) has more bins completely filled than the histogram in figure 4.5(a). These figures represents

			Bins used (Entities)					
			1	2	4	8	16	32
Large	*Access*	2	186	196	216	259	345	539
		4	99	105	118	145	208	415
		6	66	71	80	104	207	414
		8	50	54	60	104	207	414
Small		2	186	196	216	259	345	539
		4	99	104	115	137	208	415
		6	66	69	77	104	207	414
		8	50	52	58	104	207	414
Merging		2	186	196	216	259	345	539
		4	98	108	130	173	270	467
		6	69	79	101	147	245	444
		8	54	65	87	135	233	434
Naïve		2	186	196	216	259	345	539
		4	186	196	216	259	345	539
		6	186	196	216	259	345	539
		8	186	196	216	259	345	539

Table 4.4: Evaluation of heuristics, and parameters - Bins used



(a) 4 entities - 4 access

(b) 8 entities - 4 access

Figure 4.5: State Storage Histogram for small first heuristics

Table 4.5: Evaluation of heuristics, and parameters - Spread increase

		Spread Increase *Entities*					
		1	2	4	8	16	32
Large	2	0	0	0	0	0	0
	4	10	12	19	29	67	270
	6	11	14	21	43	168	373
	8	12	15	21	68	190	394
Small	2	0	0	0	0	0	0
	4	10	10	13	13	67	270
	6	10	10	13	42	165	371
	8	10	10	13	79	193	399
Merging	2	0	0	0	0	0	0
	4	0	0	0	0	0	0
	6	0	0	0	0	0	0
	8	0	0	0	0	0	0
Naïve	2	0	0	0	0	0	0
	4	0	0	0	0	0	0
	6	0	0	0	0	0	0
	8	0	0	0	0	0	0

the parameters for 8 and 4 entities respectively (with access 4). The histogram for 16 entities with 4 access is not shown; 206 out of 208 bins were completely used.

Furthermore, the state storage generator, as will be explained in section 5.1.3, has to instantiate extra components for memories that are mapped onto multiple bins. The *spread factor* is introduced as a measure that counts into how many pieces all memories have been divided. As a base case the naïve heuristic is used, since this heuristics does only divide memories that do not fit into a single bin. The spread of the other heuristics is compared to this base case in table 4.5. First, the merging heuristics also does not divide memories if they are smaller than a single bin, and will not use large memories as padding, hence it does spread the memories more than the naïve heuristic. The results of the small, and large heuristics are comparable, the small heuristics seems to perform slightly better. The overall trend is that when more access ports are used on the bins, the spread increases.

Chapter 5

Implementation

Abstract

The state storage pipeline is responsible for loading, and storing the state in the Block RAMs of the FPGA. The main component of the pipeline is the state storage component, which acts as the interface from the memories of the hardware design to the Block RAMs. The pipeline itself is responsible for correctly loading, and saving the state information in the state storage component. Due to the design of the pipeline it is possible to request an entity vector in each clock cycle.

Outline

In chapter 3 new techniques for extracting large amounts of memory were introduced. How these memories can be stored efficiently on an FPGA has been elaborated in chapter 4. In this chapter the VHDL implementation for the state storage pipeline, and the generation of the state storage component will be elaborated. Section 5.1 will introduce how the memory mapping is translated into a VHDL description of the state storage component. This state storage is then used in the state storage pipeline, which will be discussed in section 5.2.

Table 5.1: Example mapping

Item	Item address range	Bin	Bin address range	Bin port range
0	0 - 127	0	0 - 63	0 - 1
0	128 - 255	1	0 - 63	0 - 1
1	0 - 127	0	64 - 127	2 - 3
1	128 - 255	1	64 - 127	2 - 3

5.1 State Storage Component

The state storage component is responsible for implementing the mapping of the memories. In table 5.1 an example mapping is given. The mapping describes how two items, each with a depth of 256, and with two accesses, are mapped onto two bins. Obviously the mapping is not optimal, but it illustrates the two transformations that can be performed on an item. The first transformation that has been performed is the normalization step (See section 4.3) This can be seen in table 5.1, the address range of the item is twice as large as the address range it will occupy in the bin. This behavior has been implemented in the COMBINE component. The second transformation has split up the items, the resulting pieces have been distributed over the two bins. This behavior has been implemented in the ADDRESS MAP component. Furthermore, the bins that are used to store the items have an arbitrary number of ports. But the Block RAMs that will represent the bins only have two ports. Therefore, the MPRAM component has been implemented, which will time multiplex the ports of a bin on a Block RAM.

The COMBINE-, ADDRESS MAP-, and MPRAM component will be elaborated in the following subsections.

5.1.1 MPRAM component

The *n*-port RAM (NPRAM) uses time multiplexing to create virtual ports on a Block RAM component, this component is used by Rutgers to efficiently use the Block RAMs for the link ports [4]. A second clock, running at a higher clock frequency, is used to time multiplex those virtual ports onto the available dual ports. How many virtual ports can be instantiated depends on the clock frequency of the second clock.

However, for the state storage component the NPRAM has one shortcoming. Only half of the ports can be used for writing. The state storage memories might have more write ports than read ports. In that case several read ports would be unused on the NPRAM, which is undesirable.

Furthermore, a memory can be mapped onto an arbitrary address range within the Block RAM. The address from the original memory needs to be shifted to coincide with the allocated address range for that memory. The shift is implemented by the addition of an offset to the original address. When this addition is placed outside the MPRAM (See figure 5.1(b)), an adder is instantiated for each virtual port. The addition can also be placed within the MPRAM (See figure 5.1(b)), now the complete MPRAM requires two adders, regardless of the number of virtual ports.

The MPRAM does have the following control ports:

CLK The system clock

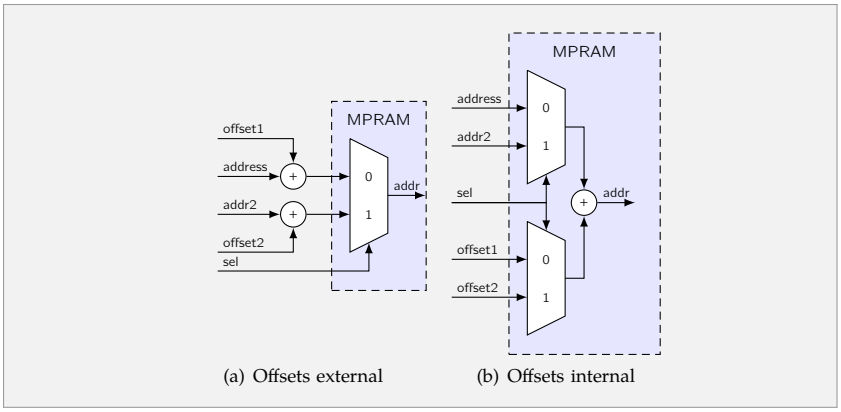


Figure 5.1: MPRAM offset placement

CLKM The Block RAM clock, should be set to a multiple of the CLK port, and phase aligned to the CLK port.

RST The RST signal for the Block RAM

Furthermore, each virtual port has the following ports:

DO The data output port

DI The data input port

WE[3..0] Byte wide write enable port

EN Enable port for this specific port

OFFSET Offset for address shifting, should be added to the address.

5.1.2 Address map component

Besides that the address range of a memory can be shifted within the MPRAM, it is also possible that the address range is split into several parts. These parts are stored on multiple MPRAMs. How these parts will be divided will be the responsibility of the ADDRESS MAP component. The component takes an address, and how the memory is split, and will generate control signals for which of the MPRAMs should be enabled for writing, and that the output from that MPRAM is also selected by the mux for reading.

The description of the ADDRESS MAP component ports:

ADDR The address that has to be read from a memory in the hardware design.

PORTIONS An array of indices, these indices divide memory in parts.

EN Using the ADDR and PORTIONS ports a one hot encoded selection array is calculated.

N Using the ADDR and PORTIONS ports the correct MPRAM is chosen for reading.

