Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

7th August 2017

Abstract

The Synchronous Message Exchange (SME) model, is a programming model, which closely resembles the CSP model and which is suitable for describing hardware. This thesis aims to combine the theory taught in a machine architecture class with the SME model, by implementing a MIPS processor using SME. I show how to construct the components of a MIPS processor as SME processes, and how to connect them by using SME busses. Furthermore, I show how to extend the processor, by introducing additional instructions and by pipelining the processor. Finally, I succesfully implement the Single Cycle and Pipelined MIPS processors onto an FPGA.

Thesis supervisors: Brian Vinter and Kenneth Skovhede.

Acknowledgements

I would like to acknowledge my thesis supervisors Brian Vinter and Kenneth Skovhede for guiding me through writing my thesis and through writing the corresponding paper [1]. I would also like to thank the employees in the eScience group at the Niels Bohr Institute, for taking their time to listen to my project and give me feedback on my progress. Finally, thanks to Daniel Lundberg Pedersen and Jeanette Johnsen for proof reading.

Contents

1 Introduction

In the Machine Architecture class (ARK) [2] at the Department of Computer Science at the University of Copenhagen (DIKU), the theory of computer organization and design is taught. The course teaches how to construct combinatorial circuits, which were later used to implement the components of a MIPS (Microprocessor without Interlocked Pipeline Stages) processor [3], and how to connect the components. However, while this course focused on the theory, it did not focus on how to construct specialized hardware, as one might implement on an FPGA (Field Programmable Gate Array).

In some applications, FPGAs are more attractive than general purpose CPUs, as they do not necessarily have the same overhead in both performance and power usage. This is due to the FPGA being more specialized towards a specific task, where the CPU is much more complex, and as such requires additional circuitry [4].

However, the biggest problem with FPGAs is that they are programmed using Hardware Description Languages, such as VHDL (Very high speed integrated circuit Hardware Description Language) and Verilog. These languages are largely unchanged from their initial design, are very tedious to work with and lack many of the features of modern programming languages [4].

This has changed with Synchronous Message Exchange (SME) [4, 5, 6]. SME is a programming model, which is similar to the Communicating Sequential Processes (CSP) model [7]. Both models compute their results, by having multiple processes that communicate with each other. The key differences are that SME is globally synchronous, has broadcasting channels and a hidden clock. As such, SME is more suitable for developing hardware models than CSP. Furthermore, SME can be transpiled into VHDL [8], as the structure of hardware described in VHDL is very similar to the hardware described in SME. However, SME is simpler to implement and verify than VHDL. As such, SME gives an approachable model for software developers, who are already familiar with the CSP model, for generating hardware models.

Contribution

In this thesis, I will combine the theory from the ARK course with the C# version of the SME programming model [4]. As such, I will implement a MIPS processor in SME, which can be transpiled into VHDL, and be further synthesized onto an FPGA.

I will start by implementing some basic combinatorial circuits in SME. Then I will combine these circuits into the basic components of the MIPS processor. Then I will combine these components into a sincle cycle MIPS processor and pipeline the processor, while handling data hazards and control hazards. Finally, I go through the procedure of transpiling SME into VHDL, and how to synhesize, place and route the transpiled VHDL onto an FPGA.

The material in this thesis could be used for teaching software developers, who already know the CSP programming model and hardware organization, how to construct specialized hardware.

The contents in sections 2, 3, 4 and 5 have been published [1] to the Communicating Processes Architecture conference [9].

Other projects

There are several other projects, which implement a MIPS processor on an FPGA. However, they are all written in hardware description languages VHDL and Verilog. As such, they are closer to hardware than software and are more difficult to understand with a background in software development. One such project is the MIPSfpga project [10], which is implemented in Verilog.

There are additional projects, which gives different approaches to hardware description languages. One of the projects is Pyrope [11]. Pyrope is a modern hardware description language for live synthesis flow. I have not used Pyrope enough to compare it to SME, but like SME, Pyrope can be transpiled into Verilog with a testbench. As such, it should also provide a better approach for software developers for generating hardware.

2 Basic logic circuits

In this section, I will be looking at some basic combinatorial circuits. I use this as an entry point for hardware design, both due to the ARK course [2] at DIKU, and due to the architecture book [3] recommending to read appendix C, which covers the basics of logic design, prior to reading chapter 4, which covers implementing a simple MIPS processor. I am only going to cover a subset of appendix C. It should be sufficient as an introduction to connecting simpler components together into a more complex network. It should also be sufficient as an introduction on going from CSP [7] to SME [5, 6, 4].

I start by looking at some logic gates, which implement some basic boolean functions. Then I will combine these basic gates into more complex networks:

- A decoder, which expands an n-bit input into 2^n outputs.
- A half adder, which takes two binary inputs and computes the sum and the carry of the two.
- A full adder, which does the same operation as the half adder, but with an additional third binary input.
- A n-bit adder, by combining a chain of a single half adder followed by $n - 1$ full adders.

For each of these combinatorial circuits, I go through the theory behind it and describe the procedure of translating the theory into an SME. All of the networks follow the same procedure for testing: I construct an SME process, which sends data on the input busses, and verifies that the data on the output busses is as expected. Since these networks are very small, I can, in most cases, test for every possible input. The reason for using this testing technique is that I can specify the behavior of the processes before constructing them, and as such don't care how they compute their result, focussing on the correctness of the result.

2.1 Basic logic gates

A logic gate is a circuit abstraction, which has inputs and outputs and computes the logic function that corresponds to the gates name, i.e. its output values are

Bit1	Bit2	AND	OR	NOT	XOR
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Table 1: The truth table for the four basic logic gates. Note: `NOT` is only considering `Bit1`.

```
1  public class AND : SimpleProcess
2  {
3      [InputBus]
4      Input input;
5
6      [OutputBus]
7      Output output;
8
9      protected override void OnTick()
10     {
11         output.And = input.bit1 && input.bit2;
12     }
13 }
```

Listing 1: The SME process for the `AND` gate.

based upon the input values. I am going to implement the following logic gates:

`AND` - outputs `1` iff all of its inputs are `1`, otherwise it outputs `0`.

`OR` - outputs `1` iff one or more of its inputs are `1`, otherwise it outputs `0`.

`NOT` - outputs the inverse of its input, i.e. `1` becomes `0` and `0` becomes `1`.

`XOR` - outputs `1` iff exactly one of its inputs are `1`, otherwise it outputs `0`

The full truth table for all of the four logic gates with 2-bit input (`NOT` only looks at Bit1) can be seen in Table 1.

Implementation [1]

Implementing each of these four logic gates is straightforward: There is an input bus with two 1-bit values, a process for each of the gates and an output bus with a 1-bit value for each of the logic gates. I do not need additional input busses, as each process which uses the bus as input receives its own copy of the input in each clock, due to the broadcasting in SME [5]. Furthermore, I only need one output bus, as each process writes to its own bit within it. Each process takes the two bits from the input bus, computes its respective logical function and sends the result out onto its bit on the output bus. The code for the `AND` gate can be seen in Listing 1.

I could have made a more complex process (i.e. one that takes the two inputs, and computes each function). This would reduce the amount of bus connections,

[1] The source code for the Logic Gates is available at [12] at `sme/src/Examples/LogicGates/`

```
1 public static void Main(string[] args)
2 {
3     new Simulation()
4         .BuildCSVFile()
5         .BuildVHDL()
6         .Run(typeof(MainClass).Assembly);
7 }
```

Listing 2: The `Main` function for running an SME simulation, generating the CSV trace file and generating the VHDL code.

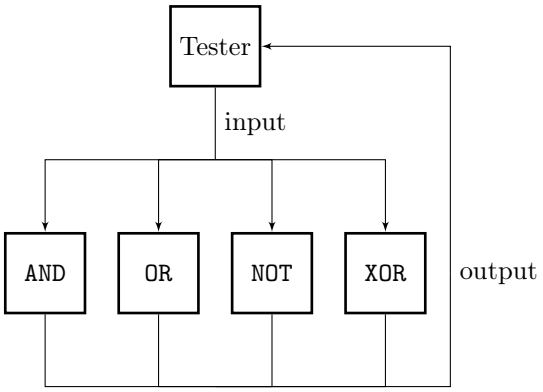

Figure 1: The structure of the test of the Logic Gates.

but I want to make each process as simple as possible when constructing the processor. This is closer to the CSP model, and makes arguing for correctness simpler. How the logic gate processes and the tester process are connected can be seen in Figure 1. How to run the simulation and generate the VHDL code can be seen in Listing 2.

2.2 Decoder

A decoder is a component, which takes an n-bit input, and produces an 2^n-bit output, where the bit corresponding to the input numbers binary representation is set to `1`. E.g. if the input value is the binary representation of the number 5, then the 5th output bit will be `1`, and the rest will be `0`. Note: the output bits are zero-based, i.e. there is also an output bit for the binary representation of 0.

A decoder can be made from a set of `NOT` and `AND` gates [13]. I need to have n `NOT` gates and 2^n `AND` gates. I duplicate each input and connect one copy to a `NOT` gate. Then for each output I attach an `AND` gate and give it inputs corresponding to the binary representation of the number. E.g. for the number 5, binary representation 101, the 5th `AND` gate gets input from Bit0, `NOT` Bit1 and Bit2. The network of a 2-bit decoder can be seen in Figure 2.

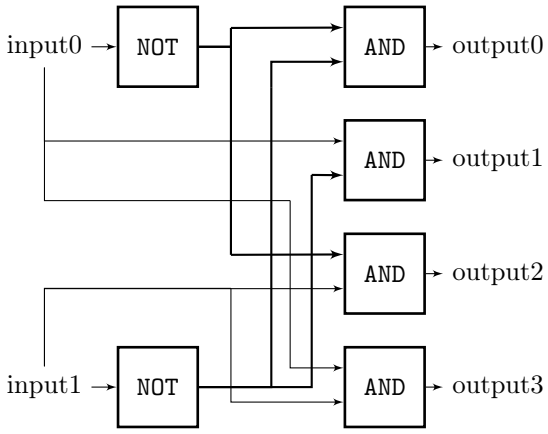

Figure 2: A 2-bit decoder made by `AND` and `NOT` gates.

Implementation [2]

To implement a 2-bit decoder, I just need to connect logic gate processes, which I have already constructed in Section 2.1. How to connect a 2-bit decoder can be seen in Figure 2. I could have made the decoder a single process. However, I am trying to emphasize connecting multiple processes together, in order to gain functionality and to be closer to the CSP model.

However, making a scalable decoder is not trivial, as SME requires everything to be known at compile time, and I cannot make a generic process, as these depend on the names of the different busses. To solve this problem, I can use C# templates to generate SME code. I chose C# templates, as I was already using the C# version of SME, and as such a C# template was easy to add. Any code generator would have been equally good. For each input bit, I create an input bus. Then, for each input bus, I create an `NOT` gate process, which takes the input bus corresponding to its index. Then, I create 2^n output busses, and for each of these, I connect an `AND` gate with its corresponding output bus. Finally, for each of these `AND` gates, I connect the busses whose logical `AND` will produce a `1`. E.g. for `output0`, I connect all the busses from all of the `NOT` gates. For `output1`, I connect all the `NOT` gates, except from the bus from the first `NOT` gate, as this should be the first input bus.

In order to test the n-bit decoder, I also needed to construct the tester process by using C# templates, as the tester process also needed a variable amount of busses.

2.3 Adder

An adder is a component, which adds two binary numbers together. As with the decoder, an adder can be constructed by a combination of `AND`, `OR` and `XOR` gates [13]. An n-bit adder is a chain of two major components: a half adder and $n-1$ full adders [3].

The half adder is the initial component in the chain. It takes two binary

[2] The source code for the decoder and the scalable decoder is available at [12] in `sme/src/Examples/Decoder/` and `sme/src/Examples/ScalDecoder/`

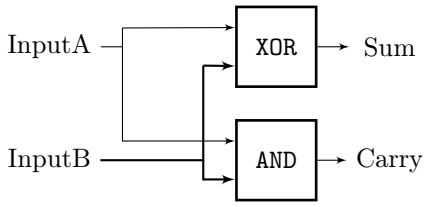

Figure 3: An half adder composed of `XOR` and `AND` gates.

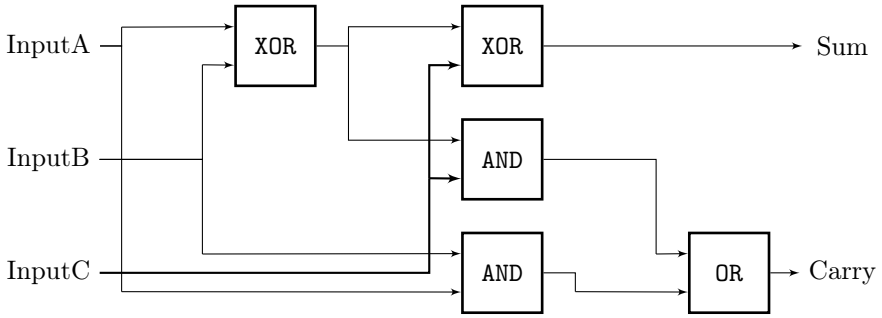

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates.

inputs, and outputs the sum and the carry of the addition (See Figure 3). The rest of the n-bit adder consists of a chain of $n-1$ full adders that take three inputs A, B, and the carry from the previous link in the chain, and outputs the sum and the carry of the addition (See Figure 4). The combination of the components can be seen in Figure 5.

Implementation [3]

The half adder and the full adder are made like the decoder, in which I have the basic logic gate processes and connect them as specified in Figure 3 and Figure 4.

To make the n-bit adder, I have to use C# templates like I did when constructing the n-bit decoder. I program it so that each of the input bits has its own bus, each of the output sums has its own bus, and each of the carries has its own bus. I start by making a half adder, which has the inputs: input A bit 0 and input B bit 0. The initial half adder outputs its sum on sum 0, and outputs the carry on carry 0. Then I construct $n-1$ full adders, where full adder i (1-based) has the inputs: input A bit i, input B bit i and carry $i-1$. Each full adder has sum i and carry i as output.

