Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

27th July 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 Introduction

Background and terminology for:

CSP - hele konceptet med processer der kommunikerer

SME - processer, bus og clock

Machine architecture - hvertfald hvad word, bit og clock er for noget.

DIKU course

VHDL

C# - især hvorfor der er `uint` over det hele.

beskriv hvordan figurene fungerer, i.e. box/circle = process, black wire = data, blue wire = control, green wire = clock

Beskriv hvad MARS er, og hvordan det virker.

2 Getting started with SME

2.1 Installing required software for SME

Muligvis ryk alt det her ned i appendix, da det mest bare er fifs? monodevelop git

2.2 Writing first SME program

2.3 C# and bit hacking

2.4 Translating first SME program into VHDL

2.5 Running and verifying VHDL

3 Basic logic circuits

In this section, we will be looking at some basic combinatorial circuits. We use this as an entry point for hardware design, both due to the ARK course[4] at DIKU, and due to the 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. We are only going to cover a subset of appendix C, as it should be sufficient as an introduction to connecting simpler components together into a more complex network.

We start by looking at some logic gates, which implement some basic boolean functions. Then we 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 half adder followed by $n-1$ full adders.

For each of these combinatorial circuits, we go through the theory behind it, describe the procedure of translating the theory into an SME network and finally how to test and verify the SME networks.

3.1 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 based upon the input values. We are going to implement the following logic gates:

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

`OR` - outputs `1` if 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 (except for `NOT`, which only uses 1 bit) can be seen in Table 1.

Implementation

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. We 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. Furthermore, we only need one output bus, as each process writes to its own bit within it. Each process takes the two bits from the input

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`.

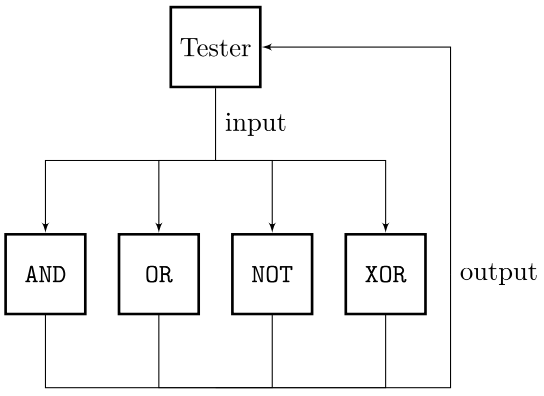

Figure 1: The structure of the test of the logic gates

bus, computes their respective logical function and sends the result out onto its bit on the output bus.

We 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, but we want to make each process as simple as possible, when constructing the processor, as this makes arguing for correctness simpler.

Testing

To ensure that the processes functions as expected, we construct a tester process, which sets the bits on the input bus to all of the values in the truth table, and checks whether or not each process puts the expected value from the truth table onto their bit on the output bus. We use this testing technique, as it tests the functionality of the processes, without knowing how they work internally. We could also just have printed the output values, and verified the results our selves, however automating this reduces the time needed for verification. How the processes are connected can be seen in Figure 1.

3.2 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

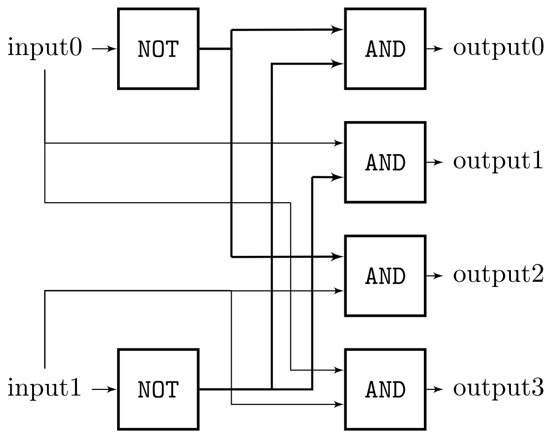

Figure 2: An 2-bit decoder made by `AND` and `NOT` gates.

are zero indexed, 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[5]. We need to have n `NOT` gates, and 2^n `AND` gates. For each input, we split it into two, and send the copy to a `NOT` gate. Then for each output, we attach an `AND` gate, and give it inputs corresponding to the binary representation of the number E.g. if we get the number 5, the binary representation is 101, i.e. the 5th `AND` gate gets input from Bit0, `NOT` Bit1 and Bit2. An example of a 2-bit decoder can be seen in Figure 2.

Implementation

To implement a 2-bit decoder, we just need to connect logic gate processes, which we already have. How to connect a 2-bit decoder can be seen in Figure 2.

We could have made the decoder a single process, but we are trying to emphasize connecting multiple processes together, in order to gain functionality.

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

Testing

To test the 2-bit decoder, we follow the same procedure as with the logic gates. We construct a tester process, which sends input to the logic gates, and verifies

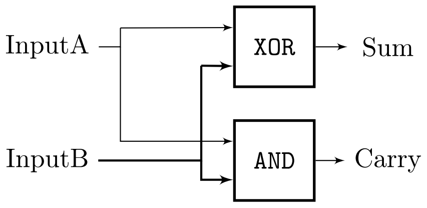

Figure 3: An half adder composed of `XOR` and `AND` gates

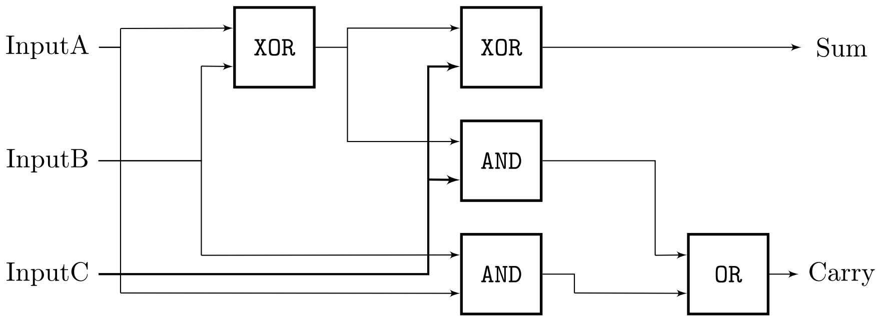

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates

the output from the `AND` gates. This is also what we need to do with the n-bit decoder. However as with the n-bit decoder, we are going to need C# templates, as the tester process also needs a variable amount of busses. Each of the two tester processes should send all possible inputs as input.

3.3 Adder

An adder is an 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[5]. An n-bit adder is a chain of two major components: an half adder and a full adder.