See figure 5.2 for an example. In the example a memory with more 256 locations is divided over two MPRAMs. The first 128 locations are mapped onto the first MPRAM. The address range of the original memory coincides with the address range onto which the locations are mapped, which implies that there is no offset required. The second part of the memory is mapped onto the second MPRAM. In this case the address ranges of the two memories do

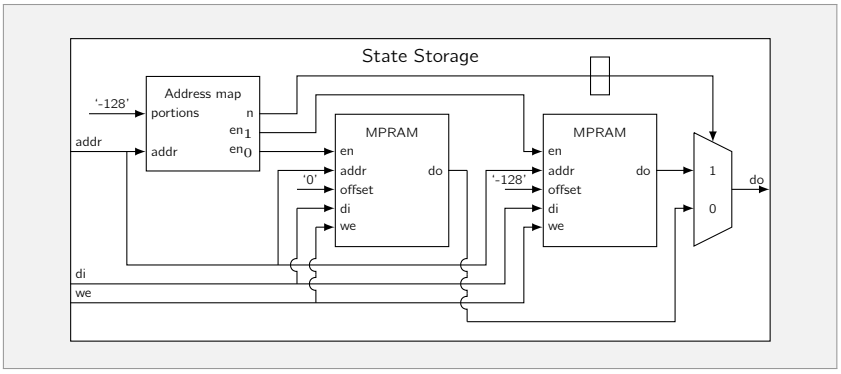
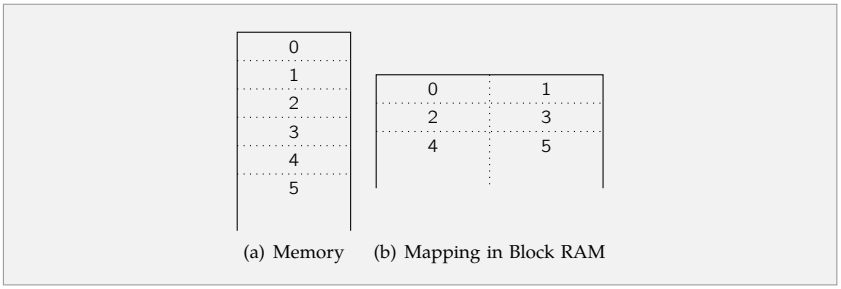


Figure 5.2: State storage example



(a) Memory (b) Mapping in Block RAM

Figure 5.3: Normalization example

not match. Therefore an offset of '-128' is required to match the two address ranges. The ADDRESS MAP component controls which of the MPRAMs should be active for a given address, and set the mux accordingly. For example; when the address 204 is supplied to the state storage the address map will only enable the second MPRAM, and set the mux to read that MPRAM.

5.1.3 Combine component

When an memory has been normalized, a location in the Block RAM will store more than one location from the memory. Figure 5.3 shows an example of how a memory is stored in the Block RAM, each location in the Block RAM stores two locations from the memory. Which of the parts of the location in the Block RAM should be updated or retrieved depends on the least significant bits of the address. The Block RAM component supports byte wide enables, which can be used to efficiently update only a part of a single location. The width of all Block RAM ports is configured at 36 bits, this width can be divided into four parts (the byte + a parity bit), hence the byte enables can be used when a memory is normalized to half or a quarter of its original size. Further normalization is currently not supported. For the output port, a specific part of the location should be retrieved. Because the data read from the Block RAM is synchronous, the address bits that are necessary for selecting the correct piece is registered. Figure 5.4 shows the combine component when the memory is normalized to half its original size.

5.1.4 Component generation

Using the MPRAM, COMBINE and ADDRESS MAP components the state storage component is generated.

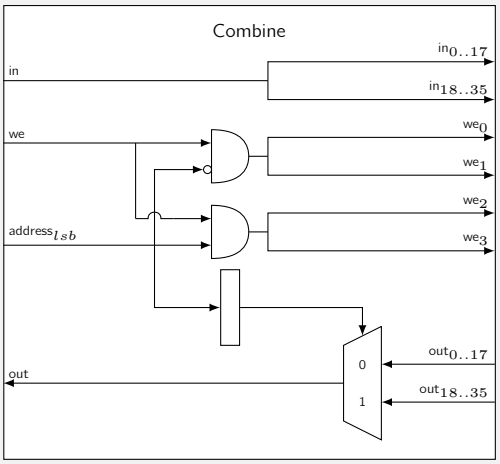


Figure 5.4: Combine component

5.2 State Storage Pipeline

As mentioned in chapter 2.1.4, a ping-pong mode memory is used to store the old and new state. But this technique imposes a large overhead, because the complete old-, and new state are available. However, since the output state vector now is much smaller than the complete state, i.e. only the difference with the old state, it should be enough to only store updates. After a system clock cycle is completed these updates can be stored transparently to the state storage.

The current flow, as described in section 2, did only extract flip-flops, and must therefore load the complete input state vector at once. For our reduced memory structures this does not suffice. First, the addresses which have to be read, are retrieved from the update vector. In the next clock cycle these specific memory locations and the flip-flops can be read from memory. Hence it will take one extra clock cycle to load the input state vector.

Update vector storage The update vector storage will be used to buffer all the outputs from the hyper cells, because, before all entities have been stabilized in a system clock cycle, the hyper cell needs the old state vector. Furthermore, the storage pipeline needs to store the addresses that are read from the memories, so that when this address has stabilized the correct memory location can be retrieved from the state storage.

Pipeline The state storage and the update vector storage are the main components in the state storage pipeline. Figure 5.6 gives an overview of the main components in the simulator; For further details see [4]. The state storage pipeline is responsible for loading the correct reduced state vector for the entity that has to be evaluated, (See figure 5.5 for a RTL description). There are three ports, which the controller should set for the correct retrieval of the state vector for an entity. First, the EVALUATE signals enables the evaluation of an entity. When the port is set to '1' the storage pipeline will retrieve the state for the entity which is given by the IAR port. When the UPDATE port is high, the buffered output from the previous evaluation for that entity in the update state is written to the state storage. The HCO port is the port that will output the state vector. At the same time the OA port will output the identifier for