3 Core components

In this section, I describe the major components of the MIPS processor. I look at the components before wiring them together into a processor, as it keeps each

[3]The source code for the half adder, full adder and n-bit adder is available at [12] in `sme/src/Examples/Adder/`, `sme/src/Examples/FullAdder/` and `sme/src/Examples/Adder32/`

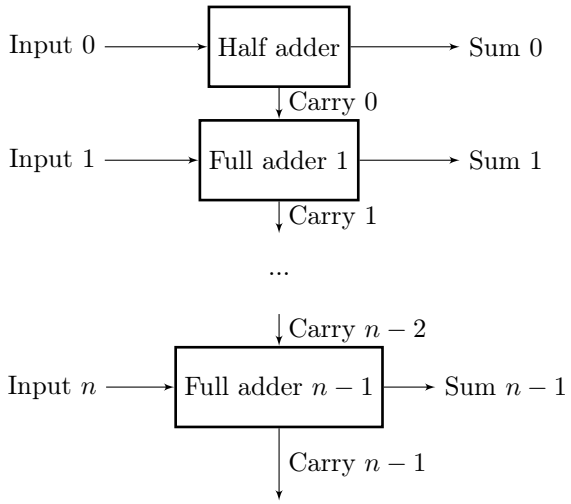

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

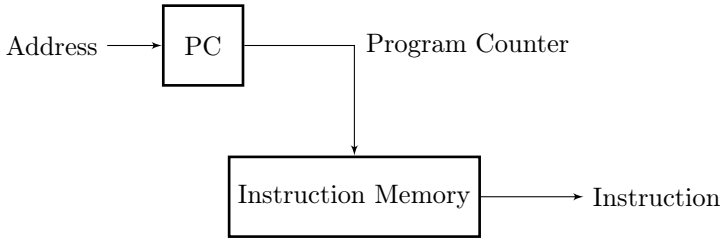

Figure 6: The Instruction Memory.

step more isolated and simple. The source code for each of the components is available at [12] in `sme/src/Examples/SingleCycleMIPS/`.

For each component, I will go through the theory of the component, then I will look at translating theory into an SME implementation. To verify the implementation, I construct a tester process, just as with the basic logic circuits. The order in which the components are mentioned is derived from chapter 4.3 in the architecture book [2], and should be the order of implementation.

By the end of this section, I should have all of the required resources for building the single cycle MIPS processor using SME.

3.1 Instruction Memory

The first component that I need is a memory unit to hold the instructions of a program, and to supply instructions at a given address. I will also need a register holding the current address in the program called the Program Counter (PC). The memory unit has one input: the address from the PC, and one output: the read instruction. The PC has one input: the input address. The Instruction Memory, the PC and their connections can be seen in Figure 6.

Implementation

Both the PC and the Instruction Memory should be SME processes. I could have combined the two components into a single SME process, but as mentoined before, I am trying to be closer to the CSP model. The input bus for the PC, the Program Counter bus and the instruction bus, should all contain a `uint` value.

The PC should have a single `uint` value holding the current address. It should only output its stored value, when the clock ticks. This can be expressed in SME, by giving a process the `ClockedProcess` attribute. This ensures that the process is no longer combinatorial, but only performs its operation when the clock ticks.

The Instruction Memory should have an `byte` array, as this will make indexing into the array simpler. Usually, the Instruction Memory and the Memory Unit share the same address space. However, for simplicity, I give each component its own memory. On each tick, the Instruction Memory should take the input address, read the `byte` at the given index and the following 3 `byte`s, and finally pack all 4 `byte`s together into an `uint`, and place it on the instruction bus.

I could have had the memory as an `uint` array, but then I would have to divide the address by 4, since the memory in the MIPS processor are byte addressed. I could modify the processor to deal with addresses in an indexing manner, however this would make programs harder to translate, as the instructions in a MIPS binary can have absolute addresses.

3.2 Register file

The MIPS processor has 32 general-purpose 32-bit registers, which are stored in a structure called the Register File. Each of the registers can be read from or written to, except for register zero, which is immutable and always 0. It is the first level in a memory hierarchy, and is thus the fastest memory available. The registers are divided into groups based on their usage, but this does not matter from a hardware perspective.

The Register File has 5 inputs: Read Register A, Read Register B, Write Enabled (`RegWrite`), Write Address and Write Data. The Register File has two outputs: Output A and Output B. There are two stages of the Register File: reading and writing. In the reading stage, it takes the value in the register at the address of Read Register A, and outputs it on Output A, and vice versa for Read Register B and Output B. In the writing stage, if the Write Enabled flag is set, it takes the value from the Write Data bus, and stores it in the register with the address given in the Write Address bus.

I need to be careful of the order in which I read and write from the Register File. I need to make sure that when an instruction reads from the Register File, it always gets the latest data (i.e. if an instruction reads from the same register as a previous instruction writes to, it should read the value written by the previous instruction). This is easy to fix in the single cycle processor, as the register file should just write before reading. It should not be in the reverse order, as the Register File might then output old values. The Register File and its connections can be seen in Figure 7.

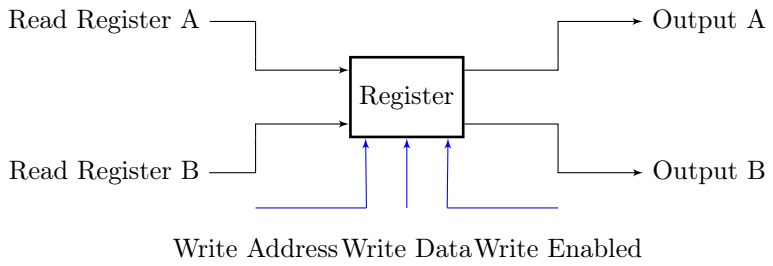

Figure 7: The register file.

```
1  public class Register : SimpleProcess
2  {
3      ...
4
5      uint[] data = new uint[32];
6
7      protected override void OnTick()
8      {
9          if (regWrite.flg && writeAddr.val > 0)
10             data[writeAddr.val] = writeData.data;
11         outputA.data = data[readA.addr];
12         outputB.data = data[readB.addr];
13     }
14 }
```

Listing 3: The SME process for the register file.

Implementation

The Register File should be an SME process. The collection of registers should be a `uint` array of length 32. The address busses should all contain a `byte` value, as the number of addressable registers never exceeds $2^8 = 256$. The output busses and the Write Data bus should all contain a `uint` value. Finally, the `RegWrite` bus should contain a `bool`.

On each clock tick, the Register File should check if the `RegWrite` flag is set, in which case it should take the value from the Write Data bus, and store in the register at the address from the Write Address bus. Then it should take the value in the register at the address from the Read Register A bus, and output onto the Output A bus, and similarly for Read Register B and Output B. It should be noted that if the write address is 0, then the write should be ignored, as register zero is immutable. The simplified code for the Register File SME process can be seen in Listing 3.

3.3 ALU

The ALU (Arithmetic Logic Unit) is the part of the processor, which makes the actual computation. It takes three inputs: Input A, Input B and an ALU Opcode indicating which computation to perform. It has two outputs: The result of the computation, and a zero flag indicating whether or not the result of the computation was 0. The ALU and its connections can be seen in Figure 8.

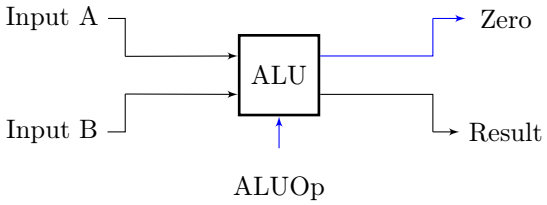

Figure 8: The ALU.

The ALU starts by looking at the value in the ALU Opcode, as this determines which operation to perform. Then it reads the values from Input A and Input B, and performs the operation specified by the ALU Opcode. Finally, it outputs the result, and sets the zero flag to whether the result was 0.

Implementation & Testing

To implement the ALU, I start by making an `enum`, so that the code becomes more human readable. Each entry in the `enum` corresponds to a computation that the ALU should perform. I could have followed the ALU Opcode given in the architecture book [2], however it states that its encoding of the ALU Opcode is generated using a CAD tool, and as such I will let the VHDL transpiler generate it too.

I start by implementing the same instructions as the architecture book [2] proposes in chapter 4.1: `add`, `sub`, `and`, `or`, `slt`, `sw`, `lw` and `beq`. To perform these instructions, the ALU should be able to perform addition, subtraction, bitwise `AND`, bitwise `OR` and the comparison less than.

The two input busses and the Result bus should all contain a `uint` value, the ALU Opcode bus should contain a `byte` value and the Zero bus should contain a `bool` value.

Constructing the ALU process in SME is straightforward, it reads the ALU Opcode from the ALU Opcode bus, and then it performs a `switch` on the ALU Opcode. For the instructions that it accepts, it takes the input from the two input busses, do the computation, and output the result on the Result bus. SME will handle the the actual computation, and as such I do not need to implement the ALU circuitry, but rather do the right computation in C#. Note: it is important to cast the `uint` input to `int`, if the operation is signed.

Finally, the ALU should set the flag on the Zero bus, depending on whether or not the result of the computation was 0. How the ALU opcode is encoded is described in Section 3.8.

3.4 Sign Extend

The Sign Extend is used for extracting the 16-bit values from the instruction, called the Immediate. It has one input: the lower 16 bits from the instruction, and one output: the 32-bit sign extended value. It takes its input, which is 16-bit, and converts it into a 32-bit value, extending the sign if present. The Sign Extend and its connections can be seen in Figure 9.

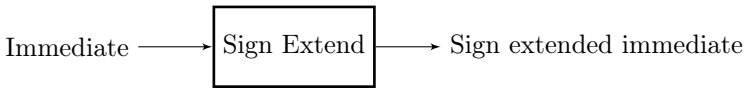

Figure 9: The Sign Extend.

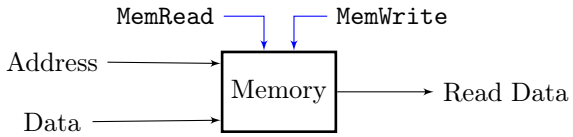

Figure 10: The Memory Unit.

Implementation

To implement the Sign Extend, I construct an SME process. The input bus should contain a `short` value, and the output bus should contain a `uint` value. On each clock tick, the SME process takes the 16-bit immediate, and outputs it on its 32-bit output bus. I could manually do the sign extending, but I just let C# handle the sign extension.

3.5 Memory unit

The Memory Unit is the main memory of the processor. It is the second step of the memory hierarchy, and is thus slower than the Register File, but can contain a lot more data. The processor can either read or write to the Memory Unit in a single cycle. The addresses for the memory are byte addresses and word aligned (i.e. in the 32-bit processor, the word size is 32 bit) and therefore the address should be divisable by 4.

The Memory Unit has four inputs: Address, Data, `MemRead` and `MemWrite`. It has a single output: Read Data. In one clock cycle, the Memory Unit either reads or writes. The Memory Unit and its connections can be seen in Figure 10.

Implementation

To implement the Memory Unit, I construct an SME process. The Address, Data and Output busses should all contain a `uint` value. The two control busses `MemRead` and `MemWrite` should both contain a `bool` value.

As with the Instruction Memory, it is going to need a chunk of memory. Like the Instruction Memory, the memory chunk should be a `byte` array, and should read and write in the same manner, i.e. either packing or unpacking 4 `byte`s from and to memory.

On each clock tick, the process should check if the `MemRead` flag is set, in which case it should read the value on the Address bus, and output the value stored in memory at the read address. If that was not the case, it should check if the `MemWrite` flag is set, in which case it should read the value at the Address bus and the Data bus, store the read data in memory at the read address.

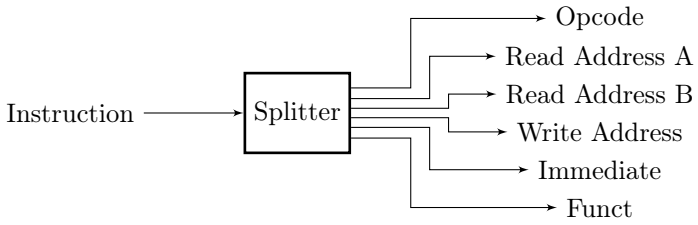

Figure 11: The Splitter.

3.6 Splitter

The architecture book [2] takes the instruction read from the Instruction Memory, and splits it out to the different components. One way of handling this could be to send the instruction bus to all the components that needs something from the instruction. However, then the resulting VHDL might send all 32 bits to multiple components, when 5 bits might have been sufficient. As such, I construct my own component for splitting up the instruction.

The splitter is a very simple component: It takes the instruction, which has been fetched from memory, and divides it into chunks for the different parts of the decoding. The instruction is partitioned as followed (the bits are inclusive):

- Opcode - bits 26-31
- Read Address A - bits 21-25
- Read Address B - bits 16-20
- Write Address - bits 11-15
- Immediate - bits 0-15
- Funct - bits 0-5

The Splitter and its connections can be seen in Figure 11

Implementation

The implementation is straightforward: I construct an SME process, which takes the instruction coming from the Instruction Memory, and extract the bits at the indices. It would be nice to be able to split a signal, in the same manner as VHDL does it (i.e. by having a slice index), however SME does not support this. Instead, I use use C# bit hacking, by shifting and masking the number, as this has the semantics as splitting a signal. The Input bus should contain an `uint` value. The Immediate bus should contain a `short` value. The rest of the busses should contain a `byte` value. Finally, the process should output the extracted values on the respective output busses.

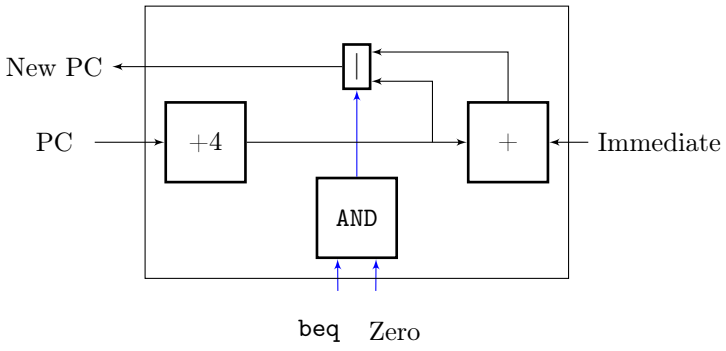

Figure 12: The Jump Unit.

3.7 Jump Unit

The architecture book [2] describes how to implement `beq`, by the use of multiple components. However, I combine these components into a single unit, the Jump Unit, to simplify the processor. The unit should still be composed of the subcomponents as suggested by the architecture book [2].

The Jump Unit is the one controlling which instruction to load next. It takes four inputs: sign extend, Zero, `beq` and the PC. It produces one output: the new PC.