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

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

To make the n-bit adder, we have to use C# templates like we did when constructing the n-bit decoder. We 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 carrys have their own bus. Then we 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

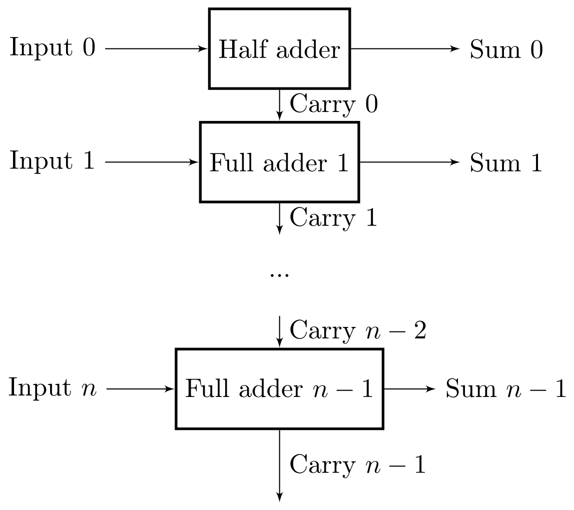

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

0, and outputs the carry on carry 0. Then we construct $n-1$ full adders, where full adder i (1-indexed) has the inputs: input A bit i, input B bit i and carry $i-1$. Each full adder has output i and carry i as output.

Testing

To test the adder, we construct a tester process as before. We start by testing the two subcomponents, the half and the full adder, by giving each every possible input, and verifying the output. For the n-bit adder, we make a function that takes an integer as input, and sends it along the corresponding input wires. We also make a similar function that takes the values from the output wires and packs it into an integer. This makes the test more human readable, and reduces the chance of failure, when hand encoding/decoding binary numbers. There are two tests: the simplest addition: 2+2, adding two numbers together that produce an overflow and finally a series of adding random numbers together.

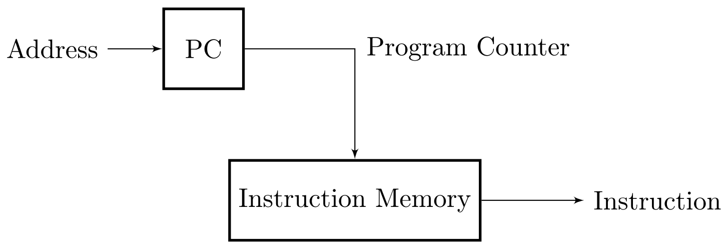

Figure 6: The Instruction Memory

4 Core components

In this section, we will be looking at the major components of the MIPS processor. We look at the components before wiring them together into a processor, as it keeps each step more isolated and simple.

For each component, we will go through the theory of the component, then we will look at translating theory into an SME implementation, and finally the verification of the implementations. When we are verifying the implementation, we should 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 book[3], and should be the order of implementation.

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

4.1 Instruction Memory

The first component that we need is a memory unit to hold the instructions of a program, and to supply instructions at a given address. We 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, PC and their connections can be seen in Figure 6.

Implementation & Testing

Both the PC and the Instruction Memory should be SME processes. We could have combined the two components into a single SME process, but we are trying to keep each process as simple as possible. The input bus for the PC, the Program Counter bus and the instruction bus, should all contain an `uint` value.

The PC should have a single `uint` value holding the current address, and on each tick, it should output its stored value, and store the new input value. It should also be clocked, as this is the starting point of every instruction.

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, we 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 bytes, and finally pack all 4 bytes together into an `uint`, and place it on the instruction bus.

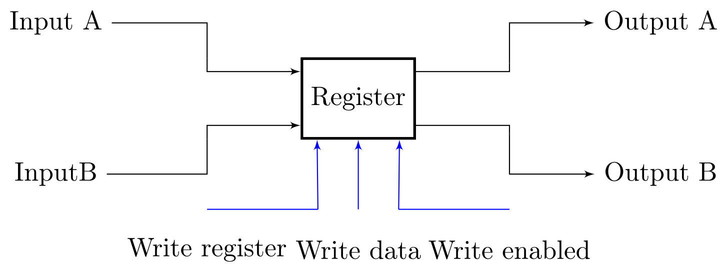

Figure 7: The register file

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

Testing the Instruction Memory is very trivial. The program should be hardcoded into the `byte` array in the memory. The tester process should then send values to the PC, and verify that the output on the instruction bus, matches the value at the given address.

4.2 Register file

The MIPS processor has 32 general-purpose 32-bit registers, which is stored in a structure called the Register File. Each of the registers can be read from or written to, except for register 0, which is immutable and always 0. It is the first step 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 Address A, Read Address 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 Address A, and outputs it on Output A, and vice versa for Read Address 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.

We need to be careful of the order in which we read and write from the Register File. We 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 get the newly written value. This is easy to fix in the single cycle processor, as we just need to write before reading. We cannot do it in the reverse order, as the Register File might then output old values. The Register File and its inputs and outputs can be seen in Figure 7.

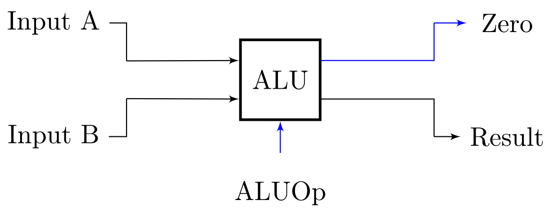

Figure 8: The ALU

Implementation & Testing

The Register File should be an SME process. The collection of registers should be an `uint` array of length 32. The address busses should all contain an `byte` value, as the number of addressable registers never exceed $2^8 = 256$. The output busses and the Write Data bus should all contain an `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 Address A bus, and output onto the Output A bus, and analougously for the Read Address B bus and the Output B bus.

It should be noted that if the write address is 0, then the write should be ignored, as register 0 is immutable.

Testing the register file is very trivial. We start by sending some values on the write data bus, along with some addresses and the `RegWrite` flag set.

Then we just try to send some addresses on the Read address A and Read address B buses, and verify that the register file outputs the values stored at these addresses. It is also important to verify the behaviour of register zero.

4.3 ALU

The ALU (Arithmetic Logic Unit) is the part of the processor, which makes the actual computation. It takes three inputs: InputA, InputB 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 overview of the component and its inputs and outputs can be seen in Figure 8.

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 a flag indicating whether the result was 0.

Implementation & Testing

To implement the ALU, we 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. We could have followed the ALU Opcode given in

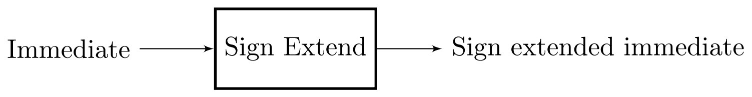

Figure 9: The Sign Extend

the book[3], however it states that its encoding of the ALU Opcode is generated using a CAD tool, and as such we will generate it ourselves too.

We start by implementing the same instructions as the book[3] 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, logical `AND`, logical `OR` and the comparison less than.

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

Constructing the ALU process in SME is straightforward, it reads the ALU Opcode from the ALUOp 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 we do not need to do anything but 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 4.8.

There are no special cases to consider when testing the ALU, we should just test whether or not it can compute the right result, based on the inputs.

4.4 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 can be seen in Figure 9

Implementation & Testing

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

Testing the Sign Extend is straightforward: send values on the input bus, and verify that the same number is on the output bus.

4.5 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

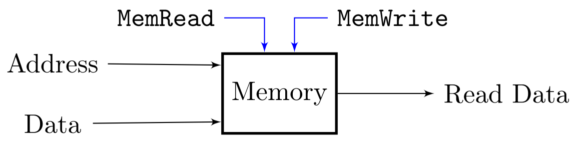

Figure 10: The Memory Unit

a lot more data. The processor can either read or write to the Memory Unit. 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 dividable 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 can be seen in Figure 10.