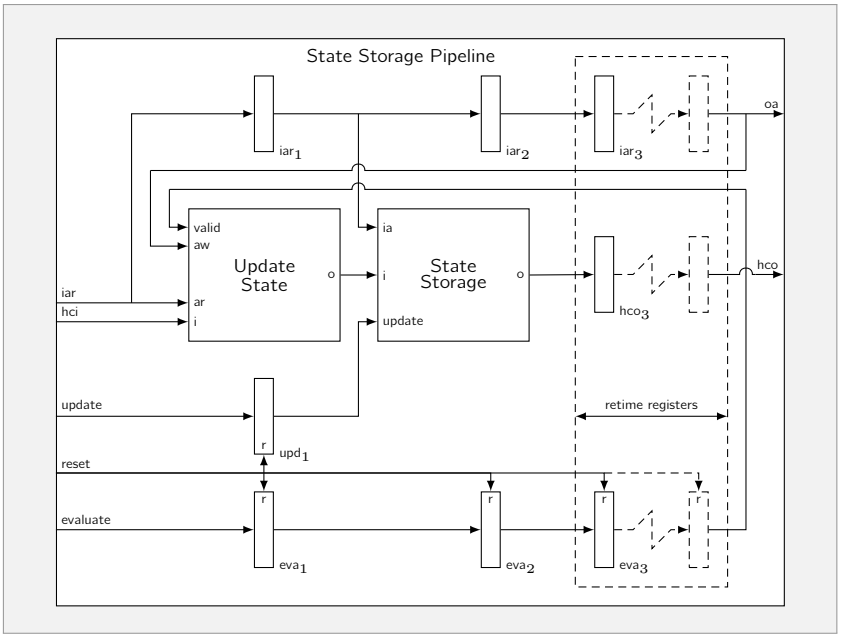


Figure 5.5: State loading pipeline

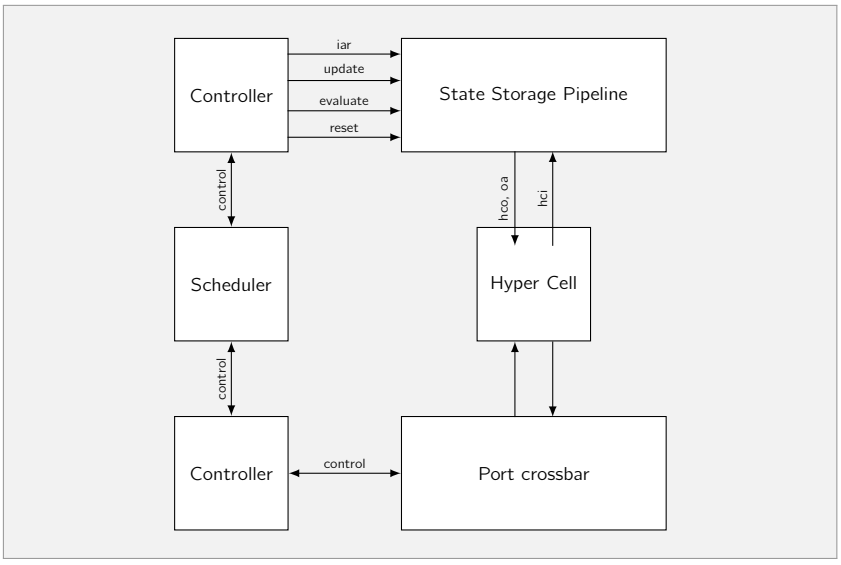


Figure 5.6: Simulator component overview

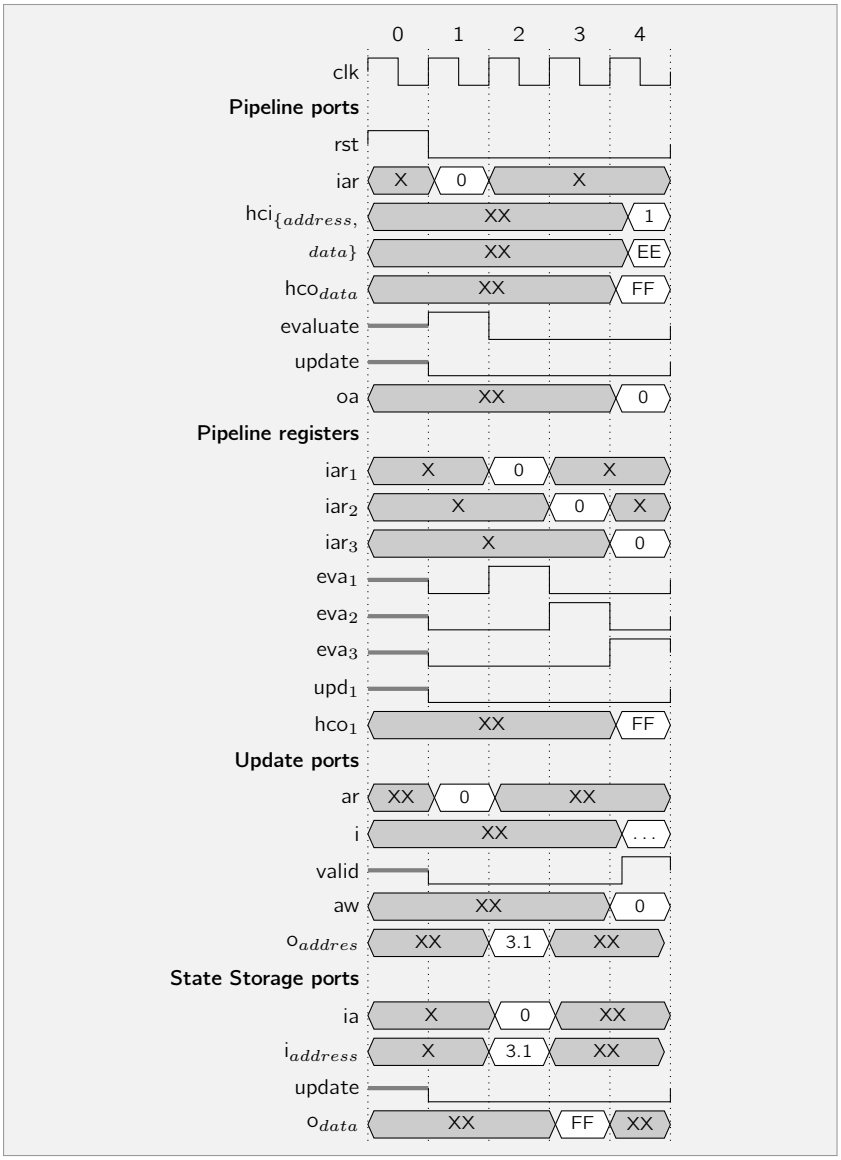


Figure 5.7: Timing diagram for state storage pipeline

the current entity, this enables the hyper cell to correctly set its own behavior, which is necessary for the integration with the new hyper cell generation by Rutgers [3]. The new state vector is set on input port HCI, which will only be saved when the valid port on the update state storage is high.

In figure 5.7 a single evaluation of the state storage pipeline is available as timing diagram. First, in clock cycle zero the pipeline is reset.

An evaluation of an entity is request in clock cycle one. Clock cycle two is used to retrieve the address, which should be read from the update state storage. This address is used in clock cycle three to retrieve the location from the memory for that entity from the state storage. Clock cycle four is used to evaluate the entity, and store the result in the update state storage. For simplicity only one evaluation is performed, but the pipeline is able to handle a new request for evaluation in each clock cycle.

Description of the ports for pipelined state loading

UPDATE If the current stored update in the output state storage has to be stored in the state storage. If this port is set to '0' then the update will only be used to determine which memory locations should be read.

IAR Input Address Read. For which entity the state should be fetched

HCI Hyper Cell Input. The new state vector, which is the output of the hyper cell, to be stored in the output state storage.

EVALUATE Fetch the state for the currently selected entity and evaluate this entity.

HCO Hyper Cell Output. The current state which was fetched.

OA Output Address. The entity the HCO belongs to.

The complete pipeline is configurable using two generics:

HYPER CELLS The number of hyper cells instantiated for this pipeline.

ENTITIES The number of entities that are simulated.

Furthermore, all types for the pipeline are generated by the extraction tool.

Chapter 6

Case study: NoC

Abstract

In this chapter we evaluate the state extraction algorithms. The main results are that the hyper cell behaves as expected, and the clock frequencies that can be achieved are comparable to the current extraction flow. Furthermore, the state storage pipeline works as expected, and the clock frequency that is necessary for the pipeline does not limit the performance of the simulator.

Outline

First, in section 6.1 the looprouter is introduced. In section 6.2 the state extraction algorithms are applied on the looprouter. The resulting hyper cell is simulated in section 6.3 to confirm that the state extraction algorithms do not break the basic functionality of the looprouter. Furthermore, the hyper cell is synthesized, and evaluated in section 6.4. The synthesis results for the state storage pipeline for the looprouter, and Montium are evaluated in section 6.5

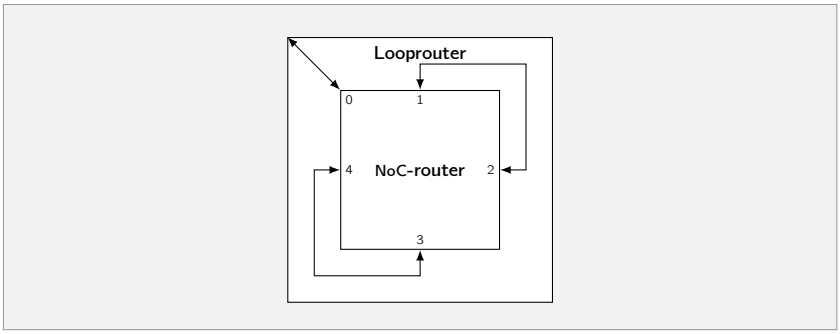


Figure 6.1: Looprouter

6.1 Looprouter

Because only the storage pipeline is analyzed, a wrapper is written around the NoC-router in order to simulate a small network. In a 2d Mesh network each router has five ports, four of them are used to connect to other routers, and the last port is used to communicate with a processing element. The wrapper connects four of the ports together so the router will route messages to itself (See figure 6.1 for a graphical view), the resulting component is now called a looprouter.

6.2 Extracted State

The state extraction algorithms, as presented in section 3, are applied to the looprouter. The only state primitives present in the intermediate edif were 120 flip-flops without a clock enable, and 3360 flip-flops with a clock enable. When the register bank algorithms are not applied to this hardware description the state vector width would be 3480.

The register bank algorithm detected five register banks. The five register banks corresponded to the fifo's in the looprouter that are placed at each input port. An interesting observation is that at each input port there are four fifo's, these four fifo's together are detected as a single register bank. Because at most one of the four fifo's for each input port would be written to, the algorithm detected that the four fifo's combined represented one register bank. But in each clock cycle This register bank was replaced by a clocked_ram primitive, and extracted as such. The five register banks represented 2800 flip-flops in the hardware design. The other flip-flops, 680 in total, are not part of a register bank, and will be extracted as flip-flops.