For the simple single cycle MIPS processor, it should only have support for normal program traversal (i.e. execute the instructions in order) and the `beq` instruction. I will be adding support for more branch and jump instructions later in Section 4.3.

For the simple traversal, the Jump Unit takes the previous PC, and increments it by 4. For the `beq` instruction, it takes the value from the Sign Extend, shifts it left by 2, and add that value to the incremented PC. Finally, it chooses between the incremented PC and the added sign extend, based on the `beq` and Zero flags. The Jump Unit, its connections and its internals can be seen in Figure 12.

Implementation

I have the all of the logic described in the background. To implement the Jump Unit, I implement all of the logic components as SME processes. I am going to need 4 components: an +4 incrementer, an adder, a `AND` gate and a multiplexor. I could have made all the logic in a single process. However this makes it easier to extend, increases concurrency and is closer to the CSP model, as each subcomponent is very simple.

The Incrementer takes the PC bus as input, and produces a single output: the incremented bus. The two busses should both contain a `uint` value. On each clock tick, it should take the value from the PC bus, add 4 to it and put it on the incremented bus.

The Adder takes the inc bus and the immediate bus from the Sign Extend as input. It has a single output bus: the added bus. All of the busses should contain a `uint` value. On each clock tick, the Adder should add its two inputs

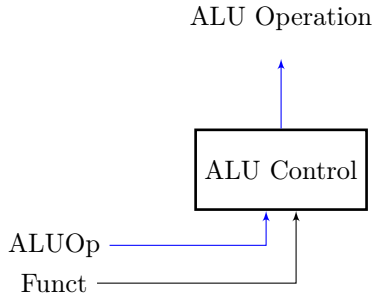

Figure 13: The ALU Control.

together, and put the result on the add bus.

The `AND` gate is like in the Logic Gates example, but with the `beq` and Zero as inputs, and its output should be on a new bus: anded. All of its busses should contain a `bool` value.

Finally, I have the multiplexor, which takes the incremented, added and anded busses as input, and produces a single output: the new PC. The new PC bus should contain a `uint` value. On each clock tick, if the flag from the anded bus is set to `1`, then it should put the value from the added bus onto the new PC bus. Otherwise it should output the value from the inced bus.

3.8 ALU control

The ALU control is used for generating the ALU Operation control code, which the ALU uses for selecting which operation to perform. It takes two inputs: the `ALUOp` code from the control unit, and the `funct` code from the instruction. It produces a single output: The ALU Opcode. The ALU Control and its connections can be seen in Figure 13.

The MIPS processor accepts three basic instruction formats [3]: The R format, the I format and the J format. The R format instructions are decoded in the ALU Control. The I and J formats are decoded in the Control Unit. The R format is divided into six parts:

- `opcode` (31-26) : Opcode of the instruction
- `rs` (25-21) : Source register 1
- `rt` (20-16) : Source register 2
- `rd` (15-11) : Destination register
- `shamt` (10-6) : Shift amount (only used by shift instructions)
- `funct` (5-0) : Function opcode

If the `ALUOp` indicates that the instruction is an R format instruction, it uses the `funct` code in the instruction for selecting the operation. Otherwise it bases its output on the `ALUOp` code. I will return to this component, when I need to extend the instruction set in Section 4.3.

Implementation

The architecture book [3] describes the logic to implement the ALU Control. However, like the ALU, it has been generated, and is difficult to extend. As such, I construct my own ALU Control, and let the VHDL transpiler handle the logic generation.

To make the source code more human readable, I have two additional `enum`s: one for the `ALUOp` code and one for the `funct`. Then I construct an SME process, which has the connections as specified in the background section. All of its busses should contain `byte` values.

On each clock tick, the ALU Control checks if the `ALUOp` code indicate R format instruction or not. If the instruction is an R format, the process should `switch` on the `funct` code. If not, it should `switch` on the `ALUOp` code. In all cases in both `switch`s, the ALU Control should output the ALU Operation corresponding to the computation that the instruction expects.

3.9 Control Unit

The control unit is part of the decoding step. It takes the opcode of the instruction, and based on the opcode, it sets control flags used throughout the processor. All of the control flags mentioned throughout the other components are set by the Control Unit. It sets the following control flags:

RegDst	Controls which part of the instruction that indicates the Read Address B to the Register File.
Branch	Controls whether or not the instruction is a branch instruction.
MemRead	Controls whether or not there should be read from memory.
MemtoReg	Controls whether or not the value from memory should be stored in the register file.
ALUOp	Opcode indicating which operation should be performed in the ALU. It is sent to the ALU Control for further processing.
MemWrite	Controls whether or not data should be written to memory.
ALUSrc	Controls whether the B input for the ALU should be the value read from the register file, or if it should be the value extracted from the instruction.
RegWrite	Controls whether or not data should be written to the register file.

As mentioned in Section 3.8, the Control Unit decodes I and J format instructions. The J format instructions are described in *Jump instruction* in Section 4.3. The I instructions are the immediate instructions, i.e. instructions where one of the source values are contained within the instruction [3]. An I format instruction is divided into four parts:

- `opcode` (31-26) Opcode of the instruction
- `rs` (25-21) Source register
- `rd` (20-16) Destination register
- `immediate` (15-0) Immediate source

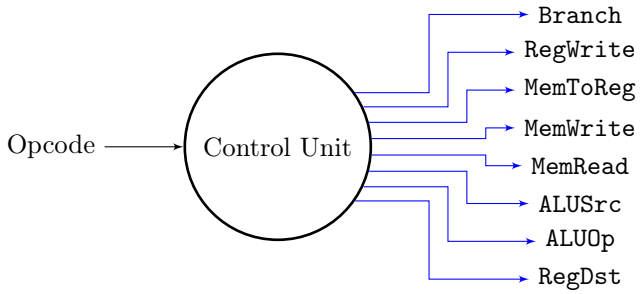

Figure 14: The Control Unit.

Each of the control flags goes to their respective part of the processor. I will return to this component, when I have to extend it to handle more instructions in Section 4.3. The control unit and its connections can be seen in Figure 14.

Implementation

As with the ALU Control, the architecture book [2] describes the logic needed to implement this unit. However, again it is not trivial to extend later on, so I construct my own Control Unit, and let the VHDL transpiler handle the logic generation.

I construct a SME process, which has all of the busses as described previously. The input opcode bus and the ALUOp code bus should both contain an `byte` value. The rest of the busses should all contain a `bool` value. As with the ALU Control, I have an `enum` on the opcode, to make it more human readable. I do not need one for the ALUOp, as the ALU Control already describes it.

On each clock tick, the Control Unit reads the opcode from the input bus. Then it should `switch` on the read opcode, and set all of the flags accordingly. How the flags should be set, depends on the instruction, and should be straightforward, but time demanding, as each accepted instruction needs to be in the `switch`.

3.10 Write back

The final stage of the processor is the Write Back. Here, the values are sent to the Register File for storing.

Implementation

Usually in the single cycle MIPS processor, there is nothing special in the Write Back stage. However, in SME it is not allowed to have unclocked cycles, and there is a cycle from the Register File, through the ALU and the Memory Unit, and back to the Register File.

To solve this, I introduce a Write Buffer. The write buffer should be clocked, and takes the Write Data, Write Register and `WriteEnabled` as input, and produces the same output. On each clock tick, it should output its stored values, and store its input values.

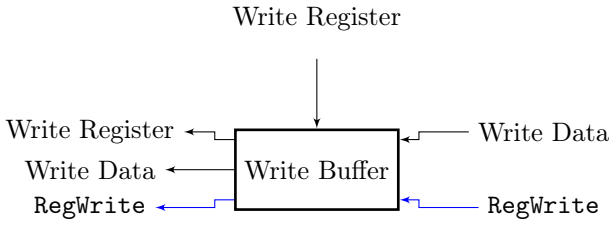

Figure 15: The Write Buffer.

I could also have made one of the previous components clocked, however I do not want to do this, as this specific problem will be solved when I pipeline the processor in Section 5, as the Write Buffer can be removed.

The Write Buffer and its connections can be seen in Figure 15.

4 Single cycle MIPS processor

In this section, I will be combining the core components into a single cycle MIPS processor, i.e. a processor where exactly one instruction is executed per clock cycle. When it is in place, I will be writing the first program, and compiling it into the processor, and running it.

Following the single cycle MIPS processor, I will be extending the processor so that it can handle more instructions. Along each added instruction, I will be extending the first program, in order to verify that the added instruction works.

Finally, I will be writing two larger programs, and look into compiling them into a series of hex values, that I can copy straight into the Instruction Memory.

4.1 Wiring up the processor

The source code for the single cycle processor is available at [12] in `sme/src/Examples/SingleCycleMIPS/`. With all the components in place, wiring up the single cycle MIPS processor is straightforward. I just need to declare the busses with the corresponding names, and then SME handles the wiring process. Had this been implemented in CSP, each writer process would have to ensure that each reader process would get a copy. However, as mentioned in the introduction of the paper, SME has broadcasting channels, which does exactly that so there is no need for additional logic. Note that as previously mentioned, the Write Buffer and the PC register has to be clocked processes. The wiring of the single cycle MIPS processor can be seen in Figure 16. Note that the multiplexors have not been mentioned as components, as these should be simple and straightforward to insert into the places specified in Figure 16.

4.2 Writing the first program

The source code for the first program, with the additional instructions introduced in Section 4.3, is available at [12] in `MIPS/first_program.asm`. As mentioned before, the first single cycle MIPS processor should be able to handle `add`,

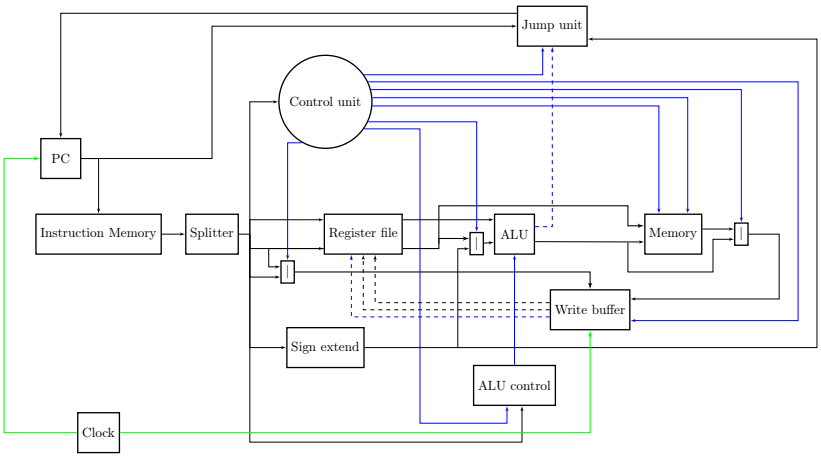

Figure 16: Simple single cycle MIPS processor. The units with | indicate multiplexors. The Clock is not an SME process, but has been added to emphasize which processes that are clocked. The blue wires indicate control wires, i.e. buses that contain a `bool` value. The black wires indicate wires containing data (i.e. `byte`, `short` and `uint`). The dashed and bold wires do not indicate anything special, but are there to emphasize crossing wires.

`sub`, `and`, `or`, `slt`, `sw`, `lw` and `beq`. As such, the first program should consist of these. The program and the different parts of each of the instructions can be seen in Table 2.

Inserting this program into the Instruction Memory is straightforward: just create a `byte` array, which is initialized to the hex values from the instruction column in Table 2. Since I do not have any way of feeding values into the Register File at this moment, I am going to hardcode registers 1 and 2 with the values 5 and 2 respectively. When the program has finished, the contents of the register file should be as in Table 3.

4.3 Extending the accepted instruction set

Then I will extend the accepted instruction set of the processor, in order to execute more complex programs. To test each of the added instructions, I just append the instructions to the initial program.

Additional simple R format instructions

The simplest instructions to add, are the remaining simple R format instructions. These are `addu`, `subu`, `xor`, `nor` and `sltu`. The only modifications are to extend the ALU Operation `enum`, the `funct enum` and then add the remaining cases in the ALU Control and the ALU. As described in Section 3.3, it is important to cast the input `uint` to `int` before making the computation, if the operation is signed.

Address	Instruction	opcode	rs	rt	rd/imm	shmt	funct	hex
0x00	add $3 $1 $2	0x00	0x01	0x02	0x03	0x00	0x20	0x00221820
0x04	sub $4 $3 $2	0x00	0x03	0x02	0x04	0x00	0x22	0x00622022
0x08	and $5 $3 $1	0x00	0x03	0x03	0x05	0x00	0x24	0x00612824
0x0C	or $6 $3 $1	0x00	0x03	0x03	0x06	0x00	0x25	0x00613025
0x10	slt $7 $6 $5	0x00	0x06	0x05	0x07	0x00	0x2A	0x00C5382A
0x14	sw $6 0x0($0)	0x2B	0x00	0x06	0x0000	-	-	0xAC060000
0x18	lw $8 0x0($0)	0x23	0x00	0x07	0x0000	-	-	0x8C070000
0x1C	beq $5 $4 0x4	0x04	0x05	0x04	0x0004	-	-	0x10A40004
0x20	add $9 $8 $6	0x00	0x08	0x06	0x09	0x00	0x20	0x01064820
0x24	add $10 $8 $6	0x00	0x08	0x06	0x0A	0x00	0x20	0x01065020

Table 2: The first MIPS program testing the `add`, `sub`, `and`, `or`, `slt`, `sw`, `lw` and `beq` instructions. Note that the instruction at `0x20` should be skipped due to the `beq` at `0x1C`.

Address	0	1	2	3	4	5	6	7	8	9	10
Value	0	5	2	7	5	5	7	1	7	0	14

Table 3: The register file after the first program has finished.

Immediate instructions

The next instruction I am going to add is the `ori` instruction, as this would allow me to feed values into the processor without hardcoding it. While I am doing this, I also add `addi`, `addiu`, `slti`, `sltiu`, `andi` and `xori`, as these require the same amount of work.

For the logical immediates, it is important, that the Sign Extend does not sign extend. As such, I add another signal to the Control Unit: `LogicalImmediate`, which goes to the Sign Extend. The Sign Extend should then, in the case the `LogicalImmediate` flag is `1`, cast its input as `unsigned`, such that any potential sign bits are not extended. Then I just need to extend the opcode and ALUOp `enum`, and add the remaining `case`s in the Control Unit and the ALU Control.