Implementation & Testing

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

As with the Instruction Memory, we are 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 bytes, 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.

To test the Memory Unit, we should just try to store some values, and check if we can fetch the same values again.

4.6 Splitter

The book[3] 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, we construct our 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

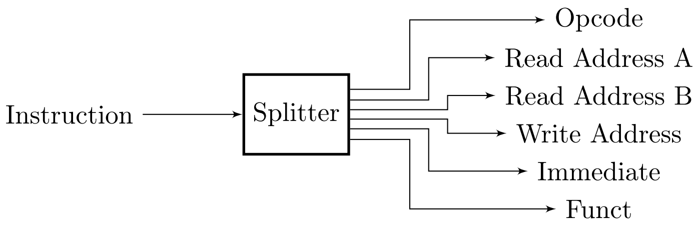

Figure 11: The Splitter

- Read Address B - bits 16-20
- Write Address - bits 11-15
- Immediate - bits 0-15
- Funct - bits 0-5

The Splitter can be seen in Figure 11

Implementation & Testing

The implementation is straightforward: We construct an SME process, which takes the instruction coming from the Instruction Memory, and extract the bits at the indices, by using C# bit hacking. The Input bus should contain an `uint` value. The Immediate bus should contain an `short` value. The rest of the busses should contain an `byte` value. Finally, the process should output the extracted values on the respective output busses.

Testing the Splitter requires no special procedures.

4.7 Jump Unit

The book[3] describes how to implement `beq`, by the use of multiple components. However, we should combine these components into a single unit, the Jump Unit, to simplify the processor.

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. We will be adding support for more branch and jump instructions later.

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 and its internals can be seen in Figure 12.

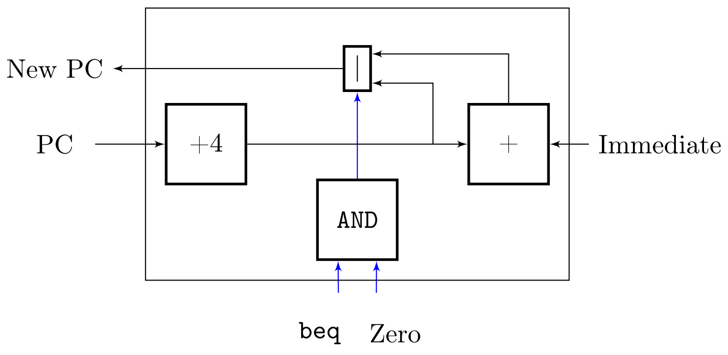

Figure 12: The Jump Unit

Implementation & Testing

We have the all of the logic described in the background. To implement the Jump Unit, we implement all of the logic components as SME processes. We are going to need 4 components: an +4 incrementer, an adder, an AND gate and a multiplexor.

The Incrementer takes the PC bus as input, and produces a single output on the inced bus. The two busses should both contain an `uint` value. On each clock tick, it should take the value from the PC bus, add 4 to it and put it on the inc 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 an `uint` value. On each clock tick, the Adder should add its two inputs 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, we have the multiplexor, which takes the inced, added and anded busses as input, and produces a single output: the new PC. The new PC bus should contain an `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.

We could have made all the logic in C#, however this makes it easier to extend, as each subcomponent is very simple.

We test the Jump Unit by initially verifying that the PC increments as it should, if the two control signals are `0`. Then it should be tested, if the correct branching addresses are constructed.

4.8 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 can be seen in Figure 13.

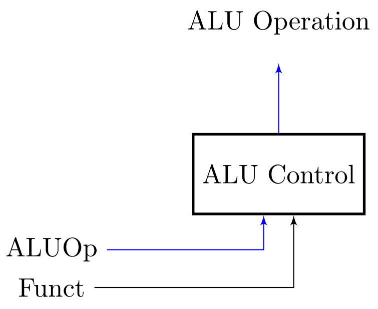

Figure 13: The ALU Control

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

Implementation & Testing

The 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, we construct our own ALU Control, and let the VHDL generator handle the logic generation.

To make the source code more human readable, we should have two additional `enum`s: one for the `ALUOp` code and one for the `funct`. Then we 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.

To test the ALU Control, the tester process should try all possible combinations, as there should not be too many combinations.

4.9 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 which B register to read from.

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.

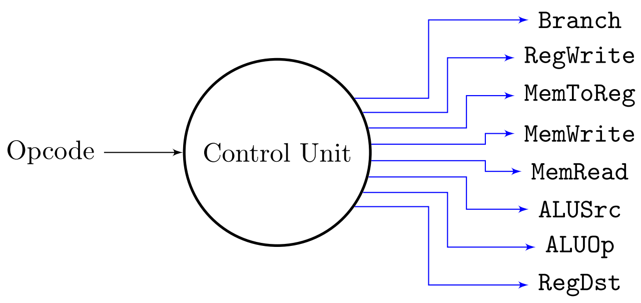

Figure 14: The Control Unit

ALUOp Opcode indicating which operation should be performed in the ALU. It is send 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.

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

Implementation & Testing

As with the ALU Control, the book[3] describes the logic needed to implement this unit. However, again it is not trivial to extend later on, so we construct our own Control Unit, and let the VHDL generator handle the logic generation.

We construct an SME process, which has all of the busses described in the background section. The input opcode bus and the ALUOp code bus should both contain an `byte` value. The rest of the busses should all contain an `bool` value.

As with the ALU Control, we should have an `enum` on the opcode, to make it more human readable. We do not need one for the ALUOp, as the ALU Control already described 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 with the ALU Control, testing the Control Unit should be done by trying every opcode available.

4.10 Write back

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

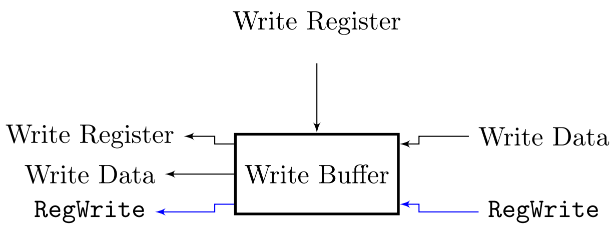

Figure 15: The Write Buffer

Implementation & Testing

Usually in the single cycle MIPS processor, there is nothing special in the Write Back stage. However, in SME we are 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, we 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.

We could also have made one of the previous components clocked, however we do not want to this, as this specific problem will be solved when we pipeline the processor in section 6, and then the Write Buffer can be removed again.

The Write Buffer and its connections can be seen in Figure 15. Testing the Write Buffer is very trivial: verify that it outputs its input from the previous clock tick.