6.3 Looprouter test

To verify that the looprouter still works after the state extraction, a small test is performed. In a test bench several flits are supplied to this looprouter. The flits together form a packet. The first flit configures the route of the packet in the network, the flits will travel from port zero to two, one to three, and finally from four to zero. The second flit is a normal data packet, and carries a payload. The third also carries a payload, and it is also the tail of the three flits, which means that when this flit is received by a port, it can release the route for this packet.

First the original VHDL sources for the looprouter were simulated using Questasim. In figure 6.2 a waveform of that simulation is given. The inputs are used to supply the packets to the network. The outputs of the internal ports

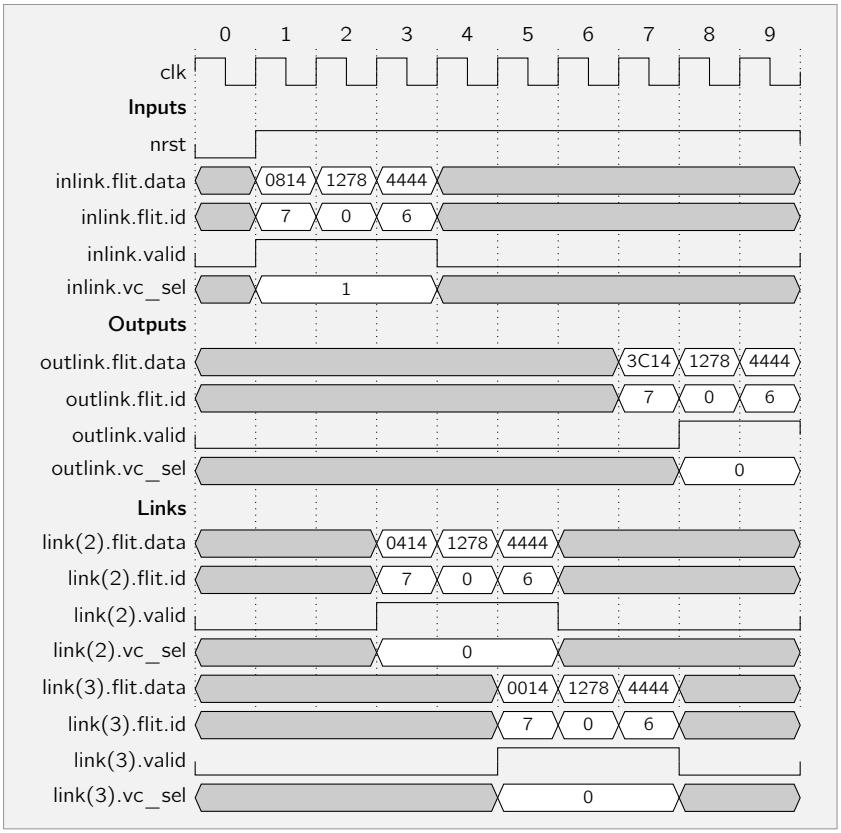


Figure 6.2: Looprouter simulation

two, and three are also shown, so that the packets can be followed through the looprouter. After some clock cycles the two payload packets are available from the outputs of the looprouter.

After this simulation the state extraction algorithms are applied to the looprouter, the results of this extraction can be found in the next section. Furthermore the state storage component for the looprouter was generated as well. In a test bench the hyper cell, and state component were simulated together. The results of this simulation confirmed that the pipeline, and state storage were able to simulate the hyper cell correctly.

6.4 Looprouter Synthesis Results

The simulator speed depends on the clock frequency on which the hyper cell, and the state storage, and some other components, which are not covered in this thesis. Several configurations of the tool flow were used to generate several hyper cells of the looprouter. In this section a comparison is made between the synthesis results of the hyper cells of the looprouter. The comparison results can be found in table 6.1. Precision was used to compile, and synthesize the hardware descriptions. A normal synthesis run, the synthesized flow by Rutgers [4], and a combination of the new algorithms and the synthesized by Rutgers have been included in the results.

Configuration A is the normal synthesis flow, the original VHDL sources have been synthesized, this gives us a base case for clock frequency, and resource usage. Configuration B uses the new flow as proposed in this thesis, but does

not do the register bank detection, and does not extract the state elements. Hence it only reads the intermediate format from Precision, and exports is directly to VHDL, after which it is completely synthesized. The results from this configuration are about the same as configuration A, the clock frequency is slightly higher, and the resource usage a slightly less. This might be explained by the fact that the hardware design is compiled twice, which could have enabled the synthesis tool to perform more optimizations. Another explanation could be that the tool does not correctly handle the graph, but the simulation in section 6.3 did run correctly, which suggest that at least the basic functions of the looprouter work. Configuration C applies the register bank detection, and extracts all state elements (the clocked_ram, and flip-flop primitives) in the new flow. The clock frequency of the hyper cell in this configuration is dramatically slow, the hyper cell can only run at 7.5 Mhz. Compared to the former configuration this is a reduction with almost a factor of eight, which is unacceptable. At this point the reason for this dramatic result was unclear. Therefore, some other configurations were tested in order to pinpoint the problem. In configuration D the register bank detection, and its extraction was applied, but the flip-flops were not extracted. Configuration E is the opposite of the former configuration, it does not apply the register bank detection, it only extracts all flip-flops in the intermediate format. Configuration D, and E both performed similar to configuration A, which suggest that separately the register bank detection, and flip-flop extraction work as expected, but that they cannot be used together in the new flow. Therefore, a hybrid flow was run, which combined the register bank detection from the new flow, and the flip-flop extraction from the synthesized flow from Rutgers [3] in configuration F. The clock frequency of the hybrid flow is higher than the original configuration A, and it only uses slightly more resources. The reason for the modest increase in resource is the register bank extraction, because all primitives that together form a register bank are replaces by a single primitive. Furthermore, a configuration is run where the hardware design was imported, and directly exported by the intermediate flow, after which the flip-flops were extracted by the synthesized flow. When compared to configuration E, which does the same extraction on the intermediate format, it is clear that this configuration is fast, but that the performance increase comes with a resource penalty. The last run, configuration H, did only run the synthesized flow. When compared to the hybrid flow, configuration F, it can be seen that the synthesized flow has a lower clock frequency, and uses almost twice as any resources.

6.5 State Storage Pipeline Synthesis Results

In the previous section the performance of the hyper cell for the looprouter was evaluated, in this section the performance of the state storage pipeline will be evaluated. The storage storage component was generated with six virtual ports on the MPRAM. The results of this evaluation can be found in table 6.2. The clock frequencies in the table represent the clock frequency of the Block RAMs, the clock frequency of the storage pipeline itself is a factor three lower.

First, the storage pipeline itself was synthesized. The clock frequency of the design was high, this is due to the fact that all pipeline stages contain not much combinatorial logic. However, the Block RAM usage was higher than expected, in this test the state storage component was generated for 16 entities. The Block RAMs necessary for the state storage itself was, as expected, 15. But the update state storage did use 32 Block RAMs, more than twice the number for the state storage. However, the update state storage uses normal Block RAMs for the update state storage. When the update state storage would also

Table 6.1: Synthesis results for looprouter

Configuration	Intermediate flow	RB extraction	Intermediate FF extraction	Synthesized FF extraction	Clock frequency before PnR (Mhz)	Cell delay (%)	Net delay (%)	Function generators	CLB-slices	Flip-flops
A						56.0	44.7	44.7	9623	4812	3480
B		•				56.8	45.5	45.5	9474	4737	3480
C		•	•	•		7.5	32.7	67.3	10075	5038	0
D		•	•			55.7	45.4	54.6	7920	3960	680
E		•		•		53.5	44.3	55.7	13892	6946	0
F		•	•		•	69.7	33.1	66.9	10513	5257	0
G		•			•	64.2	33.2	66.8	17645	8823	0
H					•	61.0	32.5	67.5	18061	9031	0

be stored in MPRAMs, it would only use 11 Block RAMs.

Second, the design was combined with the hyper cell (configuration C), and synthesized again.