Once the processor has been extended, I can prepend instructions at the beginning of the program, to feed values into the processor. Furthermore, I can remove the previous hardcoding of initial values in the Register File.

Jump instruction

I am going to introduce another format: the J format [3]. This format is used for executing the `j` instruction. An J format instruction has two parts:

- `opcode` (31-26) Opcode of the instruction
- `address` (25-0) Jump address

The first step is to extend the Splitter, as it should now send the lower 26 bits (25-0) to the Jump Unit. Then the Control Unit should be extended, both in the opcode `enum`, and it should have another control signal output: `jump`, which should be connected to the Jump Unit.

Finally, the Jump Unit should be extended. It should take the 26 bits from the Splitter, and left shift it by 2, such that it becomes a 28 bit number. This is done to pack more information into the address in the instruction, as there are only 26 bits available. The adresses into the Instruction Memory has to be word aligned. Since I am making a 32-bit processor, the word size is 4 bytes. As such, the 2 least significant bits are always 0 to keep the word alignment.

In order to get the address to be a full 32 bit number, the Jump Unit should take the 4 most significant bits from the PC+4, and prepend them to the extended number. Finally, I am going to add a multiplexor, which takes this newly computed address, the previous output address, and the `jump` control signal. If the control signal is `0`, then the original output should be output, otherwise the newly computed jump address.

Branching instruction `bne`

Then I am going to add the branch instruction `bne`. I add another signal from the Control Unit to the Jump Unit. Then I split the Zero signal into two, where one of the signals goes to a newly added `NOT` gate. Then I add a multiplexor, which has the `bne` signal from the Control Unit as the control signal, and the Zero and the `NOT` Zero as inputs. If the `bne` control signal is `0`, then the original Zero should be put on the output, otherwise the `NOT` Zero. The output from the multiplexor should go into the `AND` gate, where the Zero signal originally went.

Additional jump instructions

Then I am going to add the jump instructions: `jr` and `jal`, as these are useful when writing the larger programs later.

I start with `jr`. The instruction is in R format, so I do not know it is `jr`, until it has reached the ALU Control. As such, I am going to need a control signal from the ALU Control to the Jump Unit. I am also going to forward Output A from the Register File to the Jump Unit, as this is the address that the processor should jump to in the `jr` instruction. The Jump Unit should not compute the new address in the same manner as with the `j` instruction. This is due to the registers being a full 32 bit, and thus can contain the whole address space. The Jump Unit should also have a multiplexor, controlling whether the jump address should be the immediate value, or if it should be the value from the instruction.

For the `jal` instruction, I am going to need an extra unit following the ALU. In the case of a `jal` instruction, the PC+4 address is stored in register `31` (which is called the `$ra` register). I add an additional control signal from the Control Unit: the `jal` signal. The new JAL Unit should take three inputs: the ALU Result, the PC+4 and the `jal` control signal. It should produce two outputs: the Write Address for the Write Buffer, and the value to store. If the `jal` signal is `1`, the JAL Unit should output the PC+4 on the value bus, and `31` on the address bus. Otherwise it should output the regular ALU Result, and the regular Write Address.

Shift instructions

Then I am going to add the shift instructions: `sll`, `slr`, `sra`, `sllv`, `srlv` and `srav`, as shifting is often useful.

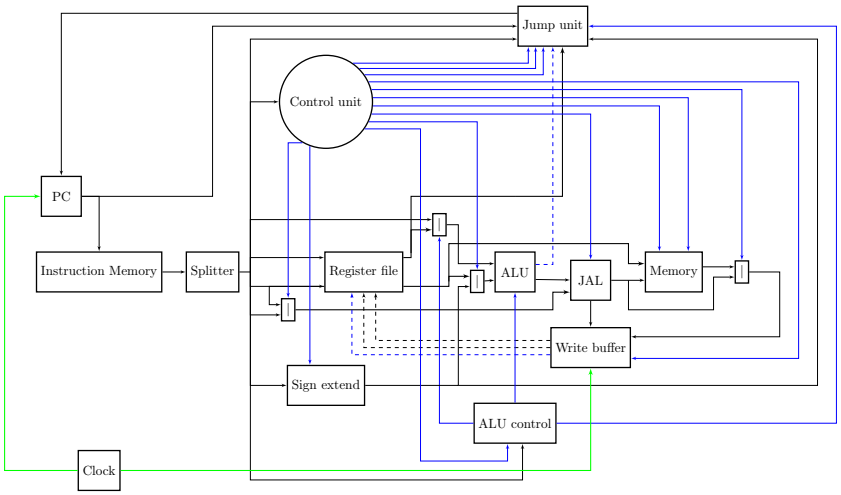

Figure 17: Full single cycle MIPS processor. The units with | indicate multiplexors. Black wires indicate data wires. Blue wires indicate control wires. Green wires indicate clock.

The shifting itself is performed in the ALU, and modifying the ALU to handle these is straightforward. The problem is that in an R format instruction, which the shift operations are, the shift amount (`shamt`) is stored in its own field within the instruction. As such, the splitter should extract these 5 bits, and send them to a new multiplexor, which also takes Output A from the Register File. As with the `jr` instruction, I do not know it is a shifting instruction until it reaches the ALU Control. So to control the new multiplexor, I need a Control signal from the ALU Control, indicating whether the multiplexor should output either the `shamt` or Output A from the Register File.

Multiplication and Division instructions

Then I am going to add the multiplication and division instructions: `mult`, `multu`, `div` and `divu`. All of these instructions put their result in two special registers: HI and LO. As such, to get the results from them, I also need to add the `mfhi`, `mthi`, `mflo` and `mtlo` instructions.

I start by adding the special registers. Since they are performed in the ALU, I implement them as variables inside the ALU. Then, when the ALU is doing the computation, it puts the values there, and since the instructions do not write to registers, touch memory or change the PC register, it does not matter what is put on the ALU Result or Zero busses.

The instructions handling the moving to and from the HI and LO registers are fairly simple to implement: just either input or output the corresponding register, to or from the ALU. The layout for the fully extended single cycle MIPS processor can be seen in Figure 17.

4.4 Larger MIPS programs

To test the full single cycle MIPS processor, I am going to implement two programs in MIPS assembly: Quicksort and Towers Of Hanoi. I chose these two examples, as both are simple to implement, and they do not require anything special from the environment.

I am not going to give the implementation in MIPS assembly, as this would not be readable. Instead, I give some low level C code, which should be easy translatable into MIPS assembly, and easy to verify on a real machine. Once I have made the assembly, I can easily dump it into MARS [14], which can then produce a ascii hex dump, which I can then paste into the instruction memory.

Quicksort [4]

Quicksort is a sorting algorithm, for doing comparison based sorting. There are three parts of the Quicksort program: Loading data into memory, the partition function and the quicksort function. I am going to use the partition function from the algorithm book [15], and the quicksort function, also from the algorithm book, as both are simple to implement.

I could have implemented the Hoare partitioning algorithm. However, it is not as simple, and performance is not the target of the program, but rather correctness of the processor. I.e. that the processor implementation produce the same result as MARS, in the same amount of clock ticks.

Loading data into memory I start by loading data into memory, so that the quicksort program has some data to sort. Which numbers are not important, rather the amount of numbers, as the quicksort algorithm uses it as argument. I construct a function called `load`, which inserts 8 numbers into the given memory address.

```
1  void load(int *a) {
2     *(a)   = 5;
3     *(a+1) = 8;
4     *(a+2) = 2;
5     *(a+3) = 9;
6     *(a+4) = 1;
7     *(a+5) = 3;
8     *(a+6) = 7;
9     *(a+7) = 11;
10 }
```

The partitioning function Then I implement the `partition` function as described in the algorithm book [15]. Note that statements from the algorithm book have been expanded to more closely resemble assembly.

[4] The source code for the MIPS assembly code and the C code is available at [12] in `MIPS/qsort.asm` and `MIPS/qsort.c`

```
1  int partition(int *a, int p, int r) {
2      int x, i, j, tmp1, tmp2, *addr1, *addr2;
3      addr1 = a + r;
4      x = *(addr1);
5      i = p - 1;
6      for (j = p; j < r; j++) {
7          addr1 = a + j;
8          if (*(addr1) <= x) {
9              i++;
10             addr1 = a + i;
11             addr2 = a + j;
12             tmp1 = *(addr1);
13             tmp2 = *(addr2);
14             *(addr1) = tmp2;
15             *(addr2) = tmp1;
16         }
17     }
18     addr1 = a + i + 1;
19     addr2 = a + r;
20     tmp1 = *(addr1);
21     tmp2 = *(addr2);
22     *(addr1) = tmp2;
23     *(addr2) = tmp1;
24     return i + 1;
25 }
```

The quicksort function Finally, I implement the `quicksort` function as described in the algorithms book.

```
1  void quicksort(int *a, int p, int r) {
2      if (p < r) {
3          int q = partition(a, p, r);
4          quicksort(a, p, q-1);
5          quicksort(a, q+1, r);
6      }
7  }
```

Initial call to the algorithm To emphasize the order of calling, and the arguments, I also construct a `main` function.

```
1  int main() {
2      int arr[8], i;
3      load(arr);
4      quicksort(arr, 0, 7);
5  }
```

Do note that the arguments to both the `quicksort` and `partition` functions, are inclusive. To verify the correctness, we should look at the memory, where the data now should be in sorted order. The performance of the quicksort program can be seen in Table 4.

		MARS			SME		
		# CT	time (ms)	CR (hz)	# CT	time (ms)	CR (hz)
Towers of Hanoi	$n = 5$	719	585	∼1229	720	516	∼1395
Quicksort	$n = 8$	483	582	∼829	484	375	∼1290

Table 4: Performance of the Quicksort and Towers of Hanoi programs on the single cycle processor. CT is clock ticks. CR is clockrate. Compared to `MARS`, the SME simulation uses an additional clock tick. This is due to the simulation using an additional clock tick for shutting down the simulation. The execution time of MARS is dominated by the startup time, which is why there is no deviation. The benchmark was performed on a laptop with an Intel Core i5-5300U (2.3 GHz).

Towers Of Hanoi [5]

Towers Of Hanoi is a puzzle, where one has to move a tower of discs, from one peg, to another, with one additional auxilary peg, by only moving one disc at a time.

By searching on the net, I find a pseudo approach to the problem, which uses recursion [16]. I am going to represent the three pegs as an array, which is three times the size of the tower. As such, each peg is just one third of the array. Furthermore, as with quicksort, I start by loading data into memory, have a `tower` function, which performs the move, and finally a correct way to call the functions.

Loading data into memory I need to initialize the memory. I am going to fill the first third of the array, i.e. the first peg, with descending numbers. Each number indicates which disc it is. The rest of the pegs with 0, indicating no disc.

```
1  void init(int num, int *from, int *to, int *aux) {
2      int i;
3      for (i = 0; i < num; i++) {
4          *(from+i) = num - i;
5          *(to+i) = 0;
6          *(aux+i) = 0;
7      }
8  }
```

The tower function This is the `tower` function, which moves the discs from peg to peg. I use the same approach as the pseudo code [16]. As with quicksort, the statements have been expanded to more closely resemble assembly. I make the three arguments as stack pointers to each of the pegs. A stack pointer is the address to the next free memory position in the stack data structure [3]. To preserve each of the pointers, I have pointers to each of them, i.e. pointer pointer.

[5] The source code for the MIPS assembly code and the C code is available at [12] in `MIPS/hanoi.asm` and `MIPS/hanoi.c`

```
1  void tower(int num, int **from, int **to, int **aux) {
2      int *t, *f;
3      if (num == 0) {
4          t = *to;
5          f = *from;
6          f--;
7          *t = *f;
8          t++;
9          *f = 0;
10         *to = t;
11         *from = f;
12     } else {
13         tower(num-1, from, aux, to);
14         t = *to;
15         f = *from;
16         f--;
17         *t = *f;
18         t++;
19         *f = 0;
20         *to = t;
21         *from = f;
22         tower(num-1, aux, to, from);
23     }
24 }
```

Calling the algorithm I construct a `main` function, to emphasize the order of calling and the arguments.

```
1  int main() {
2      int num = 5;
3      int *arr = int[num*3];
4      init(num, arr, arr+num, arr+(2*num));
5      int *f=num, *t=num, *a=2*num;
6      tower(num-1, &f, &a, &t);
7  }
```

When the program has finished, the last third of the array should contain the numbers in descending order. The performance of the hanoi program can be seen in Table 4.

5 Pipelining

In this section, I will be looking at pipelining the single cycle MIPS processor, and handle the problems, which are introduced by pipelining. I start by going through the background and motivation for pipelining, and then proceed on how to extend the single cycle MIPS processor to have pipes.

As mentioned, pipelining introduces new problems to handle in the processor, and I will solve it by adding two new components: the Forwarding Unit, which forwards results from previous instructions to later instructions, and the Hazard Detection Unit, which controls when to stall the pipeline. Throughout each step, I will also be writing programs, in order to verify that the processor behaves as specified. The source code for the pipelined processor with forwarding and hazard detection is available at [12] in `sme/src/Examples/PipelinedMIPS/`.

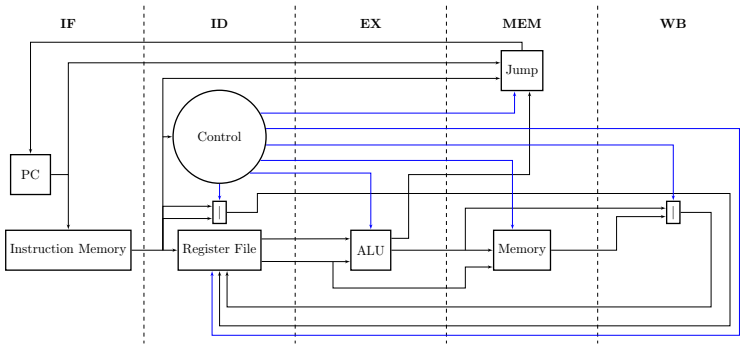

Figure 18: The simplified single cycle processor, and its stages. The stage names are highlighted in **bold**. The stages are divided by dashed lines.

5.1 Introducing the pipes

I have the single cycle MIPS processor, which accepts some of the core integer instruction set. However, it is not very efficient, as the clock rate of the processor is determined by the longest possible path in the processor. A path in a processor is the components that a signal goes through, until it reaches a safe state. A safe state in a processor is a state, where the signals are safely stored in either registers or memory. A longer path implies a lower clockrate. So in order to increase the clock rate, I must decrease the longest path in the processor. I solve this by introducing pipes.