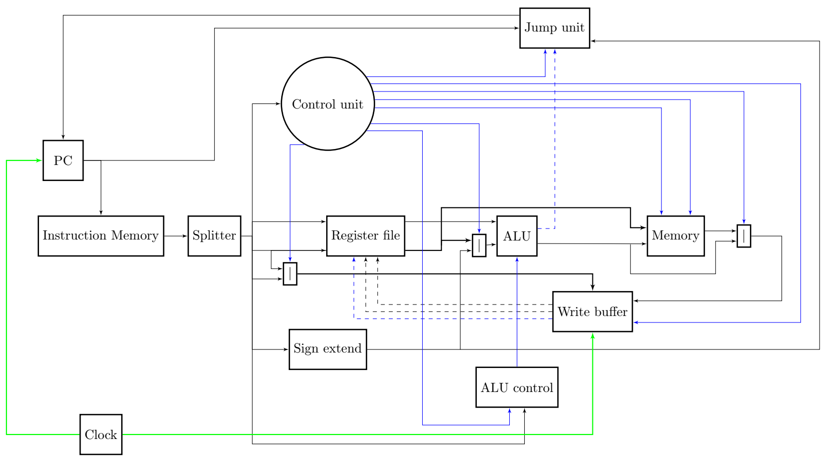

Figure 16: Simple single cycle MIPS processor. The units with | indicate multiplexors. The Clock is not an SME process, but has been added to emphasize which processes that are clocked.

5 Single cycle MIPS processor

In this section, we 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, we will be writing the first program, and compiling it into the processor, and running it.

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

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

5.1 Wiring up the processor

With all the components in place, wiring up the single cycle MIPS processor is straightforward. We just need to declare the busses with the corresponding names, and then SME handles the wiring process. Note that as previously mentioned, the Write Buffer and the PC register should be clocked processes. The wiring of the single cycle MIPS processor can be seen in Figure 16.

5.2 Writing the first program

As mentioned before, the first single cycle MIPS processor should be able to handle `add`, `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 an `byte` array, which is initialized to the hex values from the instruction

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	`and $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. 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.

column in Table 2. Since we do not have any way of feeding values into the Register File at this moment, we are 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.

5.3 Extending the accepted instruction set

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`, and then add the remaining cases in the ALU Control and the ALU. As described in section 4.3, it is important to cast the input `uint` to `int` before making the computation, if the operation is signed.

To test each of these instructions, we can just append them to our initial program.

Immediate instructions

The next instruction we want to add is the `ori` instruction, as this would allow us to feed values into the processor without hardcoding it. While we are doing this, we should also add `addi`, `addiu`, `slti`, `sltiu`, `andi` and `xori`, as these requires the same amount of work.

For the logical immediates, it is important, that the Sign Extend does not sign extend. As such, we 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 we 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 have been extended, we can prepend instructions at the beginning of the program. Furthermore, we can remove our previous hardcoding of initial values in the Register File.

Jump instruction

We are going to introduce another format: the J format. This format is used for executing the `j` instruction. The first step is to extend the Splitter, as it should now send the lower 26 bits 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. Then, it should take the 4 most significant bits from the PC+4, and prepend them to the extended number. Finally, we are 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.

Then the program can be extended with a `j` instruction, in the same manner as the `beq` instruction was tested before, i.e. trying to jump over some instructions.

Branching instructions

Then we are going to add the remaining branching instructions `bne`, `blez` and `bgtz` instructions.

To implement the `bne` instruction, we are going to add another signal from the Control Unit to the Jump Unit. Then we should split the Zero signal into two, where one of the signals goes to a newly added `NOT` gate. Then, we should 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.

TODO `blez`

TODO `bgtz`

Remaining jump instructions

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

We start with `jr`. The instruction is in R format, so we do not know it is `jr`, until it has reached the ALU Control. As such, we are going to need a control signal from the ALU Control to the Jump Unit. We are 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, we are going to need an extra unit following the ALU. In the case of a `jal` instruction, we should store the PC+4 address in register `31` (which is called the `$ra` register). There should be 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.

TODO `jalr`

Shift instructions

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

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, we do not know it is a shifting instruction until it reaches the ALU Control. So to control the new multiplexor, we 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 we are 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, we are also going to need the `mfhi`, `mthi`, `mflo` and `mtlo` instructions.

We start by adding the special registers. Since they are performed in the ALU, we might as well put them there. Then, when we are doing the computation, we should just put 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

5.4 Larger MIPS programs

To test the full single cycle MIPS processor, we are going to implement two programs in MIPS assembly: Quicksort and Towers Of Hanoi. We choose these

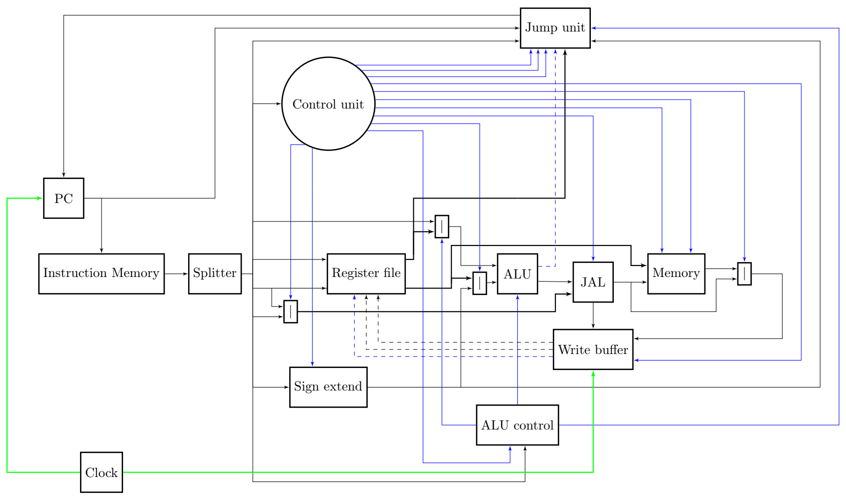

Figure 17: Full single cycle MIPS processor. The units with | indicate multiplexors. Black wires indicate data wires. Blue wires indicate control wires. Green wires indicate clock

two examples, as both are simple to implement, and they do not require anything special from the environment.

We are not going to give the implementation in MIPS assembly, as this would not be readable. Instead, we are going to construct some low level C code, which should be easy translatable into MIPS assembly, and easilier verifiable on a real machine. Once we have made the assembly, we can easily dump it into MARS, which can then produce a ascii hex dump, which we can then paste into our instruction memory.

Quicksort

Quicksort is a sorting algorithm, for doing comparison based sorting. The algorithm is heavily described in the algorithm book[6], and as such we wont go into details about the algorithm. We could have chosen one of the other sorting algorithms from the book, but quicksort should not be too hard to implement.

There are three parts of the Quicksort program: Loading data into memory, the partition function and the quicksort function. We are going to use the partition function from the algorithm book, and the quicksort function, also from the algorithm book, as both are simple to implement.

We 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 our processor implementation produce the same result as MARS.

Loading data into memory We start by loading data into memory, so that our quicksort program has some data to sort. Which numbers we are going to use is not important, rather the amount of numbers, as the quicksort algorithm uses it as argument. We construct a function called `load`, which inserts 6 numbers into the given memory address.