The clock frequency before Place and Route (PnR) is dramatically slow. But this can be explained by how the clock is constrained, by Precision. The Precision tooling determines the relation between the input clock of the Digital Clock Manager (DCM), and the output clocks. In the storage pipeline two separate clock domains are used, but only the input port of the DCM is constrained. In the synthesized design the critical path is in the clock domain of the hyper cell, which determined the input clock frequency of the DCM. Because the tooling could not increase the critical path, it found that the clock frequency for the state storage would not be higher than 21.7 MHz. However, the synthesized design was placed and route with the Xilinx ISE tooling. This tooling did not constrain the critical path on the input clock of the DCM, but did a critical path analysis for both output clock domains. The clock frequency for the state storage pipeline is lower than the clock frequency of the state storage pipeline before Place and Route (PnR) (when it is not connected to the hyper cell), but the resulting clock frequency for the storage pipeline is still a lot higher than the clock frequency of the hyper cell, which means that it does not limit the performance of the simulator.

Because the state storage pipeline for the looprouter does not use any of the transformations as presented in section 5.1, it also did not use the additional components to support those transformation. Therefore an additional pipeline is generated that did incorporate these transformations. The design for which the pipeline is generated is the Montium, again for 16 entities. The clock frequency of the synthesized pipeline is a lot lower than the pipeline for the router, but it also used almost all Block RAMs in the design, namely 286 out of 288. Furthermore, it uses slightly more than 10% of the function generators, which leaves enough resources to implement the hyper cell, and other components in the simulator.

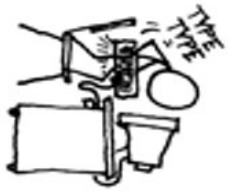
The logic that

Table 6.2: Synthesis results for state storage pipeline

	Clock frequency before PnR (Mhz)	Cell delay (%)	Net delay (%)	Clock frequency after PnR (Mhz)	Function generators	CLB-slices	Flip-flops	State storage Block RAMs	Update state Block RAMs
Router	406.5	22.1	77.9	NA	978	2163	4326	15	32
$\text{Router}_{\text{HC}}$	21.7	32.8	67.2	339.2	11061	5531	4326	15	32
Montium	87.6	47.1	52.9	NA	15199	11228	22456	192	94

Chapter 7

Conclusions



In this thesis the Sequential Hardware-in-the-Loop Simulator has been enhanced with the ability to extract large memories. This chapters will conclude this thesis with some conclusions, and further work.

7.1 Conclusions

One large obstacle for the simulation of large hardware designs in HILS has been tackled; extracting the large memories. Before the extraction algorithms were developed a detailed analysis has been performed onto which analysis level would be suitable for the memory extraction, this has been elaborated in section 3.1. This analysis provided another approach than originally developed by Wolkotte [1, 15], the intermediate format provided by Precision can be used to efficiently extract large memories.

Subsequently, two techniques to extract large memories have been introduced.

- In section 3.4 the extraction of RAM elements in the intermediate format of Precision have been elaborated.
- A Register bank extraction algorithm has been elaborated in 3.5. The algorithm has several distinct steps, it has to group flip-flops into registers, registers into register banks, and for these register banks is has to analyze if they also behave as one. When a register bank is identified it has to be replaced by an RAM element so that it can be removed from the graph.

The extracted state has to be stored in the Block RAMs of the FPGA. But how the Block RAMs are used influences the speed of the simulation. Therefore several heuristics have been developed in chapter 4. These heuristics can efficiently map the extracted memories onto the Block RAMs.

In chapter 6 an elaborate case study has been performed on a NoC-router. The case study confirmed that the memory extraction algorithms do result in a smaller hyper cell, which can be clocked at a slightly higher frequency than the hyper cell in the approach by Rutgers [3]. However not all state elements could be efficiently extracted by the new algorithms, when the flip-flops were also extracted it resulted in a dramatic decrease of the clock frequency. Hence, the conclusion that the two approaches should be combined, the large memories should be extracted with the new approach, the remaining flop-flops can be extracted by the old approach after synthesis.

7.2 Future Work

Stabilization proof extension for cycles In appendix A a proof was presented which shows that the simulator can always stabilize the system under certain assumptions. One of these assumptions was that there can be no cycles which did not contain state elements, however it might be possible to extend the proof that (under certain conditions) these cycles are permitted.

Extending Reset and Set Functionality of Register Banks In section 3.5.2 two solutions were introduced for implementing the reset and set-functionality

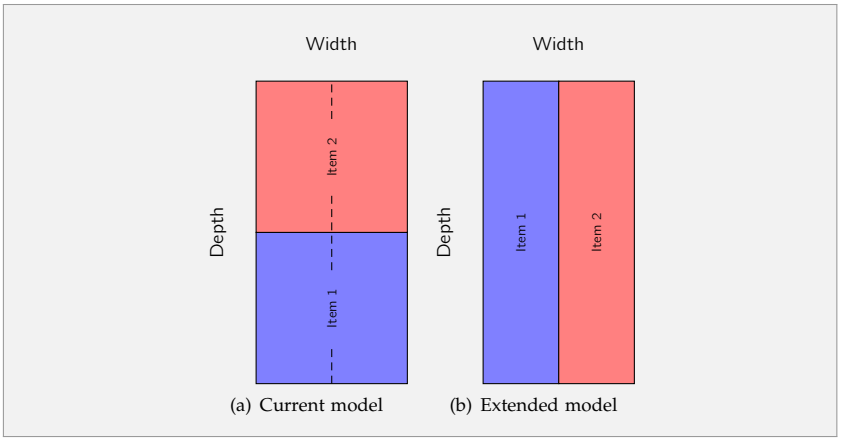


Figure 7.1: State Storage Model

of register banks. The second solution was chosen because of the simpler implementation, however from a performance perspective the first solution might be a better solution.

State storage with flexible memory normalization In section 4.3 it was stated that the memory normalization does not increase the memory requirements. This remains true, but it does increase the logic overhead needed to place more locations on one physical address of 36 bits.

In the presented model each physical location in the Block RAMs will only hold information from one memory. The model could be extended that each physical address can hold information from more than one memory.

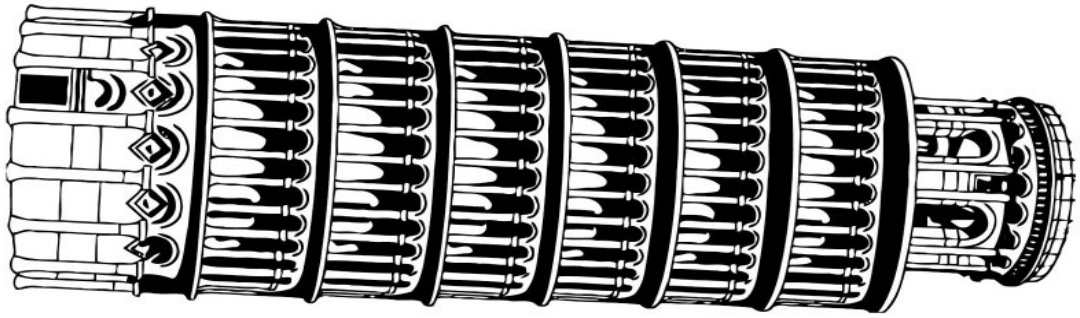
In figure 7.1(a) the current model is shown, where in one bin two memories are mapped. In the left case both memories were normalized. However, when the memories are not normalized they still occupy the same amount of memory but can be placed besides each other, saving some logic which is necessary for the normalization. Finding an efficient mapping for this extended model might take significantly more processing time, but the increase in processing time on this step might result in a faster simulator. However, this can only be done when the Block RAM properly supports that two ports write to the same location when their write enable vectors do not overlap. If this is not possible the write action of the different memories should be scheduled on different clock cycles within the MPRAM.

Extending the register bank detection Currently only register bank that only have one write port are detected. The techniques that are used to group registers, and analyze the behavior of these groups have to be extended to support multiple write ports.

Porting algorithms to synthesized EDIF Although initial analysis indicated that Precision's intermediate format is the best solution for state extractions, it could not extract all elements efficiently. At the moment the old, and new approach together remove all state elements. The state extraction algorithms could be extended so that they can also detect large memories in the synthesized EDIF, this could result in a simpler flow.

Appendix A

Proof of stabilization



The whole simulator concept works because of the assumption that a hardware design which is partitioned can be evaluated sequentially and that all will always stabilize. This chapter elaborates a proof which shows that under some basic assumption the simulator will always stabilize.