Pipes are registers in the processor, where all the values computed so far are temporarely stored. It takes all of its inputs, and holds them until the next clock tick, where it will forward the values it is holding. This ensures that the data does not have to travel as far, until it have reached a safe state.

In order to know where to put the pipes, I divide the processor into stages. I will use the same stages, as proposed in the architecture book [2], i.e. divide the processor into 5 stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write Back (WB). A simplified overview of the processor and its stages can be seen in Figure 18. In between each stage, I am going to insert a pipe, i.e. I am going to insert 4 pipes. The simplified processor with pipes can be seen in Figure 19.

Introducing pipes also introduces additional problems, which I will discuss further in sections 5.2 and 5.3

Implementation

There are two ways to implement pipes in SME: by using clocked busses, or by using clocked processes. If the bus only traverses two stages, then I can use clocked busses, as the semantics of a clocked bus is exactly the same as a pipe (i.e. the value is not passed along the bus, until the clock ticks). However, if the bus traverses more than two stages, I am going to need additional processes, as the clocked bus only stores its value for one clock tick. Furthermore, if the only change to the bus is the given `ClockedBus` attribute, then determining whether

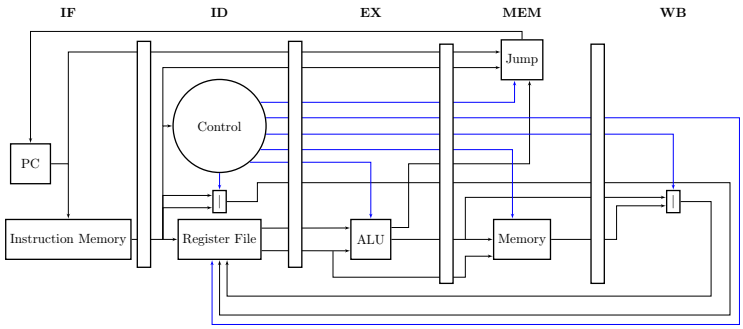

Figure 19: The simplified pipelined processor. The long bars in between each state are the pipes.

or not a bus is part of a pipe can be problematic. As such, I have the pipe and its busses in their own class, as it then receives its own namespace, and the code then becomes more readable.

Let us take a subset of the IF/ID pipe, as an example. I assume that each state is in its own classes, i.e. `IF` and `ID`, and that the `IF` stage has the `Instruction` bus, which the `ID` stage reads from. Then, by adding a subclass to the `IF` stage, the name of the bus that the `ID` class calls is converteted from `IF.Instruction` to `IF.Pipe.Instruction`, emphasizing that the value is now fetched from the piped bus.

Introducing the pipes in this manner is fairly straightforward, as I just add 4 classes, each with the same set of busses as the 'previous' stage outputs to the 'next' stage, and each with a register process, which forwards all the values from the 'previous' stages busses, onto its own busses with the same name.

This process can be repeated for all of the required pipes. By following the division of the components, as proposed in the architecture book [2], there is only one really tricky part of pipelining: the Jump Unit. The processor does not know whether to jump until the `MEM` stage, as the computations are made in the `EX` stage, and the conditions are computed in the `MEM` stage. However, the Program Counter has to increment, regardless of whether or not it should jump. As such, I should divide the Jump Unit into its subcomponents, and place some of the logic in the `IF`, some in the `EX` stage and finally some of it in the `MEM` stage.

For the `IF` stage, the incrementer and a multiplexor should be placed inside the stage. The multiplexor should take an address computed from the `MEM` stage, and the incremented Program Counter, and based on whether or not the instruction that has reached the `MEM` stage was a branch or jump instruction, it should choose the computed address.

The core computation of the jump and branch instructions should be placed in the `EX` stage. As such, the `EX` stage should compute both the branch address and the jump address.

Finally, the `MEM` stage should hold the decision logic. First, it should determine if the instruction was an branch instruction, and whether or not the condition has been satisfied, in which case, the address forwarded to the `IF` stage should be the branch address. If this was not the case, the computed jump address should be the one forwarded.

Finally, in the single cycle MIPS processor, I introduced the Write Buffer, in order to remove the cycle from and to the Register File. However, by introducing pipes, I have introduced a new buffer, and thus the Write Buffer can be removed.

Testing

To test the processor, I can use any of the programs, that I have previously written. However, since I have pipelined the processor, I need to insert bubbles, in order for data to be available for each instruction. A bubble is a No Operation (`nop`) instruction, which performs no operation, and does not modify neither the Register File nor the Memory.

I am going to implement a simple program, which is easy to verify: a small loop, which computes n fibonacci numbers, and places them in memory. As with the simple cycle, I am going to give pseudo low level C code:

```
1  void init(int *arr) {
2      *(arr)   = 1;
3      *(arr+1) = 1;
4  }
5
6  void loop(int *arr, int n) {
7      int i, tmp1, tmp2, tmp3;
8      for (i = 0; i < n; i++) {
9          tmp1 = *(arr+i);
10         tmp2 = *(arr+i+1);
11         tmp3 = tmp1 + tmp2;
12         *(arr+i+2) = tmp3;
13     }
14 }
```

Note that for verification in C, the size of the array should be $n + 2$, due to initialization. Furthermore, when I port it to MIPS assembly, after each instruction, I insert four `nop`'s, to ensure the data is ready for the next instruction. When the program has run, the $n + 2$ fibonacci numbers should be in memory, at the given address.

5.2 Forwarding

By introducing pipes, I also introduced data hazards and control hazards. I will go through control hazards in Section 5.3. Data hazards are when one instruction writes to a register, that a following instruction reads from. This is not a problem in the single cycle processor, as all data have been written to registers in the same clock cycle. This is not the case in the pipelined processor, e.g. the data might reside in the MEM stage, when it is needed in the EX stage.

I can eliminate some of the data hazards by implementing an additional unit: the Forwarding Unit. The rest of the data hazards will be handled by hazard detection in Section 5.3. The Forwarding Unit looks at the register addresses used by the instruction in the EX stage, and checks if it corresponds with the register write address of either the instruction in the MEM stage, or the instruction in the WB stage. If they correspond, it forwards the value from either the MEM or the WB stage to the EX stage. The overview of the simplified processor with the forwarding unit can be seen in Figure 20.

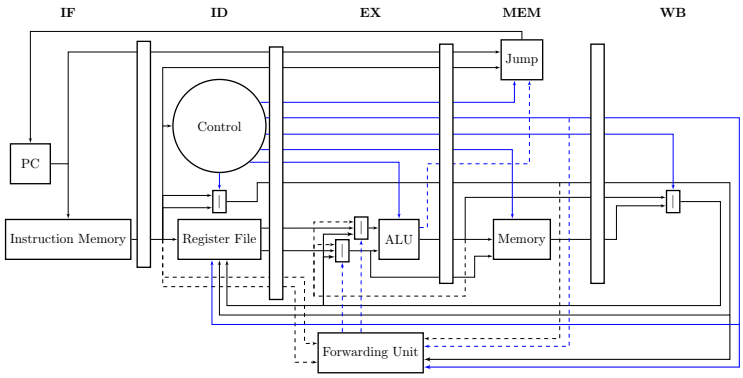

Figure 20: The simplified pipelined processor with forwarding.

I could also have gone with the bubble approach, i.e. insert bubbles every time there is a data hazard. However, this would not be optimal, as this would result in several clock ticks, where the processor is idle.

Implementation

Implementing the Forwarding Unit should be done with an SME process. As it interferes with the EX stage, it should be put in the EX stage. The unit should be controlling two multiplexors. The first multiplexor should control whether Output A should come from the ID pipe, the write data from the MEM stage or the write data from the WB stage. The other multiplexor is analogous, but with Output B. The two multiplexors should be controlled by the Forwarding Unit, i.e. it should generate the control signals for the multiplexors based on the two register read addresses of the current instruction in the EX stage, the register write address and write enabled signal both from the MEM and the WB stages.

Testing

Testing is straightforward: I can just remove all of the `nop`'s from the fibonacci program, except for those which come after either a load, a branch or a jump, as these hazards are not handled yet.

5.3 Hazard Detection

As mentioned in Section 5.2, I cannot avoid all data hazards with forwarding. This is because it only forwards to the EX stage. However, if the instruction prior to the instruction in EX is a load, then the data wont be available, until the load instruction has reached the WB stage. To handle this, the processor needs to detect the hazard and insert a bubble, as this will delay the instruction, so that when it has reached the EX stage, the load instruction will be in the WB stage, and thus it can be forwarded. When inserting a bubble in the middle of the pipeline, some of the pipes and registers must also be stalled. Stalling is the action of outputting what is stored in the register, but not updating it, i.e. outputting the same data in the next clock tick.

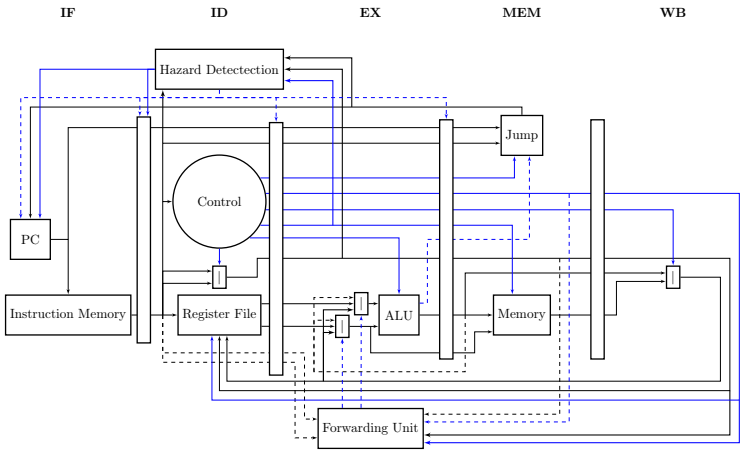

Figure 21: The simplified pipelined processor with forwarding and hazard detection.

I also need to be able to handle an additional type of hazard: control hazards. Control hazards are when either jump or branch instructions are executed. The problem is that the branch or jump is not performed until the MEM stage, which means that the pipeline following the jump or branch instruction may have been filled up by instructions, which should not be executed. To solve this, it should detect the hazard, and in such case flush the pipeline. Flushing the pipeline, is the action of resetting the registers in the pipes, so that they output a `nop` instruction. The overview of the simplified processor with the Hazard Detection Unit can be seen in Figure 21.

Implementation

The hazard detection should be implemented in its own SME process: the Hazard Detection Unit.

To solve the data hazards, it needs to read the `memread` control flag from the EX stage, the register write address from the EX stage and the two source register addresses from the ID stage. If the `memread` flag has been set, and either of the source registers match the destination register, then the ID/EX pipe should be flushed, the IF/ID pipe should be stalled, and the PC register should be stalled.

To solve the control hazards, it should read the `PCSrc` flag from the MEM stage (the one used by the PC register, to multiplex between the incremented address, and the jump/branch address), and if it is set, then the IF/ID, ID/EX and EX/MEM pipes should be flushed. Note: the PC register should read normally in the case of a flush, even though it might have to stall, as the stall is detected in the pipe that is being flushed.

Testing

Testing the new hazard detection is straightforward, as the processor should now be able to handle all of the previous programs, albeit in a larger amount

		MARS			SME		
		# CT	time (ms)	CR (hz)	# CT	time (ms)	CR (hz)
Towers of Hanoi	$n = 5$	719	585	∼1229	720 - 1058	516 - 1190	∼1395 - ∼889
Quicksort	$n = 8$	483	852	∼829	484 - 763	375 - 895	∼1290 - ∼852
Fib no optimization	$n = 10$	220	584	∼376	221 - 251	191 - 356	∼1157 - ∼753
Fib forward	$n = 10$	98	586	∼167	100 - 130	119 - 209	∼840 - ∼ 622
Fib hazard	$n = 10$	84	588	∼142	86 - 126	113 - 212	∼761 - ∼594

Table 5: Performance of the Quicksort, Towers of Hanoi and different versions of the fibonacci programs. CT is clock ticks. CR is clockrate. In the columns with multiple values, the left value is the performance of the single cycle processor and the right is the performance of the pipelined processor. The benchmark was performed on a laptop with an Intel Core i5-5300U (2.3 GHz).

of clock ticks, compared to the single cycle processor. The performance of the processor with hazard detection can be seen in Table 5.

By looking at the performance in Table 5, I see that the performance of the pipelined processor is worse both in execution time, clock ticks and clockrate. The additional clock ticks are expected of the pipelined processor, as there is an wind-up time for filling the pipeline, and as I loose some instructions by stalling and flushing.

By looking at the fibonacci program with hazard detection, I see that the single cycle processor uses 86 clock ticks. It runs the loop 10 times and for each branch it flushes the pipeline, i.e. losing $10 \times 3 = 30$ instructions. Furthermore, there is a dependency following a load instruction, producing a stall in each loop, i.e. losing an additional 10 instructions. By summing this up, I get $86 + 30 + 10 = 126$, which is exactly the amount of instructions performed by the pipelined processor.

The execution time of the SME simulation, and as such also the simulation clockrate, is worse. This is due to the SME simulation having more work to do, as I have increased the amount of processes and busses. There should not be any decrease in performance when synthesized to FPGA, but rather an increase. As mentioned in Section 5.1, it is because I should be able to increase the clockrate, compared to the single cycle implementation.

6 Transpiling and synthesizing SME

In this section, I will be describing the steps required to implement an SME network onto an actual FPGA. I start with the initial Logic Gates design, as it is very simple to verify and because it does not have any requirements regarding clocking.

Then I will be implementing the single cycle MIPS processor and finally show how to communicate with the hardware implemented on the FPGA. This section will be very hardware specific, as I have not verified or developed a general approach, which will run on additional hardware.

All of the source code, both for SME and VHDL, can be found at [12].

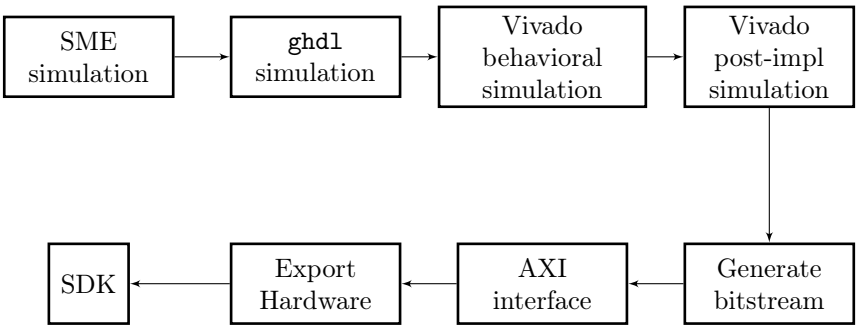

Figure 22: The general workflow for construction actual hardware using SME.