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

The partitioning function Then we implement the `partition` function as described in the algorithm book[6]. Note that statements in the book have been expanded to more closely resemble assembly.

```
int partition(int *a, int p, int r) {
    int x, i, j, tmp1, tmp2, *addr1, *addr2;
    addr1 = a + r;
    x = *(addr1);
    i = p - 1;
    for (j = p; j < r; j++) {
        addr1 = a + j;
        if (*(addr1) <= x) {
            i++;
            addr1 = a + i;
            addr2 = a + j;
            tmp1 = *(addr1);
            tmp2 = *(addr2);
            *(addr1) = tmp2;
            *(addr2) = tmp1;
        }
    }
    addr1 = a + i + 1;
    addr2 = a + r;
    tmp1 = *(addr1);
    tmp2 = *(addr2);
    *(addr1) = tmp2;
    *(addr2) = tmp1;
    return i + 1;
}
```

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

```
void quicksort(int *a, int p, int r) {
    if (p < r) {
        int q = partition(a, p, r);
        quicksort(a, p, q-1);
        quicksort(a, q+1, r);
    }
}
```

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

```
int main() {
    int arr[6], i;
    load(arr);
    quicksort(arr, 0, 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.

Towers Of Hanoi

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 the time.

By searching, we find a pseudo approach to the problem, by using recursion[7]. We are 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, we start by loading data into memory, have a `tower` function, which performs the move, and finally correct way of calling the functions.

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

```
void init(int num, int *from, int *to, int *aux) {
    int i;
    for (i = 0; i < num; i++) {
        *(from+i) = num - i;
        *(to+i) = 0;
        *(aux+i) = 0;
    }
}
```

The tower function This is the `tower` function, which moves the discs from peg to peg. We use the same approach as the pseudo code[7]. As with quicksort, the statements have been expanded to more closely resemble assembly. We make the three arguments as stack pointers to each of the pegs. To preserve each of the pointers, we have pointers to each of them, i.e. pointer pointer.

```
void tower(int num, int **from, int **to, int **aux) {
    int *t, *f;
    if (num == 0) {
        t = *to;
        f = *from;
        f--;
        *t = *f;
        t++;
        *f = 0;
        *to = t;
        *from = f;
    } else {
        tower(num-1, from, aux, to);
        t = *to;
        f = *from;
        f--;
        *t = *f;
        t++;
        *f = 0;
        *to = t;
        *from = f;
        tower(num-1, aux, to, from);
    }
}
```

Calling the algorithm We construct a `main` function, to emphasize how to make the arguments.

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

When the program has finished, the last third of the array should contain the numbers in descending order.

6 Pipelining

In this section, we will be looking at pipelining our single cycle MIPS processor, and handle the problems, which are introduced by pipelining.

We start by going through the background and motivation for pipelining, and then proceed on how to extend our single cycle MIPS processor to have pipes.

As mentioned, pipelining introduces new problems to handle in the processor, and we 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, we will also be writing programs, in order to verify that our processor behaves as specified.

6.1 Introducing the pipes

We have the single cycle MIPS processor, which accepts 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, so in order to increase the clock rate, we must decrease the longest path in the processor. We solve this by introducing pipes.

Pipes are registers in the processor, where we temporarely store all the values computed so far. 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, we divide the processor into stages. We will use the same stages, as proposed in the book[3], 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, we are going to insert a pipe, i.e. we are going to insert 4 pipes. The simplified processor with pipes can be seen in figure 19.

Introducing pipes also introduces additional problems, which we will discuss further in sections 6.2 and 6.3

Implementation

There are two ways to implement pipes in SME: by using clocked busses, or by using clocked processes. If the bus only traverses 2 stages, then we can use clocked busses, as the semantics of a clocked bus is exactly the same as a pipe. However, if the bus traverses more than 2 stages, we are going to need additional processes, as the clocked bus only stores its value for one clock. Furthermore, if the only change to the bus is the given `ClockedBus` attribute, then determining whether or not a bus is part of a pipe can be problematic. As such, we should have the pipe and its busses in its own class, as it then receives its own prefix, and the code then becomes more readable.

Introducing the pipes in this manner is fairly straightforward, as we 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.

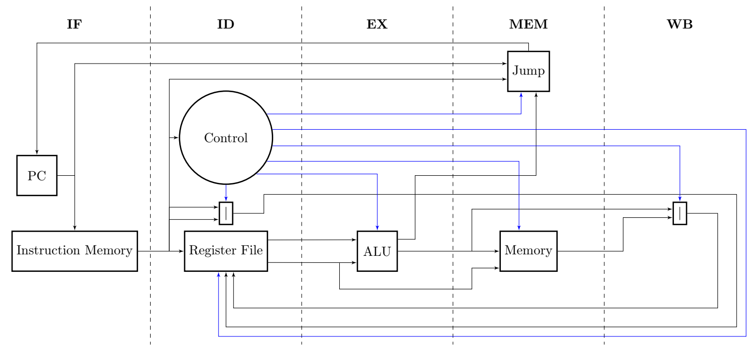

Figure 18: The simplified single cycle processor, and its stages. The stage names are highlighted in **bold**. The stages are divided by dashed lines.

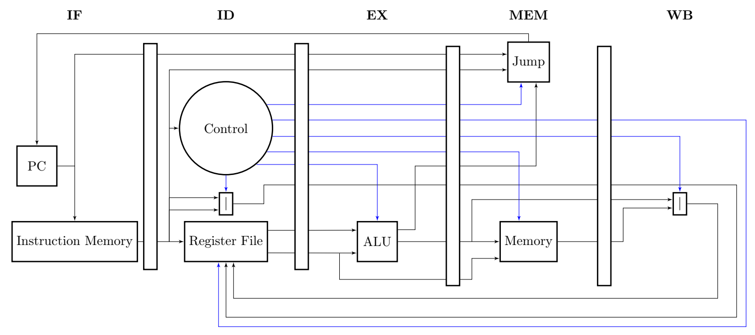

Figure 19: The simplified pipelined processor. The long bars in between each state is the pipes.

Let us take a subset of the IF/ID pipe, as an example. We 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 bus is now piped.

This process can be repeated for all of the required pipes. By following the division of the components, as proposed in the book [3], there is only one really tricky part of pipelining: the Jump Unit. We dont know whether we should 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 we should jump. As such, we 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, we introduced the Write Buffer, in order to remove the cycle from and to the Register File. However, by introducing pipes, we have introduced a new buffer, and thus the Write Buffer can be removed.

Testing

To test the processor, we can use any of the programs, that we have previously written. However, since we have pipelined the processor, we 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.

We are 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, we are going to give pseudo low level C code:

```c
void init(int *arr) {
    *(arr)   = 1;
    *(arr+1) = 1;
}

void loop(int *arr, int n) {
    int i, tmp1, tmp2, tmp3;
    for (i = 0; i < n; i++) {
```

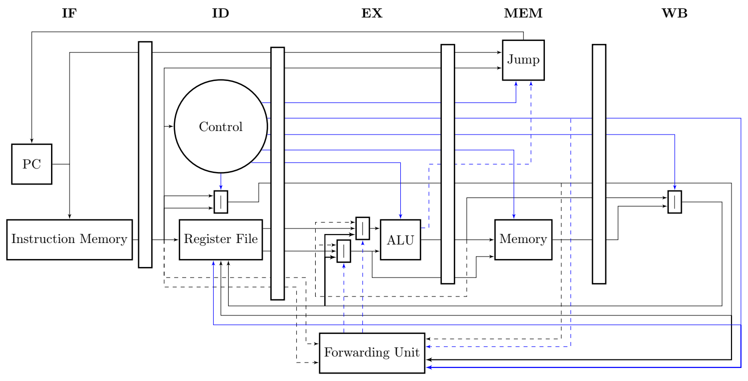

Figure 20: The simplified pipelined processor with forwarding.