A.1 Assumptions

The proof assumes the following:

- The hardware design is described as a graph, see appendix 3.2 for a detailed description of this graph format.
- In the graph there are no cycles which do not contain a state element.
- All state elements which are not synchronous (such as latches) can be represented by another structure which is synchronous. Because all state elements are now represented by synchronous elements only the clocked edges do have to be examined. See [3] for some examples of these structures.
- The output of the state elements are stable at each clock edge.
- A primitive stabilizes when all its inputs are stable.

A.2 Proof

Let g be the graph that represents the hardware design, it contains a finite number of vertices. Let P be the set which contains all partitions. A partition $p_i \in P$ is a set primitives and all its ports.

A partition p_i contains at least one primitive:

$$\forall i \in P \bullet p_i \neq \emptyset \tag{A.1}$$

Two partition never share nodes:

$$\forall i, j \in P \bullet p_i \cap p_j = \emptyset \tag{A.2}$$

The partition is complete:

$$\bigcup_{i \in P} p_i = V_g \tag{A.3}$$

From all primitives which are state elements the outputs are disconnected from its original primitive node. A new primitive node is created for this output, hence all state elements now have a separated input and output. For the formal graph transformation, see figure A.1(a). A small example which shows a state element before and after the transformation is shown in figure A.1(b) and A.1(c).

Because the only cycles that were allowed did have a state elements in its path, and all these cycles are now broken, the graph has become acyclic. All primitives can now be marked according to the maximal length of the path

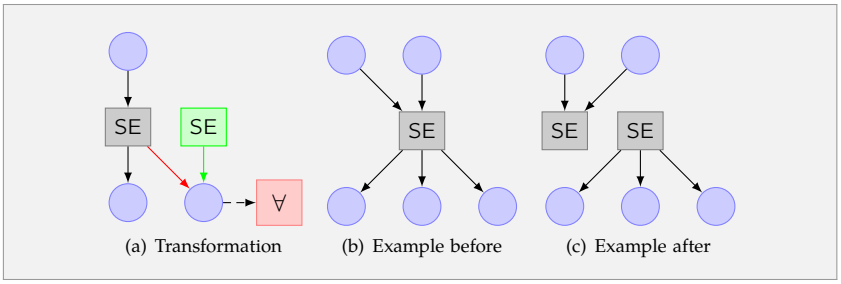


Figure A.1: State element input and output separation

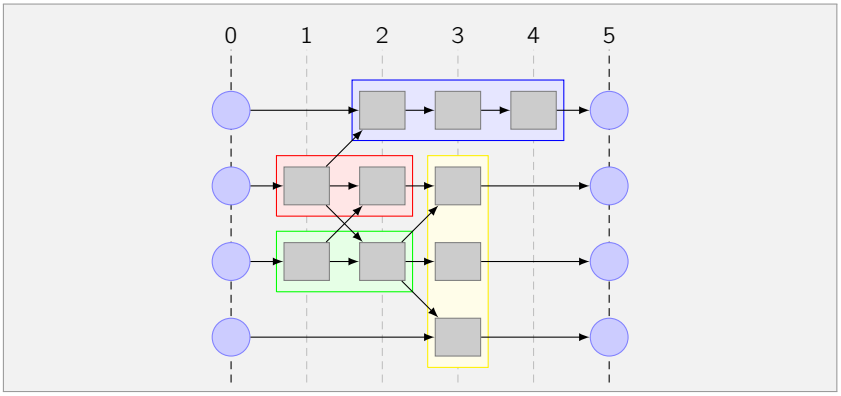


Figure A.2: Evaluation example

between an output of a state element or input port and itself. the input ports themselves are marked with a zero.

The inputs of this graph can now consists of two types of ports: the normal input ports for the complete design and the input ports which formerly were the state elements. Both are stable at clock edges, and because these are the only moments that information is stored, this is used as the starting point of this induction proof. Hence the base case is defined as:

Base case All the inputs of the graph are stable, which implies that all nodes with are marked with 0 are stable. In the worst case scenario all signals within the graph are unstable.

Induction step When all primitives at a certain level are stable all inputs for the next level are stable. Therefore when all partitions are evaluated once the next level also because stable. Hence each time all partitions are evaluated once, the levels that are stable is increased by one, and because the number of levels is limited all partitions eventually become stable.

Final case When the last level of primitives is stable the complete hardware design is stable.

A.3 Evaluation example

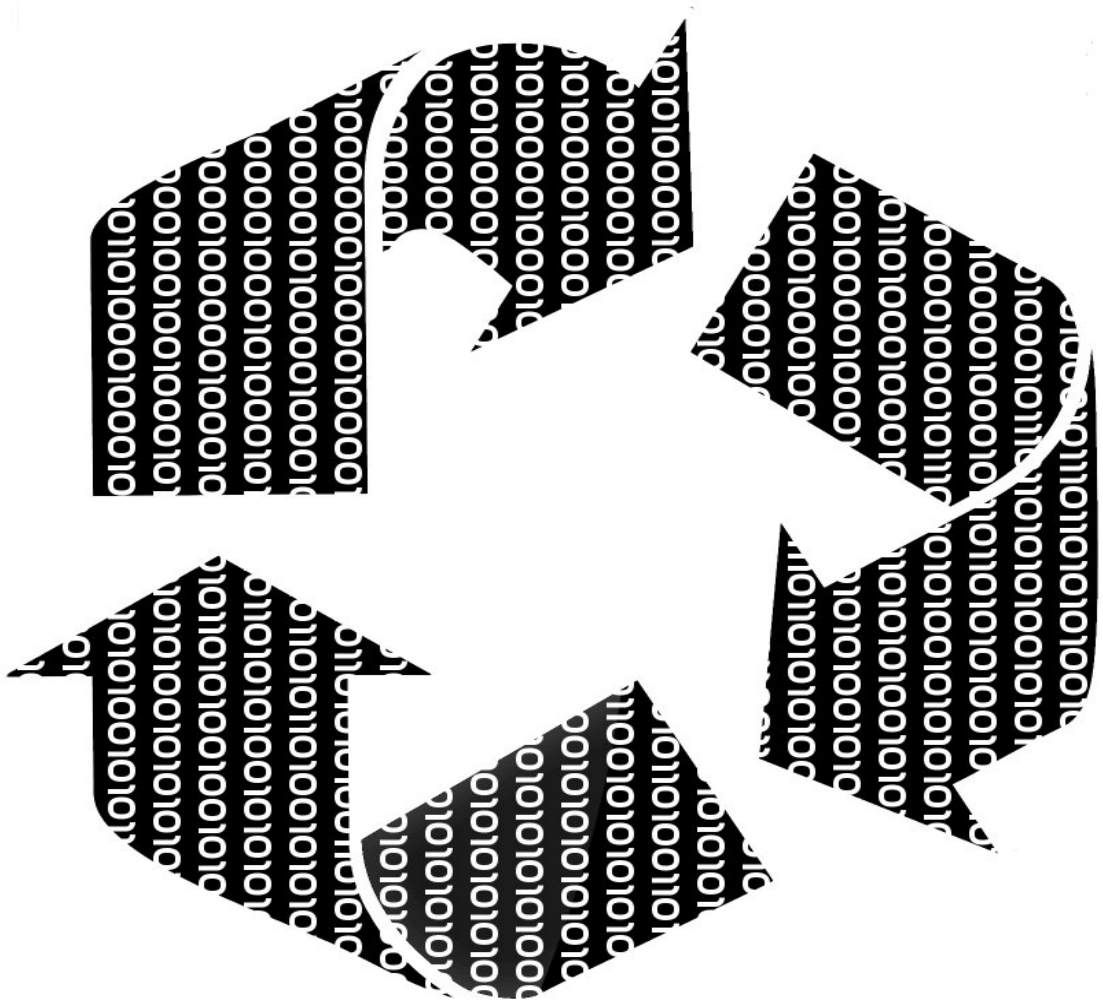
See figure A.2 for the graphical representation of the graph used in this example, not that only the primitives are shown. The partitions are represented by the blue, red, green and yellow boxes.

In the base case only the input at level 0 are stable. Then all partitions are evaluated once, and level 1 becomes stable. Again all partitions are evaluated once, and level 2 becomes stable. This continues until level 5 becomes stable.

Hence in the worst case 16 evaluation were necessary to reach a stable system.