6.1 General workflow

This section will give a general description of the workflow required for implementing hardware on an FPGA using SME. A short overview of the steps and their order, can be seen in Figure 22,

In order to implement hardware onto an FPGA, one must describe the hardware using a hardware description language such as VHDL or Verilog. As mentioned before, SME can be transpiled into VHDL and as such provides a high level approach for hardware design. Therefore, the first thing to do is to write an SME network, verify that it runs as expected, and then checking that the transpiler does not fail to transpile. The top-level input and output busses has to be specified in SME. A top-level bus is a bus going either in or out of the SME network.

To generate VHDL from SME, the `Main()` function needs two additional lines: `.BuildCSVFile()` and `.BuildVHDL()`

```
1 public static void Main(string[] args)
2 {
3     new Simulation()
4         .BuildCSVFile()
5         .BuildVHDL()
6         .Run(typeof(MainClass).Assembly);
7 }
```

This will generate the VHDL code, a testbench for the VHDL code and a `csv` trace file used by the testbench. The trace file contains all the values sent on all of the busses within the network during simulation. The testbench takes all the input values in the `csv` file, sends them along the input busses, and finally verifies that the values on the other busses matches the value stored in the `csv` file.

Once the VHDL has been generated, the VHDL can be verified by using `ghdl` [17]. `ghdl` is an commandline simulator for VHDL. SME provides a testbench and a `Makefile` for running the testbench using `ghdl`. As such, to verify the generated VHDL, one only needs to run `make` inside the output `vhdl/` folder.

Once the `ghdl` simulation has been passed, the VHDL files can be imported into a project in Vivado [18]. Vivado is a development environment created by Xilinx, an FPGA vendor, for designing and implementing hardware on an

FPGA. When creating the project, Vivado needs to know the target FPGA. In this project, I have been using a ZedBoard.

A ZedBoard is a development board, which contains a Xilinx Zynq system on chip. A Zynq chip has two parts: a processing system (an ARM processor) and programmable logic (FPGA). These two parts are connected in two places: through an AXI interconnect and through DDR memory. I will be using the AXI interconnect, when communicating with the FPGA. AXI (Advanced eXtensible Interface) [19] is an interconnect specification, which communicates by using transactions. I am not going to focus on the details of the protocol. Additionally, the ZedBoard comes with a few buttons, switches and LEDs, which are connected to the FPGA part of the Zynq chip. These buttons and LEDs can be used as easy to set up input/output.

Once the project have been created in Vivado, the behavorial simulation should be run. The behavorial simulation is the same as the `ghdl` simulation and should as such produce the same result.

Then the design should be elaborated into Register Transfer Level (RTL). RTL is a design abstraction, which consists of lower level components (E.g. gates and registers) wired together. The RTL schematic can also be used to verify whether or not the design interpreted by Vivado matches the SME network. The schematic shows all of the top-level input and output busses. Additionally, there are the `CLK` (clock) and `RST` (reset) busses. The `CLK` bus is the clock signal used throughout the network. The `RST` bus is the reset signal. All the SME processes uses the `RST` signal, but only the clocked processes uses the `CLK` signal.

Within the RTL, the top-level input and output busses has to be connected to some specified wire on the FPGA (E.g. input signal from a button). At the same time, the communication standard should be specified. I just choose the default, as I do not have any special requirements. This is also where the clock signal and the reset signal are connected to wires. All of this is required by Vivado, in order for it to place and route the design.

Then the project is ready to be synthesized, placed and routed. This is done by the click of a button in Vivado and takes quite some time, especially for larger projects.

If the design uses a clock signal (i.e. has clocked processes), the first step is to synthesize the design. This is due to once the design have been synthesized, clocking constraints can be introduced. Vivado uses these constraints as a target, when it is placing and routing the synthesized design. The only clocking constraint for the designs in this thesis is the clockrate. When Vivado has placed and routed the design, it creates a timing report, stating whether or not the timing constraints where met. The timing report computes a slack on each of the paths in the implementation. Slack is the difference on how long it takes the signal to traverse a path, compared to the clocrate constraint. E.g. if it takes a signal 13 nanoseconds to traverse a path and the clockrate is specified to 10 nanoseconds per clock cycle, then the slack will be -3 nanoseconds. If the slack is positive, the clockrate should be increased. If the slack is negative, the clockrate should be reduced. After the clocking constraints have been introduced, the project needs to be synthesized again.

Once the project have been placed and routed, the post-implementation timing simulation should be run. This simulation is the closest to actual hardware, as it models all of the components of the FPGA, all the wires and the timing on the wires. As such, this is a very heavy simulation and takes some time to

run.

Furthermore, in the implemented design, Vivado can produce some interesting reports. The first interesting report is the utilization report, which reports how much of the logic available on the target board have been used. Generally, the most interesting metrics in the report are: Logic, IO, Registers, Block Memory and MMCM (clock multiplier/divider). In theory, the utilization report can be used to see how many instances of a design the target board can hold, by taking 100 divided by the metric with the highest value. E.g. the Single Cycle processor has Logic as the highest usage at 17% on the ZedBoard. I.e. $100/17 = 5.88 \approx 5$ cores could be fitted onto the ZedBoard.

Then we have the Power report, which reports an estimate of the power consumption of the design. The report contains a lot of metrics, but the one I focus on is the dynamic power consumption. This is the sum of approximate power consumption of each of the implemented components on the FPGA. I.e. the approximate power consumption of the implemented design.

The final report is the timing summary, which I will describe later.

If the post-implementation simulation passes, it should also work as expected on actual hardware. If clocking constraints were needed, the clock signal should be modified to match the clocking constraints. In order to do this, the design have to be implemented in a block design. There is a problem with the transpiled VHDL from SME. The top module uses custom types and Vivado does not allow custom types in a block design. To solve this, all the custom types must be changed to be standard types. E.g. `T_SYSTEM_UINT32` becomes `std_logic_vector(31 downto 0)`. Once this have been changed, the top module can be inserted into a block design. The clocking wizard should be added, in order to modify the clock to match the clocking constraints. Once the block design have been constructed, it should be made into a HDL wrapper, such that it can be synthesized, placed and routed. The final step is to generate the bitstream and write it to the FPGA.

Once the running bitstream has been verified, it is time to communicate with it. To do this, I must construct a custom Intellectual Property (IP). An IP is an already verified hardware component, which can have standardized connection and with these be connected to other IPs in a block design. This custom IP should have an AXI interface, so it can communicate with the ARM processor on the Zynq chip.

Vivado provides a template for generating IPs with an AXI interfaces. This template uses registers to send and receive information trough the interface. So before where the top-level busses were connected to wires on the FPGA, they must now be connected to these registers. This is done in VHDL, as the current version of SME does not support this. I could have written the AXI interface as an SME process. Doing it by VHDL seemed simpler however.

Once the IP have been constructed and verified, a new project with a new block design should be added. Inside the block design, the processing system of the Zynq chip should be added and connected to the custom IP with the AXI interface. Additionally, if timing constraints on the clock was needed during the implementation of before constructing the IP, then the clocking wizard should be added to the block design in order to either multiplying or dividing the clock. Again, once the block design have been verified, it should be made into a synthesizable component, by creating a HDL wrapper.

Once the bitstream has been generated, the hardware should be exported and

opened in the Xilinx SDK. From here, the ARM processor on the Zynq chip can be programmed bare-metal. Bare-metal programming is programming without an operating system, i.e. injecting the compiled machine code directly into the instruction cache of the ARM processor. The AXI interface have received a memory address, and the SDK should contains library with functions for accessing the registers through the AXI interface. With this, a driver should be written for the new logic.

Once the driver has been written, I can interface with the FPGA on the Zynq chip, by programming the ARM processor.

6.2 Logic gates

I start by implementing the first SME network, which I constructed: the logic gates. This network consisted of five SME processes: the four gates and the tester process. The tester process was only used for verification purposes, and should not be present in the generated VHDL. To solve this, the tester process should not be a `ClockedProcess`, but rather a `SimulatedProcess`. A `SimulatedProcess` has the same behavior as the `ClockedProcess`, except it does not generate any VHDL. The value which it inputs and outputs are caputered in the CSV tracefile, which is used by the generated testbench. Furthemore, all of the inputs and outputs to the gates, should be made top-level.

When running the testbench trough `ghdl`, I get the following output:

```
1 TestBench_LogicGates.vhdl:166:8:@50ns:(report note): completed after 5
      clockcycles
```

Which states that the simulation ran as expected in 5 clockcycles.

Then I create the Vivado project. As mentioned in the general workflow, the next step is to run the behavorial simulation and verify that it runs as expected. Running the Logic Gates should produce the waveform seen in Figure 23.

Then I verify that the RTL schematic matches the SME network. The RTL network of the Logic Gates project can be seen in Figure 24 and it matches the structure in Figure 1.

Then I choose where the top-level input and output wires should be connected. On the ZedBoard, I connect the outputs to the LEDs and the inputs to the switch buttons. I wire the reset signal to a button. Since the Logic Gates project does not use the clock, anything can be connected.

The next step is to synthesize, place and route the project. Once it is done, the post-implementation simulation is run. The waveform that it produces, is the same as Figure 23. The implemented design has a maximum utilization of 4%, which is the IO. It has an approximate power consumption of 0.001 watts. The comparison to the reported values of the other implementations can be seen in Table 7.

Then the bitstream is generated and written to the ZedBoard. Flipping the switches makes the LEDs light as shown in Figure 25.

Then the AXI interface is set up. This is done by creating a new project in Vivado and within that create and package a new IP. In order to use the AXI template that Vivado offers, I need to state how many registers it should have. Since the Logic Gates project has two inputs and four outputs, I choose six registers.

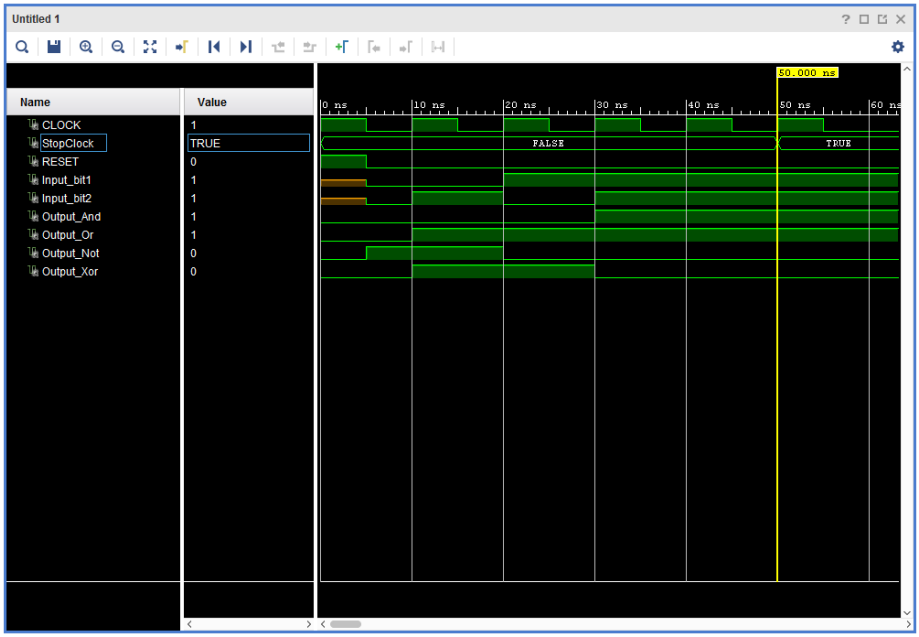

Figure 23: The waveform of the Logic Gates simulation.

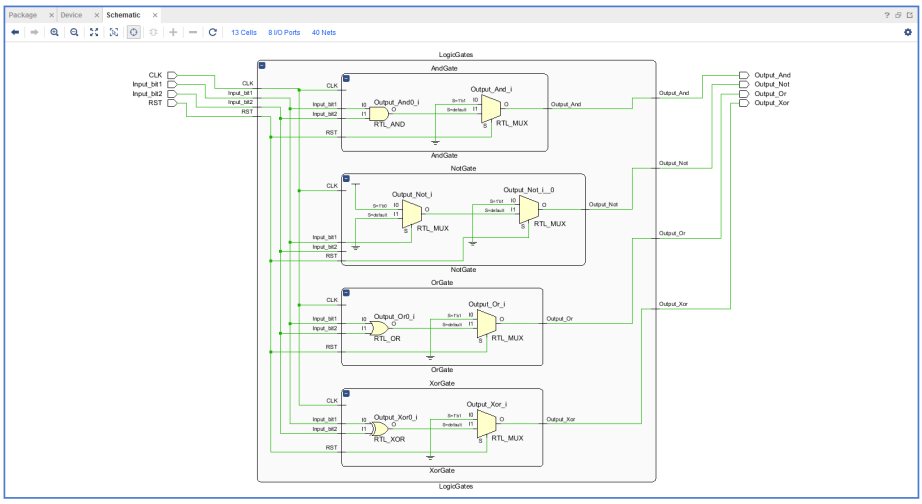

Figure 24: The RTL schematic of the Logic Gates project.

Figure 25: The LEDs with the different configurations of flip switches on the ZedBoard. From left to right the inputs on each of the images are: false-false, false-true, true-false, true-true. The four LEDs from left to right on each image are: `AND`, `OR`, `NOT` and `XOR`. Note: `NOT` only considers the first input.

The AXI template creates two files. The first file contains a wrapper component, which connects external connections to the inner components of the IP. If I needed a clock signal or reset signal, which should be seperate from the AXI clock and reset, the ports should be added here. The second file is the registers component of the AXI interface, which handles reading and writing to and from the slave registers. I am going to extend the registers to have signals from the Logic Gates and then extend the wrapper component, such that the Logic Gates are connected to the registers component.