```
        tmp1 = *(arr+i);
        tmp2 = *(arr+i+1);
        tmp3 = tmp1 + tmp2;
        *(arr+i+2) = tmp3;
    }
}
```

Note that for verification in C, the size of the array should be $n + 2$, due to initialization. Furthermore, when we port it to MIPS assembly, after each instruction, we should insert 4 `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.

6.2 Forwarding

By introducing pipes, we also introduced data hazards and control hazards. We will go through control hazards in section 6.3. 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.

We 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 6.3. 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 processor with the forwarding unit can be seen in Figure 20.

We 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 there would be several clock cycles, 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: we 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.

6.3 Hazard Detection

As mentioned in section 6.2, we cannot avoid all data hazards with forwarding. This is due to forwarding only forwarding 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. When inserting a bubble in the middle of the pipeline, we have to stall some of the pipes and registers. 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 cycle.

We 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, we 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 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.

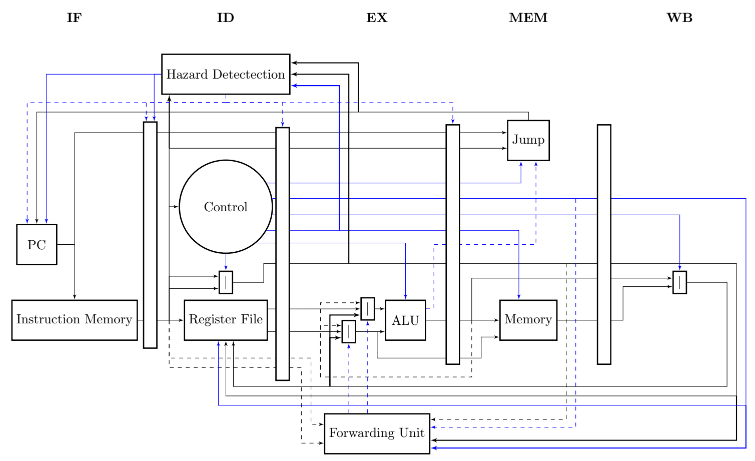

Figure 21: The simplified pipelined processor with forwarding and hazard detection.

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 to be flushed part of the pipeline.

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 of clock ticks, compared to the single cycle processor.

Note: if these programs are tested against the single cycle processor, the performance will be worse, when simulating. This is due to more processes and busses, which gives the simulation more work. There should not be any performance hit, when the processor is run on hardware.

6.4 Branch prediction?

PERFORMANCE HIT! nævn at det sker i simulering TODO

få noget ala linux til at køre?

scratchpad memory som agere cache?

flere kerner?

scalar processor?

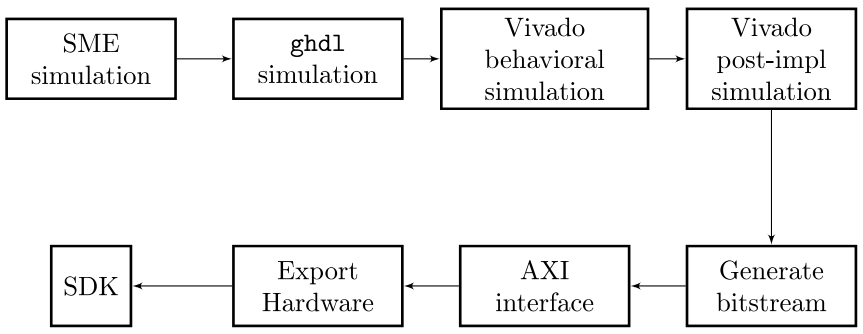

Figure 22: The general workflow for construction actual hardware using SME.

7 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 had time to verify or develop an approach, which will run on additional hardware.

7.1 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. We also need to specify which busses are top-level input and output busses. 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()`