Appendix B

Precision's Import, and Export



B.1 Formats

As mentioned in chapter, 3 Precision's intermediate format is chosen as the level to perform the analysis and extraction of memory and register banks on. How this level was exported and imported is not described in that chapter. This will be elaborated in this chapter. First, it is examined how the netlist will be exported. Than how the hardware is represented within the tooling. the chapter ends with a description how the internal representation is imported in Precision.

B.1.1 Export format

Precision's intermediate format can be exported in several formats, three of which contain the hardware design, and are readable. These are EDIF, VHDL and Verilog. In this section these three formats are evaluated and one of them is chosen as the export format.

Correctness Determining if any hardware designs will always be compiled correct and exported is not feasible. However, with a few small designs it should be possible to demonstrate if the general idea is sound. For this test a small design (the NoC router) is compiled, and exported to each of the three file formats. These files are compiled again to see if precision is able to correctly interpret its own intermediate format.

Precision could not compile the VHDL file. It reported errors while compiling some ranges. The ranges did not adhere to the VHDL synthesis standard.

The Verilog and EDIF file did compile correctly. However the resulting hardware was not complete, in the intermediate stage there were inferred black boxes. It seems that some of the components that are used in the intermediate stage are not recognized by Precision's compilation phase. With an even smaller test design (an incrementor) it was found that also the VHDL file did also infer these black boxes.

Complexity As simple solutions are generally preferred it is determined how complex it would be to parse and analyze the file formats.

Because the exported file is a netlist only a small subset of language constructs is used for the VHDL and Verilog formats. An open source parser is available for the VHDL language [30], this parser was also used for the vhdl memory extraction. For Verilog there is also an open source parser [41].

The EDIF language was developed as a standard for data exchange between different tools and is able to store a netlist in an tree structure. The original tooling for this project uses EDIF for its input format. Only small changes are necessary to adapt this tooling to correctly read the intermediate output EDIF.

B.1.2 Import formats

Complexity Generating a netlist in all three file formats is relatively simple.

Operator replacement

Table B.1: Comparison of VHDL, Verilog and EDIF import format

Feature	VHDL	Verilog	EDIF
Complexity	+	+	+
Operator replacement	+	+	-

VHDL Operators can be implemented as standard VHDL components. A behavioral description suffices. Hence the compiler could generate almost the same representation for the component.

Verilog Same as VHDL

EDIF Here it is not possible to use VHDL or Verilog components to replace the black boxes. The black boxes could be compiled as either EDIF or Ngc and than be imported. However, this is a major disadvantage because we than choose a specific implementation for that black box. But since we know nothing of the paths leading into and out this components we might make a bad choice in regard to timing and/or area.

B.1.3 Evaluation

Because of the existing tooling and lack of errors, EDIF is chosen as the export format. However, the EDIF cannot be used as import format, because the resulting hardware will be instantiated with black box operators. The choice between VHDL and Verilog is trivial, VHDL was chosen due to the lack of experience with Verilog.

B.2 Netlist Size

The export VHDL netlist has to be compiled, and synthesized. Precision will not interpret the VHDL file as a netlist, and will compile the design normally. This proved to be a bottleneck. Within the tool flow the netlist was flattened. The compiler of Precision compiles each design into pieces, each of these pieces represent a component in the design. Our flattened design consists of only one component, which resulting into several memory errors during the compilation. Therefore, the hierarchy had to be preserved for large components, this has been included in the formal graph representation (See section 3.2).

Appendix C

Implementation

In this appendix the implementation of the tool will be elaborated.

C.1 Java Implementation

In this section the Java implementation of the software will be elaborated.

C.1.1 Java Classes

GraphTransformerTask This is the main class of the state extraction. The `run` functions is responsible for importing the hardware graph, starting several state extraction algorithms, running the extraction and exporting the resulting hardware to VHDL.

HardwareGraph The hardware graph is represented in this graph. This class holds all the different components and where these components are placed within the hierarchy of the hardware. Several functions allow the splitting and combining of these components.

ComponentGraph This class actually holds the graph which represents (a portion) of the hardware.

DirectedGraph This class is the graph itself. It comes from a Graph package for Java [42].

CellNode The components in the graph are represented by this class.

PortNode The ports of a component in the graph are represented by this class.

GraphOptimizer Several small optimizations are performed by this class, namely: Buffer components are removed from the graph. All the `TRUE` and `FALSE` components are merged. These operations reduces the size of the graph.

RegisterDetector This class is responsible for the discovery of flip-flops which might be grouped together into a register.

RegisterMerger The set of flip-flops discovered by the *RegisterDetector* class are checked if they really behave as a register. When these flip-flops behave as a register they are merged into a register.

RegisterBankDetector This class is responsible for the discovery of registers which might be grouped together into a register bank.

RegisterBankMerger The register bank candidates provided by the *RegisterBankDetector* are analyzed to check if they behave as a register bank. If this analysis confirm the behavior of the register bank the registers are grouped together as a *clocked ram* component and some logic is instantiated around the clocked ram to glue it to the existing hardware.

RegisterBank This class is used to store some information about detected register banks. It is primarily used by the *RegisterBankDetector* to relay information to the *RegisterBankMerger* class.

HardwareGraphExtractor This class is responsible for the actual extraction of the state components. Classes which are extended from the *NodeExtractor* class are used to implement the extraction for several types of components. When a component is extracted a child form the *Component* class is used to store information about the extraction.

NodeExtractor This is an abstract class which is used by the *HardwareGraphExtractor* class to implement the extraction of components in the graph.

ClockedRamExtractor Child of *NodeExtractor*.

DFFEExtractor Child of *NodeExtractor*.

DFFExtractor Child of *NodeExtractor*.

RegisterExtractor Child of *NodeExtractor*.

Component Abstract class which is used by the *HardwareGraphExtractor* class to store information about the extractions.

ClockedRam Child of *Component*

DFFE Child of *Component*

Register Child of *Component*

ComponentVhdlWriter This class is used to export a *ComponentGraph* to a VHDL file.