The registers component gets two new output signals: `bit1` and `bit2`, and four new input signals: `and`, `or`, `not` and `xor`.

```
1 -- Users to add ports here
2 bit1 : out std_logic;
3 bit2 : out std_logic;
4
5 andgate : in std_logic;
6 orgate  : in std_logic;
7 notgate : in std_logic;
8 xorgate : in std_logic;
9 -- User ports ends
```

I need to ensure that the slave registers that the Logic Gates writes to are exclusively to the Logic Gates. I use slave registers 0 and 1 as the input bits, and slave registers 2, 3, 4 and 5 as the output. As such, all the occurrences where the registers component writes to slave registers 2, 3, 4 and 5, are commented out.

```
1 Before:
2 -- slave registor 2
3 slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto
      byte_index*8);
4
5 After:
6 -- slave registor 2
7 -- slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7
     downto byte_index*8);
```

Finally, the registers component needs the new logic. It outputs the contents of slave registers 0 and 1 to the `bit1` and `bit2` busses. Then it should take the

values from the four Logic Gate outputs, and store them in slave registers 2, 3, 4 and 5:

```
1  -- Add user logic here
2  bit1 <= slv_reg0(0);
3  bit2 <= slv_reg1(0);
4  slv_reg2(0) <= andgate;
5  slv_reg3(0) <= orgate;
6  slv_reg4(0) <= notgate;
7  slv_reg5(0) <= xorgate;
8  -- User logic ends
```

Then I need to extend the wrapper component. I start by adding the types generated by SME:

```
1  -- library SYSTEM_TYPES;
2  use work.SYSTEM_TYPES.ALL;
3
4  -- library CUSTOM_TYPES;
5  use work.CUSTOM_TYPES.ALL;
```

Then, I extend the component definition of the registers component to have the added ports:

```
1  port (
2      bit1 : out std_logic;
3      bit2 : out std_logic;
4      andgate : in std_logic;
5      orgate  : in std_logic;
6      notgate : in std_logic;
7      xorgate : in std_logic;
8      S_AXI_ACLK  : in std_logic;
```

Then I add the component definition of the Logic Gates project, along with the signals going from the Logic Gates to the registers component. For the ports in the component definition, I add the ports specified in the SME generated top-level file:

```
component LogicGates_export is
    port(

    -- Top-level bus Input signals
    Input_bit1: in T_SYSTEM_BOOL;
    Input_bit2: in T_SYSTEM_BOOL;

    -- Top-level bus Output signals
    Output_And: out T_SYSTEM_BOOL;
    Output_Or: out T_SYSTEM_BOOL;
    Output_Not: out T_SYSTEM_BOOL;
    Output_Xor: out T_SYSTEM_BOOL;


    -- User defined signals here
    -- #### USER-DATA-ENTITYSIGNALS-START
    -- #### USER-DATA-ENTITYSIGNALS-END

    -- Reset signal
    RST : in Std_logic;

    -- Clock signal
    CLK : in Std_logic
  );
end component LogicGates_export;
signal bit1 : std_logic;
signal bit2 : std_logic;

signal gates_and : std_logic;
signal gates_or  : std_logic;
signal gates_not : std_logic;
signal gates_xor : std_logic;
```

Then I map the signals to the instantiation of the registers component

```
port map (
    bit1 => bit1,
    bit2 => bit2,
    andgate => gates_and,
    orgate  => gates_or,
    notgate => gates_not,
    xorgate => gates_xor,
    S_AXI_ACLK  => s00_axi_aclk,
```

Finally, the Logic Gates component needs to be instantiated. The clock and reset signals uses the AXI clock and reset. Note: the AXI reset signal is inverted compared to the signal expected by the SME generated code. I.e. AXI resets when the signal is `0` and SME resets when the signal is `1`.

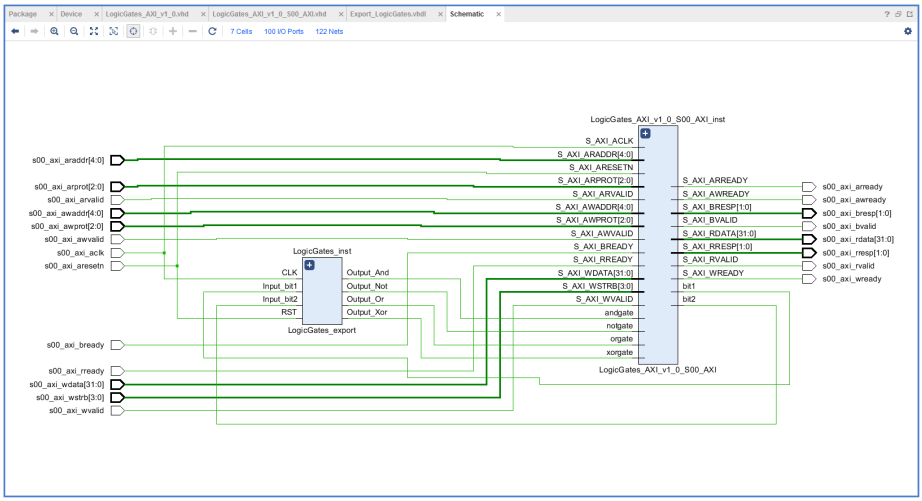

Figure 26: The RTL schematic of the Logic Gates with AXI interface.

```
1  -- Add user logic here
2  LogicGates_inst : LogicGates_export
3  port map (
4      Input_bit1 => bit1,
5      Input_bit2 => bit2,
6      Output_And => gates_and,
7      Output_Or  => gates_or,
8      Output_Not => gates_not,
9      Output_Xor => gates_xor,
10     RST => s00_axi_aresetn,
11     CLK => s00_axi_aclk
12 );
13 -- User logic ends
```

Now the Logic Gates project have been wrapped in an AXI interface. The RTL schematic can be seen in Figure 26. Then I package the IP, which is then available in the IP catalog. In the original project, I create a block design. It needs two components: the processing system and the Logic Gates with AXI interface. Upon adding the processing system, Vivado gives the option to run block and connection automation. The complete block design can be seen in Figure 27. The next step is to create a HDL wrapper from the block design, and then generate the bitstream. The custom types has to be added to the project, otherwise the IP cannot be synthesized, as it cannot recognize the types. When the bitstream has been generated, the hardware is exported to the Xilinx SDK. Note: the bitstream must be included. The implemented Logic Gates with an AXI interface has a maximum utilization of 1%, which is the Logic. It has an approximate power consumption of 1.535 watts. This however, is due to the ARM processor, which uses 1.529 watts. I.e. the Logic Gates with the AXI interface has an approximate power consumption of $1.535 - 1.529 = 0.006$ Watts. The comparison to the reported values of the other implementations can be seen in Table 7.

In the SDK, I create a new Application project. I choose the "Hello

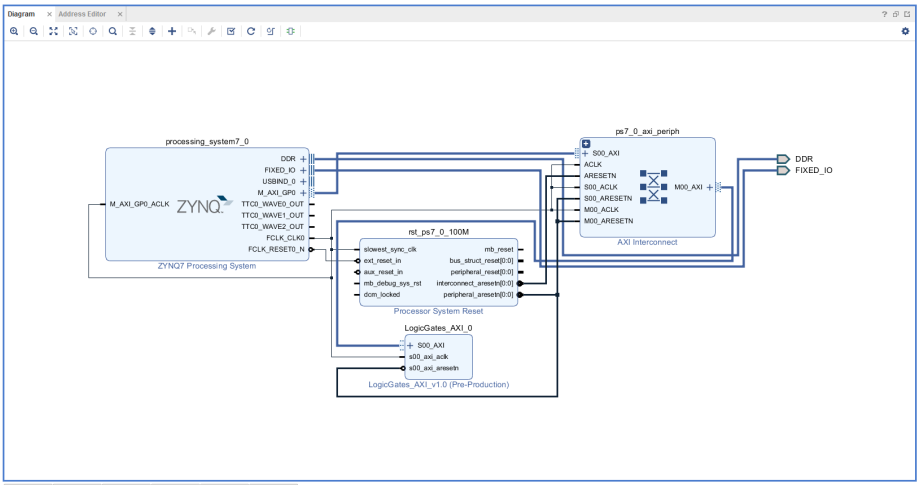

Figure 27: The finished block design contaning the processing system and the Logic Gates AXI IPs.

World" example project, and call it "Tester". The new code is added to `Tester/src/helloworld.c`.

Creating the sample project will also create a Board Support Package (BSP) called "Tester_bsp". This BSP contains all the files related to the exported hardware. The file I am most interested in is `Tester_bsp/ps7_cortex9_0/libsrc/LogicGates_AXI_v1_0/src/LogicGates_AXI.h`. This file contains all the functions for reading and writing to the AXI registers. I also need `xparameters.h`, as it contains the base address for the AXI interface. Finally, I also need `xil_io.h` as it contains the types returned byt he read and write functions.

```
1 #include "xparameters.h"
2 #include "LogicGates_AXI.h"
3 #include "xil_io.h"
```

Now, the default names given by Vivado and the SDK are a bit long and the registers are numbered. Therefore, I give them shorter and more appropriate names

```
1 int base = XPAR_LOGICGATES_AXI_0_S00_AXI_BASEADDR;
2 int bit1 = LOGICGATES_AXI_S00_AXI_SLV_REG0_OFFSET;
3 int bit2 = LOGICGATES_AXI_S00_AXI_SLV_REG1_OFFSET;
4 int and = LOGICGATES_AXI_S00_AXI_SLV_REG2_OFFSET;
5 int or  = LOGICGATES_AXI_S00_AXI_SLV_REG3_OFFSET;
6 int not = LOGICGATES_AXI_S00_AXI_SLV_REG4_OFFSET;
7 int xor = LOGICGATES_AXI_S00_AXI_SLV_REG5_OFFSET;
```

To verify the hardware, I write all combinations of inputs to the registers, and verify that it outputs the same as the truth table specified in Table 1. I write two functions: one for writing to the two input registers and one for printing all of the registers.

```
void print_regs() {
    xil_printf("%d␣%d␣|␣%d␣%d␣%d␣%d\n",
            LOGICGATES_AXI_mReadReg(base, bit1),
            LOGICGATES_AXI_mReadReg(base, bit2),
            LOGICGATES_AXI_mReadReg(base, and),
            LOGICGATES_AXI_mReadReg(base, or),
            LOGICGATES_AXI_mReadReg(base, not),
            LOGICGATES_AXI_mReadReg(base, xor));
}

void write_regs(int bit1_data, int bit2_data) {
    LOGICGATES_AXI_mWriteReg(base, bit1, bit1_data);
    LOGICGATES_AXI_mWriteReg(base, bit2, bit2_data);
}
```

Finally, the `main()` function should initialize the platform, write and read registers, and finally cleanup the platform.

```
int main()
{
    init_platform();

    write_regs(0,0);
    print_regs();

    write_regs(0,1);
    print_regs();

    write_regs(1,0);
    print_regs();

    write_regs(1,1);
    print_regs();

    cleanup_platform();
    return 0;
}
```

After programming the FPGA and programming the ARM, I get the following output:

```
0 0 | 0 0 1 0
0 1 | 0 1 1 1
1 0 | 0 1 0 1
1 1 | 1 1 0 0
```

I have succesfully synthesized, placed and routed the VHDL generated by SME. Furthermore, I have shown how to interface with the implemented hardware, by adding an AXI interface to the design.

6.3 Single cycle MIPS processor

Now I am going to implement the single cycle MIPS processor. The first step is to generate the VHDL from SME. The SME code needs a few adjustments to do this:

- Enumerations should be complete. E.g. if a `switch` is performed on a 5-bit number, the enumeration should contain 256 elements. Another possible fix is to exchange the switch statement with a series of `else if`. However, the `switch` is favored, as Vivado interprets it to a large multiplexor, where the series of `else if`'s are interpreted as a series of multiplexors.
- The HI and LO registers of the ALU should be merged into a single 64-bit register. This is due to the later synthesization connecting subcomponents together, creating a bus where two subcomponents can write to, which is not allowed on real hardware. It also tries to connect some components, which are not allowed.

Once they have been handled, the SME code generates VHDL code. However, there is a few things, that SME cannot transpile, which must do by hand. In the Splitter, where it casts a number to an enumeration:

```
1 -- C#
2 Opcodes opcode = (Opcodes)((tmp >> 26) & 0x3F);
3 Funcs funct = (Funcs)(tmp & 0x3F);
4
5 -- Hand written VHDL
6 ControlIn_opcode <= SingleCycleMIPS_Opcodes'VAL(TO_INTEGER(UNSIGNED(instruction
      (31 downto 26))));
7 ALUFunct_val <= SingleCycleMIPS_Funcs'VAL(TO_INTEGER(UNSIGNED(instruction(5
      downto 0))));
```

Furthermore, SME is not able to handle nested classes. E.g. I put each of the pipeline stages into classes, even in the single cycle processor. To make SME able to transpile again, the top class should be removed.

```
1  -- Before
2  namespace SingleCycleMIPS
3  {
4      public class IF
5      {
6          public class PC : SimpleProcess
7          {
8              ...
9
10 -- After
11 namespace SingleCycleMIPS
12 {
13     public class PC : SimpleProcess
14     {
15         ...
```

Now both `ghdl` and Vivado behavorial simulation works. However, there is a problem with the Register File and the Memory. Both of these processes were unclocked in the SME simulation, i.e. both combinatorial. This is not possible on actual hardware, because in between clock ticks, values can attain any value in a few picoseconds. This means that it is possible, that random values can be written to random adresses at random times. To solve this, they have to be clocked.

As with the initial description of the Register File, I must be careful. I want to make a true single cycle processor. However, making both of these clocked in

the SME simulation, makes it pipelined along some paths. This creates problems as the remaining paths are not pipelined. Furthermore, it would no longer be truly single cycle. To handle this, I make each of them clocked, but modify the VHDL.

I start with the register file. Before, the Register File was handled by having a write buffer. Now, because the Register File itself becomes clocked, the Write buffer no longer needs to be clocked, i.e. it should just pass its input along in a combinatorial fashion. However, now I have to change the Register File. Before it wrote before reading, but now it should handle the write in the same cycle. This is done by making the Register File read values when the clock is high, and write values when the clock is low.

Then the Memory has to be handled. The problem is that invalid writes can be sent to Memory. As with the Register File, I make it clocked. However, I cannot have it activate on the rising edge of the clock, as the data from the Instruction Memory have not reached the Memory at this time. As such, I make it activate on the falling edge of the clock. This ensures, that the instruction have reached the Memory, when it is needed. Making a process activate on the falling edge cannot be described in SME, and must as such be described in the VHDL process. Note: making the memory activate on the falling edge puts more constraint on the paths, as they now have to reach the Memory in half a clock cycle, compared to the full cycle before. The same goes from the Memory to the Register File. As such, the clockrate likely has to be lowered.