```
public static void Main(string[] args)
{
    new Simulation()
        .BuildCSVFile()
        .BuildVHDL()
        .Run(typeof(MainClass).Assembly);
}
```

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` [8]. `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 [9]. Vivado is a development environment created by Xilinx, an FPGA vendor, for designing and implementing hardware on an FPGA. When creating the project, we must specify the target platform. In this case, we are 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. We will be using the AXI interconnect, when communicating with the FPGA. AXI (Advanced eXtensible Interface) is an interconnect specification. We are not going to focus on how the protocol works. Additionally, the ZedBoard comes with a few buttons, switches and LEDs, which are connected to the FPGA part of the Zynq chip.

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

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 if the design interpreted by Vivado matches the SME network.

TODO få forklaret top-level input output, clock og reset

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. However, we just choose the default: LVCMOS18. This is also the part where we connect the clock signal and the reset signal. All of this is required in by Vivado.

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 we use a clock signal in our design, we should start by synthesizing. Because once the design have been synthesized, we can introduce clocking constraints. Vivado uses these constraints as a target, when it is placing and routing the synthesized design. The only clocking constraint we are going to need 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 in nanoseconds of how long it takes the signal for traversing the 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 we have a positive slack, we can increase the clockrate. If we have a negative slack, the clocrate should be reduced.

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 and all the wires. It also simulates timing on the wires, which is why it also takes some time to run.

If the post-implementation simulation passes, it should also work as expected on actual hardware. 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, we must construct our own 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. The IP which we create 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 store and send information trough the interface. So where we before connected our top-level busses to wires on the FPGA, they must now be connected to these registers. This is done in VHDL, not SME.

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 our IP with the AXI interface. Additionally, if timing constraints on the clock was needed during the implementation of the pure VHDL, i.e. pre IP generation, then the clocking wizard should be added to the block design in order to either multiplying or dividing the clock.

Once the block design have been verified, it should be made into a synthesizable component, by creating a HDL wrapper. By doing this, we can synthesize, place, route and generate the bitstream.

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 should have received a memory address, and the SDK should contain a library with functions for accessing the registers through the AXI interface. I.e. 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.

7.2 Logic gates

We start by implementing the first SME network, which we 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 an `ClockedProcess`, but rather an `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.

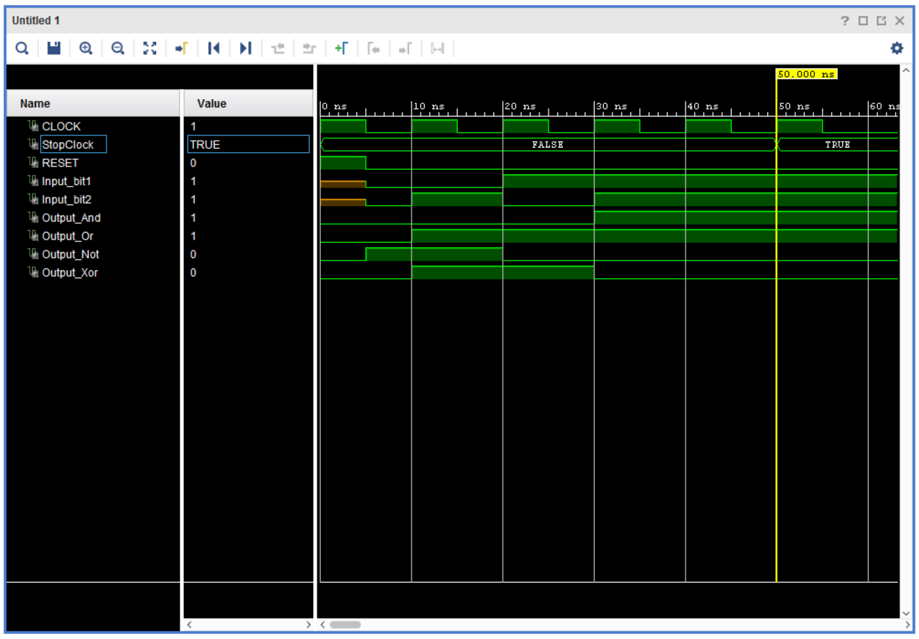

Figure 23: The waveform of the logic gates simulation.

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

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

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

Then we should create the Vivado project. As mentioned in the general workflow, the first thing to do is to run the behavorial simultion, and verify that it runs as expected. Running the logic gates should produce the waveform in figure 23.

Then we should verify that the RTL schematic matches the SME network. The RTL network of the logic gates project can be seen in 24.

Then we should choose where the top-level input and output wires should be connected. On the ZedBoard, the outputs should be connected to the LEDs and the inputs to the switch buttons. The reset signal should be wired to a button and since the logic gates project does not use the clock, anything can be connected.

Then we can synthesize, place and route the project. Once it is done, we can run the post-implementation simulation. It should produce the same waveform as the previous waveform seen in figure 23.

Then the bitstream should be generated. Once it is done, it should be written to the ZedBoard. Flipping the switches should make the LEDs light as shown in figure 25.

Then we need to set up the AXI interface. We start by creating a new RTL project in Vivado. Then we create a new IP by clicking "Tools > Create and

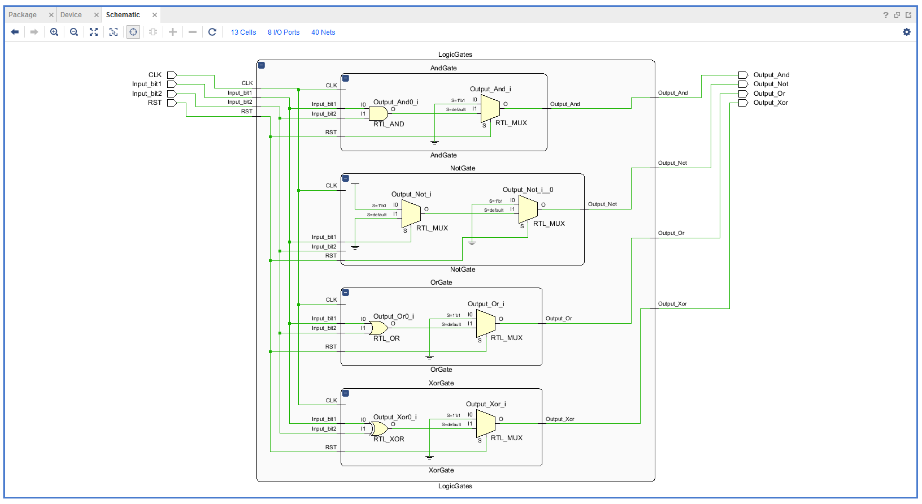

Figure 24: The RTL schematic of the logic gates project.

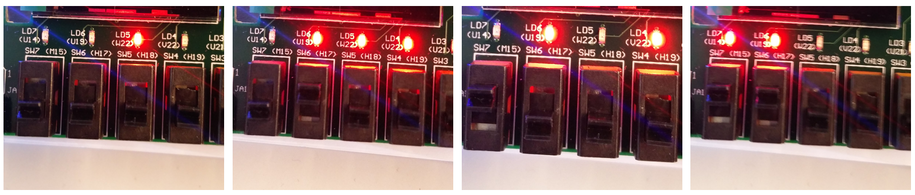

Figure 25: The LEDs with the different configurations of flip switches on the ZedBoard. From left to right the inputs on each of the images are: false-false, false-true, true-false, true-true. The four LEDs from left to right on each image are: `AND`, `OR`, `NOT` and `XOR`.

package new IP". In the new dialog, we select "Create new AXI4 peripheral". Then we go through the wizard, selecting six 32 bit registers and selecting "Edit new IP" in the very end. This opens a new Vivado window, which has the base AXI VHDL process. Then we need to add the files from the logic gates project and edit the AXI process.

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 we needed a clock signal or reset signal, which should be seperate from the AXI clock and reset, we would need to add these ports here. The second file is the registers component of the AXI interface, which handles reading and writing to and from the slave registers. We are going to extend the registers to have signals from our logic gates and then extend the wrapper component, such that the logic gates are connected to the registers component.

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

```
-- Users to add ports here
bit1 : out std_logic;
bit2 : out std_logic;

andgate : in std_logic;
orgate  : in std_logic;
notgate : in std_logic;
xorgate : in std_logic;
-- User ports ends
```

Then we need to ensure that the slave registers that the logic gate needs to write to are exclusivele to the logic gates. We are going to use slave registers 0 and 1 as the input bits, and slave registers 2, 3, 4 and 5 as the output. As such we should find all the occurrences where the registers component writes to slave registers 2, 3, 4 and 5, and comment them out.

```
Before:
-- slave registor 2
slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto
      byte_index*8);

After:
-- slave registor 2
-- slv_reg2(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7
     downto byte_index*8);
```

Finally, for the registers component, we need to add the new logic. It should output 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:

```
-- Add user logic here
bit1 <= slv_reg0(0);
bit2 <= slv_reg1(0);
slv_reg2(0) <= andgate;
slv_reg3(0) <= orgate;
```

```
slv_reg4(0) <= notgate;
slv_reg5(0) <= xorgate;
-- User logic ends
```

Then we need to extend the wrapper component. We start by adding the types generated by SME

```
-- library SYSTEM_TYPES;
use work.SYSTEM_TYPES.ALL;

-- library CUSTOM_TYPES;
use work.CUSTOM_TYPES.ALL;
```

Then, we extend the component definition of the registers component, to have the added ports.