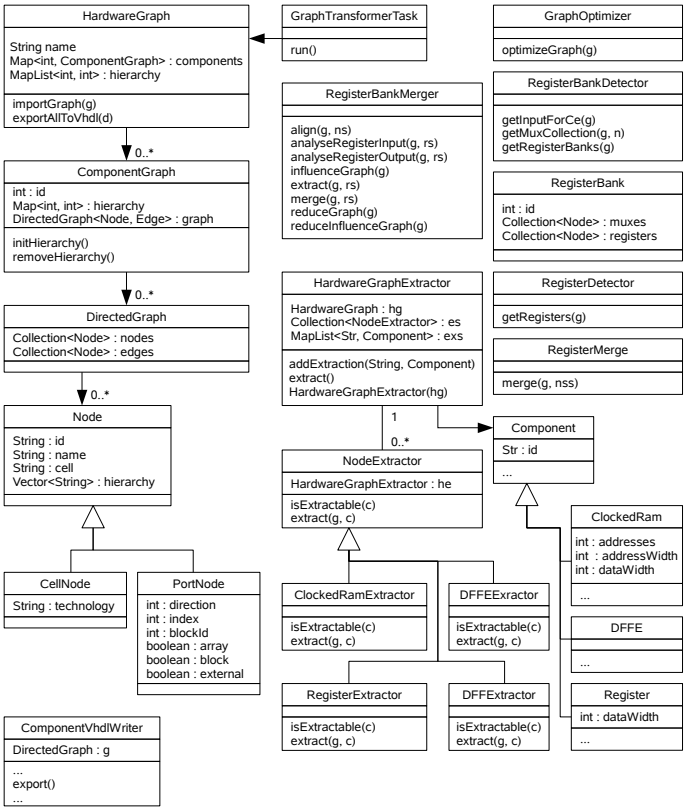


Figure C.1: Memory extraction class diagram

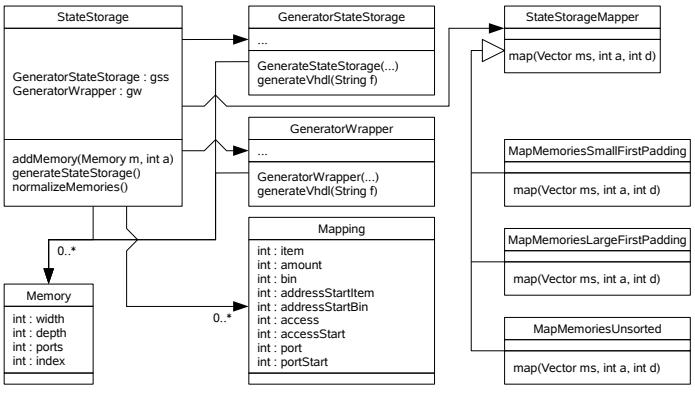


Figure C.2: State storage class diagram

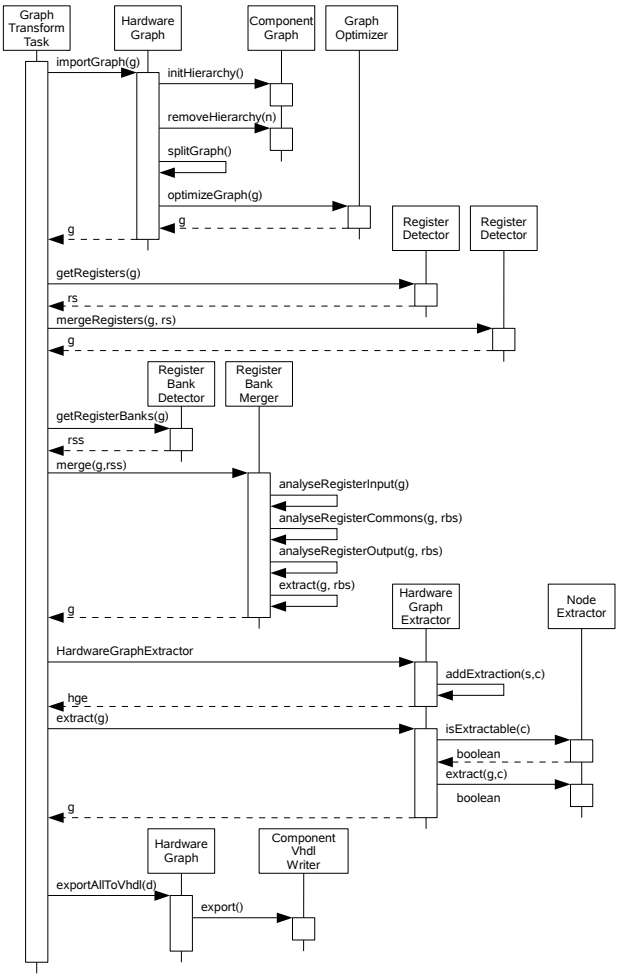


Figure C.3: Memory extraction sequence diagram

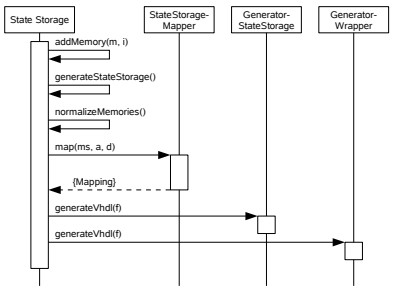


Figure C.4: State storage sequence diagram

Bibliography

[1] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Using an fpga for fast bit accurate soc simulation. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS'07) - 14th Reconfigurable Architecture Workshop (RAW 2007), Long Beach, CA, USA*. IEEE Computer Society Press, March 2007.

[2] Paul M. Heysters. *Coarse-Grained Reconfigurable Processors*. PhD thesis, University of Twente, Enschede, The Netherlands, September 2004.

[3] J.H. Rutgers. Automated sequential hardware-in-the-loop simulator generation. Multi-disciplinary Design Project, November 2007.

[4] J.H. Rutgers. Advanced automated sequential hardware-in-the-loop simulator generation. Master's thesis, University of Twente, 2008.

[5] Federico Faggin, Marcian E. Hoff Jr., Stanley Mazor, and Masatoshi Shima. The history of the 4004. *IEEE Micro*, 16(6):10–20, 1996.

[6] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[7] G.E. Moore. Progress in digital integrated electronics. *Electron Devices Meeting, 1975 International*, 21:11–13, 1975.

[8] Excerpts from a conversation with gordon moore: Moore's law, 2005.

[9] Nvidia's 1.4 billion transistor gpu. `http://www.anandtech.com/video/showdoc.aspx?i=3334&p=1`.

[10] World's first 2-billion transistor microprocessor. `http://www.intel.com/technology/architecture-silicon/2billion.htm`, 2008.

[11] The cell project at ibm research. `http://www.research.ibm.com/cell/`.

[12] Intel Corporation. Tera-scale computing research program. http://techresearch.intel.com/articles/Tera-Scale/1421.htm.

[13] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, Feb. 2007.

[14] Gerard J.M. Smit, André B.J. Kokkeler, Pascal T. Wolkotte, and Marcel D. van de Burgwal. Multi-core architectures and streaming applications. In *SLIP '08: Proceedings of the 2008 international workshop on System level interconnect prediction*, pages 35–42, New York, NY, USA, 2008. ACM.

[15] P. T. Wolkotte, P. K. F. Hölzenspies, and G. J. M. Smit. Fast, accurate and detailed noc simulations. In *Proceedings of the 1st ACM/IEEE International Symposium on Networks-on-Chip, Princeton, NJ, USA*. IEEE Computer Society Press, May 2007.

[16] Ieee std 1666 - 2005 ieee standard systemc language reference manual. *IEEE Std 1666-2005*, 2006.

[17] Modelsim. http://www.model.com.

[18] M.N. Wageeh, A.M. Wahba, A.M. Salem, and M.A. Sheirah. Fpga based accelerator for functional simulation. *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, 5:V–317–V–320 Vol.5, May 2004.

[19] J. Babb, R. Tessier, M. Dahl, S.Z. Hanono, D.M. Hoki, and A. Agarwal. Logic emulation with virtual wires. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(6):609–626, Jun 1997.

[20] Veloce mhz-class accelerator/emulator. http://www.mentor.com/products/fv/emulation-systems/veloce, 2009.

[21] V. Goncalves, J.T. de Sousa, and F. Goncalves. A low-cost scalable pipelined reconfigurable architecture for simulation of digital circuits. *Field Programmable Logic and Applications, 2005. International Conference on*, pages 481–486, Aug. 2005.

[22] Srihari Cadambi, Chandra S Mulpuri, and Pranav N Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 570–575, New York, NY, USA, 2002. ACM.

[23] P.T. Wolkotte. *Exploration within the Network-on-Chip Paradigm*. PhD thesis, University of Twente, Enschede, The Netherlands, 2008.

[24] N. Kavaldjiev, G.J.M. Smit, P.G. Jansen, and P.T. Wolkotte. A virtual channel network-on-chip for gt and be traffic. *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, 00:6 pp.–, March 2006.

[25] N. K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. PhD thesis, University of Twente, 2006.

[26] Netlist. http://en.wikipedia.org/wiki/Netlist, March 2009.

[27] Synopsys design compiler. http://www.synopsys.com.

[28] Mentor graphics precision rtl. http://www.mentor.com.

[29] John Crawford. Edif: A mechanism for the exchange of design information. *IEEE Design and Test of Computers*, 2(1):63–69, 1985.

[30] Signs - vhdl hardware developement. http://www.iti.uni-stuttgart.de/~bartscgr/signs/wiki/index.php/Main_Page.

[31] Groove, graphs for object-oriented verification. http://groove.cs.utwente.nl/.

[32] Cosy compiler development system, ace associated computer experts. http://www.ace.nl/compiler/cosy.html.

[33] Stephen A. Cook. The complexity of theorem-proving procedures. *Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[34] Benjamin Biron. Boolean expression reducer java package. http://sourceforge.net/projects/bexpred, December 2003. BExpred v0.8.

[35] W.V. Quine. The problem of simplifying truth functions. *The American mathemetical Monthly*, 59, Oct. 1952.

[36] Olivier Coudert. Doing two-level logic minimization 100 times faster. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 112–121, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[37] Z. Arevalo and J.G. Bredeson. A method to simplify a boolean function into a near minimal sum-of-products for programmable logic arrays. *Computers, IEEE Transactions on*, C-27(11):1028–1039, Nov. 1978.

[38] Virtex-5 family overview, 2009. Product Specification.

[39] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1, 1981.

[40] Ilog cplex: High-performance software for mathematical programming and optimization. `http://www.ilog.com/products/cplex/`.

[41] Karl W. Pfalzer. v2kparse - open source verilog 200x parser. http://v2kparse.sourceforge.net.

[42] Jgrapht - a free java graph library. `http://jgrapht.sourceforge.net/`.