Now, the Single Cycle processor should be able to be synthesized, placed and routed. When entering the clocking constraints, I was able to clock the processor at 5 mhz. Once it has been implemented, Vivado can report how much of the logic on the FPGA the implemented design uses. The placed and routed Single Cycle processor uses 14% of the available logic on the ZedBoard. The Single Cycle processor has an approximate power consumption of 0.001 watts. The comparison to the reported values of the other implementations can be seen in Table 7.

However, there is an additional problem. The Instruction Memory and the Memory, are both implemented as registers, within the FPGA. By doing this, the capacity of the Instruction Memory and the Memory have been lowered. An FPGA contains component called block ram. These are memory blocks, which contain more memory than the registers. I cannot use these however, due to them having a delay of a single clock cycle. I.e. once the command have been issued to the block ram, it will be performed in the next clock cycle. This does not match the semantics of a single cycle processor, as the instruction can no longer be performed in a single clock cycle. As such, for the single cycle processor, I have used registers. This is not going to be a problem in the pipelined processor, as the data is not expected to be ready until the following clock cycle.

Now, as with the Logic Gates, I want to be able to communicate with the processor, by adding an AXI interface. I have chosen to have three AXI interfaces: Instruction Memory, Memory and Control. The Instruction Memory and Memory both contain 128 registers and should function as the original components. The Control contains four registers: reset, instruction count, running and clocks. The reset is used for resetting the processor, and is connected to the processors `RST` signal. The instruction count is the number of instructions in the Instruction Memory, and is used to indicate whether or not the processor

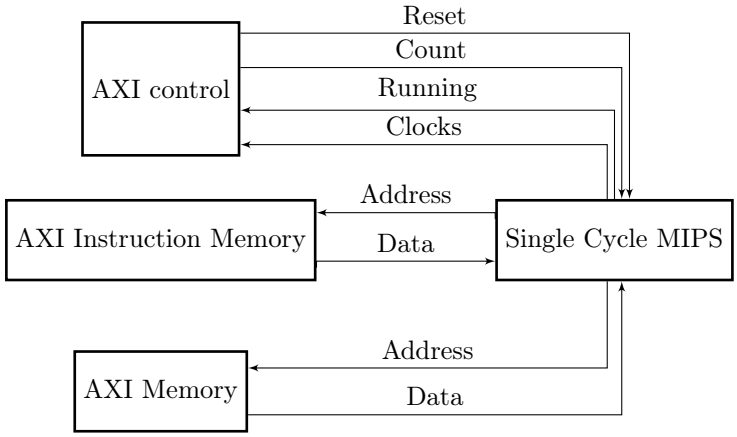

Figure 28: Simplistic overview of how the single cycle MIPS processor should be connected to the AXI registers.

is done executing its program. Running is used to indicate that the processor is running. The clocks is the number of clocks the processor have run for since the last reset.

All of the control signals, except for the reset, are connected to the Instruction Memory. The Instruction Memory now checks if the PC is larger than the instruction count, in which case it is no longer running. Otherwise, it it forwards the PC to the registers in the instruction memory AXI. This axi then returns the instruction at the given address. Finally, the Instruction Memory should increment the clocks, if it is running.

The Memory unit behaves in the same manner as the Instruction Memory. I.e. it forwards the request to the AXI registers, who are now in control of the memory. A simplistic overview of how they should be connected can be seen in Figure 28.

Once the AXI IP have been generated, I integrate it into a block design. The procedure is the same, as with the Logic Gates. I do need to add an additional IP: the clocking wizard. This IP takes an input clock signal and either multiplies it, or divides it. In this case, I need to divide it. The ZedBoard has a 100 mhz clock signal, which the FPGA can use. However, as stated before, I know that the single cycle MIPS processor could be clocked at 5 mhz. As such, the clocking wizard takes the 100 mhz signal and produce a 5 mhz signal.

Once that is in place, I synthesize, place, route and generate bitstream. The Single Cycle processor with the AXI interface uses 22% of the logic available on the ZedBoard. The Single Cycle processor has an approximate power consumption of 0.147 watts (excluding the ARM processor). The comparison to the reported values of the other implementations can be seen in Table 7.

Writing the driver for the processor is straightforward:

1. Set the control reset register to 1. I.e. resetting the processor.
2. Load the instructions into the instruction memory registers.
3. Set the control instruction count register to the amount of instructions.

	FPGA		SME		
	#CT	time (ms)	#CT	time(ms)	speedup
$n = 5$	718	∼0.1436	719	516	3593.14
$n = 10$	22572	∼4.5144	22574	13012	2882.22
$n = 20$	23068776	∼4613.7552	NA	NA	NA

Table 6: Performance of the Towers of hanoi program with different n values. CT is clock ticks. The SME benchmark was performed on a laptop with an Intel Core i5-5300U (2.3 GHz). The FPGA benchmark was performed on a Zynq-7000 All Programmable SoC XC7Z020-CLG484-1. The MIPS processor implemented on the FPGA was clocked to 5 MHz. Towers of hanoi was chosen due to it being easy to scale up.

4. Set the control reset register to 0. I.e. starting the processor.
5. Check that the control running register is set to 1.
6. Wait for the control running register to become 0.
7. Now the processor have run. The amount of clock ticks it took is in the control clocks register, and the result should be in memory (depending on the program of course).

The performance of the VHDL implementation on actual hardware, compared to the SME simulation of the single cycle processor, can be seen in Table 6.

6.4 Pipelined MIPS processor

Many of the same steps as with the Single Cycle processor are required for implementing the Pipelined processor on an FPGA. However, there are a few differences:

- The Register File should have the same read/write order as in the original SME implementation. I.e. it should write first, then read. Furthermore, it should now perform all of its actions, on the falling edge of the clock.
- Within the `switch` statement of the ALU, all the cases which write to the HILO register, should only write on the falling edge of the clock.
- The Memory should perform all its computations on the falling edge of the clock.

Other than these differences, the Pipelined processor can be implemented on the FPGA, in the same manner as with the Single Cycle processor. I was able to clock it at 68.9776 MHz. As such, pipelining the processor yielded a ∼1379% speedup. The placed and routed Pipelined processor uses 24% of the logic available on the ZedBoard. The Pipelined processor has an approximate power consumption of 0.019 watts. The comparison to the reported values of the other implementations can be seen in Table 7.

Implementing Block RAM

Another advantage gained by pipelining the processor, is the option to use block RAM as memory instead of registers. In the Single Cycle processor, the instruction had to be read, computed and possibly go through memory in the same clock cycle. However, block RAM has a delay of one clock cycle, before the data is ready. As such, it could not be used in the Single Cycle processor. However, in the Pipelined processor, both the data from the Instruction Memory and Memory, are not needed until the following clock cycle. As such, I can use block RAM to get more memory.

I start by adding two new SME processes: InstructionBRAM and MemoryBRAM. Both of these are `SimulationProcesse`s. As such, they are both clocked processes and they will not be transpiled. Then I add new busses, which go from the old processes, to the new processes. The logic in the old processes just forwards the their input to these new busses. The logic in the new processes is the same as the previous logic in the old processes. Finally, the output which previously entered the following pipe, should now go straight to the receipients. This is due to the delay in the Block Memory, which gives the same delay as the pipes did.

Once the SME simulation has run and transpiled, the VHDL is entered in a new project in Vivado. Within this project, I create a new block design. This block design consists of four modules: the Pipelined processor, two Block Memory generators and a constant. The two Block Memory modules are connected to the processor, and the constant is connected to the two Block Memory modules, in order to keeping the ports used by the processor always enabled. The processor and the Block Memory generators share the same clock and reset signal.

For easy testing the block design, the Instruction Memory can be initialized with the use of a `coe` file. The format of the `coe` file is very simple: first there is a line specifying the radix (base) of the numbers, followed by an array describing the data. E.g. some of the `coe` file for the fibonacci program would be

```
1 memory_initialization_radix=16;
2 memory_initialization_vector=
3 34080001,
4 34090001,
5 ...
6 00000000;
```

I chose to still have dedicated instruction and data memory, as I then did not have to change any of the compiled programs and due to it resembling the L1 cache of a modern processor. Furthermore, this design makes it easier to extend with an AXD interface, as Vivado has a Block Memory AXI Controller IP. Each of the Block Memory Modules can have a maximum of two ports. As such, if I had gone with the single Block Memory approach, the two ports would already be in use by the processor.

Once the block design have been constructed, it is time to synthesize, place and route the design. However, there is a problem when doing this. In the first version I made, the VHDL had the `EX_Pipe_JALOut` bus, which both goes to internal processes and outside the network as an top-level bus. Vivado did not like this particular setup. To solve it, I made an internal signal in VHDL, to

Design	Clockrate	Memory	Utilization	Power
Logic Gates	N/A	N/A	4	0.001
Logic Gates AXI	100	0.02	1	0.006
Single Cycle	5	0.19	14	0.001
Single Cycle AXI	5	1	22	0.147
Pipelined	68.98	0.19	24	0.019
Pipelined BRAM	71.43	64	7	0.041

Table 7: Comparison table of the reported values of the different implemented designs. All of them have been made with the ZedBoard as the target board. Clockrate is in megahertz. Memory is in kilobytes. Utilization is the maximum utilization metric in percent. Power is in watts.

which the processes write and read from. Futhermore, the signal also goes to the top-level output bus. This does not change the structure of the network, but satisfies Vivado.

Once the block design could be synthesized, placed and routed, I was able to clock the processor at 71.429 MHz. The increased clock is due to the Block Memory. The Block Memory is rated to handle severel hundreds MHz [20] and are as such not even close to being a bottleneck in my design. When the Instruction Memory and Memory consisted of registers, the path through the registers was longer, than the path to the Block Memory. Furthermore, since both have been moved to Block Memory, the logic used by the project have decreased to use 6% of the available logic on the ZedBoard. However, the processor now uses 7% of the available Block Memory on the Zedboard, making it the new bottleneck for having multiple cores. The Pipelined processor using Block Memory has an approximate power consumption of 0.041 watts. The comparison to the reported values of the other implementations can be seen in Table 7.

7 Conclusion

I have succesfully implemented a MIPS processor in the SME programming model, in the timeframe of ∼2 months. This shows that the SME model is suitable for designing hardware, with the prior knowledge of the CSP model, hardware organization and software development.

Furthermore, by extending the accepted instruction set, and by pipelining the processor, I have shown that extending the initial design in order to gain increased performance or additional functionality, is possible.

I have also shown that SME indeed can be transpiled into functioning VHDL, with a small amount of extra work. The extra actions are however simple enough, such that they could be implemented in a future version of SME.

As such, with the material in this thesis, a computer science student, such as myself, should be able to construct specialized hardware by using SME, in the timeframe of a university course.

8 Future work

The initial step would be to add an AXI interface to the Pipelined processor. The approach is the same as with the single cycle processor. It would be interesting to see whether or not the increased clock is applicable to the additional AXI interface.

Inspired by the manycore architecture of the SW26010 processors in the TaihuLight supercomputer [21], I want to see how many cores I am able to fit onto a single FPGA. By doing this, I also want to introduce scratchpad memory, instead of focusing on the traditional memory hierarchy, as this is what is being used in the SW26010.

It would also be interesting to extend the processor core, to a superscalar processor, i.e. by introducing multiple execution paths. This could both be by introducing additional ALUs, e.g. floating point, or by introducing coprocessors, which are specialized for certain tasks, such as the SME network of a student project, which solved partial differential equations and fast fourier transforms.

A final suggestion would be to extend the processor with the required components, such that a minimal operating system could be compiled and run on the processor. This would be useful in specialized hardware, if a device should run purely in its own environment.

References

[1] Carl-Johannes Johnsen. Implementing a mips processor using sme, 2017.

[2] Maskinarkitektur (ARK). http://kurser.ku.dk/course/ndaa04009u/2013-2014. [Online; accessed 25-Apr-2017].

[3] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[4] K. Skovhede and B. Vinter. Building hardware from C# models. In *FSP 2016; Third International Workshop on FPGAs for Software Programmers*, pages 1–9, Aug 2016.

[5] Brian Vinter and Kenneth Skovhede. *Synchronous Message Exchange for Hardware Designs*. Open Channel Publishing Ltd, 2014.

[6] Brian Vinter and Kenneth Skovhede. *Bus Centric Synchronous Message Exchange for Hardware Designs*. 8 2015.

[7] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[8] Truls Asheim, Kenneth Skovhede, and Brian Vinter. *VHDL Generation From Python Synchronous Message Exchange Networks*. Open Channel Publishing Ltd., 2016.

[9] Communicating process architectures 2017. http://wotug.org/cpa2017/. [Online; accessed 05-August-2017].

[10] The mipsfpga project. https://github.com/MIPSfpga. [Online; accessed 05-August-2017].

[11] Pyrope, a modern hardware description language for live synthesis flow. https://masc.soe.ucsc.edu/pyrope.html#1. [Online; accessed 05-August-2017].

[12] Carl-Johannes Johnsen. Sme-mips. https://github.com/carljohnsen/SME-MIPS, 2017. [Online; accessed 13-July-2017].

[13] Introduction to Combinational Logic Functions. https://www.allaboutcircuits.com/textbook/digital/chpt-9/combinational-logic-functions/. [Online; accessed 25-Apr-2017].

[14] Missouri State University. Mars (mips assembler and runtime simulator). http://courses.missouristate.edu/KenVollmar/mars/, 2014. [Online; accessed 03-May-2017].

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[16] Writing a Towers of Hanoi program. https://www.cs.cmu.edu/~cburch/survey/recurse/hanoiimpl.html. [Online; accessed 25-Apr-2017].

[17] Ghdl. https://github.com/tgingold/ghdl. [Online; accessed 25-July-2017].

[18] Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed 25-July-2017].

[19] Axi reference guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. [Online; accessed 31-July-2017].

[20] Xilinx. Block memory generator v8.3 - logicore ip product guide. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf. [Online; accessed 07-August-2017.

[21] Jack Dongarra. Report on the Sunway TaihuLight System. http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf, 2016. [Online; accessed 25-Apr-2017].

A Guide til strukturen på github!