```
port (
    bit1 : out std_logic;
    bit2 : out std_logic;
    andgate : in std_logic;
    orgate  : in std_logic;
    notgate : in std_logic;
    xorgate : in std_logic;
    S_AXI_ACLK  : in std_logic;
```

Then, we add the component definition of our logic gates project, along with the signals going from the logic gates to the registers component. For the ports in the component definition, we just 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, we need to 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, we need to instantiate the logic gates component. The clock and reset signals should just use 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`.

```
-- Add user logic here
LogicGates_inst : LogicGates_export
port map (
    Input_bit1 => bit1,
    Input_bit2 => bit2,
    Output_And => gates_and,
    Output_Or  => gates_or,
    Output_Not => gates_not,
    Output_Xor => gates_xor,
    RST => s00_axi_aresetn,
    CLK => s00_axi_aclk
);
-- User logic ends
```

Now, we have wrapped our Logic Gates project in an AXI interface. The RTL should look as figure 26. Then we just need to package the IP. It should now be available in the IP catalog. In the original project, we need to create a block design. We need to add two components: the processing system and the logic gates with AXI. Upon adding the processing system, Vivado gives us the option to run block and connection automation. When both have run, we should have a block design as in figure 27. The next step is to create a HDL wrapper from the block design, and then generate the bitstream. Remember to add the Types files to the project, otherwise the IP cannot be synthesized. When the bitstream has been generated, we need to export our hardware to the Xilinx SDK. Note: the bitstream must be included.

In the SDK, we start by creating a new Application project. We choose the "Hello World" example project, and call it "Tester". We need to write our new code in `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 we are most interested in is `Tester_bsp/ps7_cortex9_0/libsrc/LogicGates_AXI_v1_0/src/LogicGates_AXI.h`.

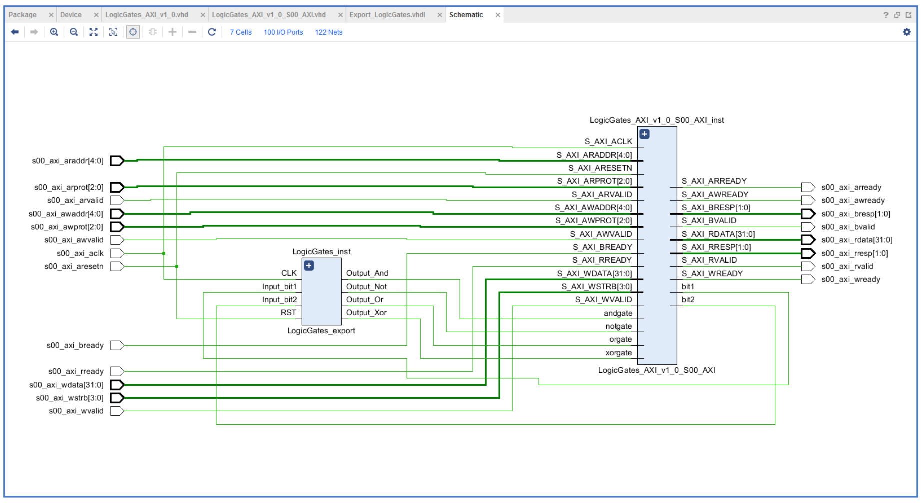

Figure 26: The RTL schematic of the Logic Gates with AXI interface.

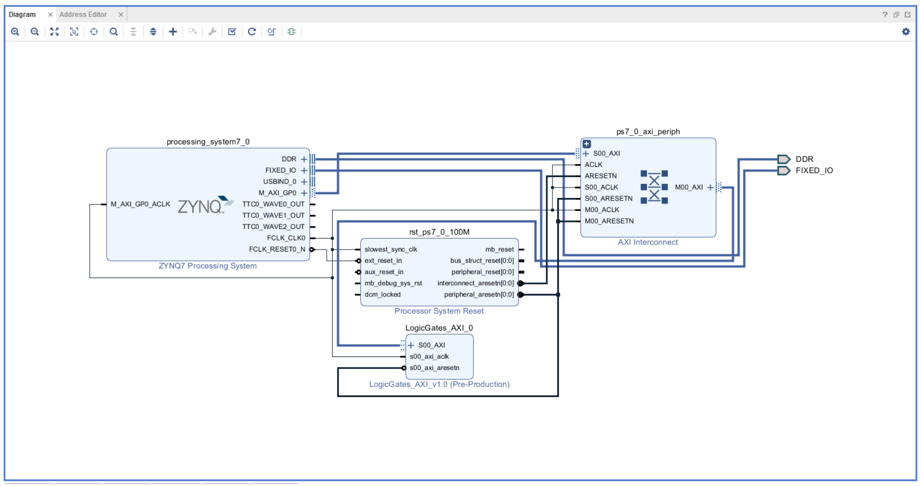

Figure 27: The finished block design contaning the processing system and the Logic Gates AXI IPs.

This file contains all the functions for reading and writing to the AXI registers. We also need `xparameters.h`, as it contains the base address for the AXI interface. We also need `xil_io.h` as it contains the types returned by the read and write functions. All of the header files should be included:

```
#include "xparameters.h"
#include "LogicGates_AXI.h"
#include "xil_io.h"
```

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

```
int base = XPAR_LOGICGATES_AXI_0_S00_AXI_BASEADDR;
int bit1 = LOGICGATES_AXI_S00_AXI_SLV_REG0_OFFSET;
int bit2 = LOGICGATES_AXI_S00_AXI_SLV_REG1_OFFSET;
int and = LOGICGATES_AXI_S00_AXI_SLV_REG2_OFFSET;
int or  = LOGICGATES_AXI_S00_AXI_SLV_REG3_OFFSET;
int not = LOGICGATES_AXI_S00_AXI_SLV_REG4_OFFSET;
int xor = LOGICGATES_AXI_S00_AXI_SLV_REG5_OFFSET;
```

We want to try and write all combinations to the registers, and verify that it outputs the same as the truth table specified in table 1. We start by writing 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;
}
```

Then we program the FPGA, program the ARM and verify the 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
```

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

7.3 Single cycle MIPS processor

Processor is clocked at 5mhz!

References

[1] Brian Vinter & Kenneth Skovhede. *Synchronous Message Exchange for Hardware Designs.* © The authors and Open Channel Publishing Ltd. 2014.

[2] C.A.R. Hoare. *Communicating Sequential Processes.* © ACM 1978.

[3] David A. Patterson & John L. Hennessy. *Computer Organization and Design - The Hardware/Software Interface (Revised 4th edition).* © Elsevier 2012.

[4] *Maskinarkitektur (ARK) 2013/2014* `http://kurser.ku.dk/course/ndaa04009u/2013-2014`

[5] Introduction to Combinational Logic Functions `https://www.allaboutcircuits.com/textbook/digital/chpt-9/combinational-logic-functions/`

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. *Introduction to Algorithms 3rd edition.* ©MIT 2009.

[7] Writing a Towers of Hanoi program `https://www.cs.cmu.edu/~cburch/survey/recurse/hanoiimpl.html`

[8] GHDL `https://github.com/tgingold/ghdl`

[9] Vivado Design Suite `https://www.xilinx.com/products/design-tools/vivado.html`