Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

13th March 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 Introduction

Background and terminology for:

- CSP
- SME
- Machine architecture
- DIKU course
- VHDL
- C#

2 Getting started with SME

2.1 Installing required software for SME

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.

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

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`.

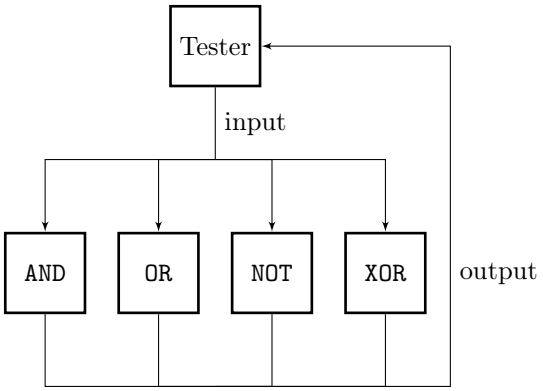

Figure 1: The structure of the test of the logic gates

Testing

To test the four processes, we construct a 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. How the processes are connected can be seen in Figure 1.

3.2 Decoder

A decoder 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`.

A decoder can be made from a set of `NOT` and `AND` gates. 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

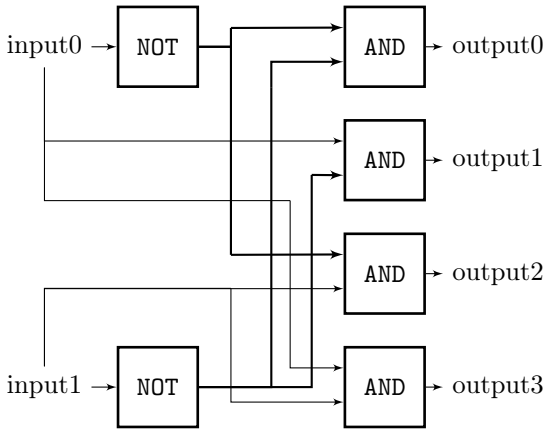

Figure 2: An 2-bit decoder made by `AND` and `NOT` gates.

2.

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 before, with a tester process, which sends input to the logic gates, and verifies the output from the gates. This is also what we need to do with the n-bit decoder. However as with the actual n-bit decoder, we are going to need C# templates. Each of the two tester processes should send all possible inputs as input.

3.3 Adder

As with the decoder, an adder can be constructed by a combination of `AND`, `OR` and `XOR` gates. 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.

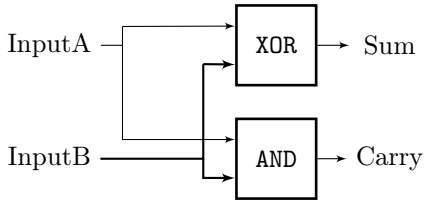

Figure 3: An half adder composed of `XOR` and `AND` gates

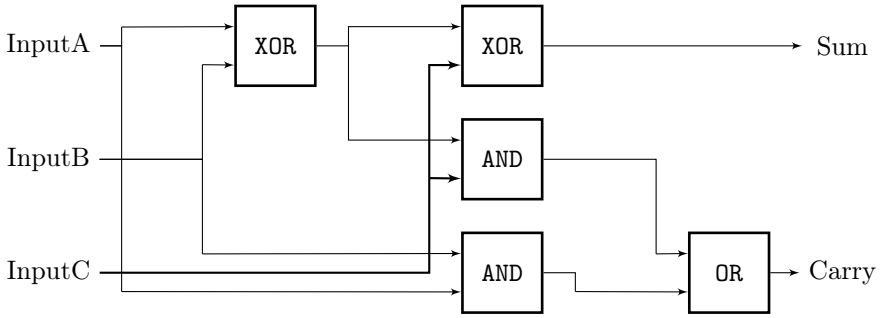

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates

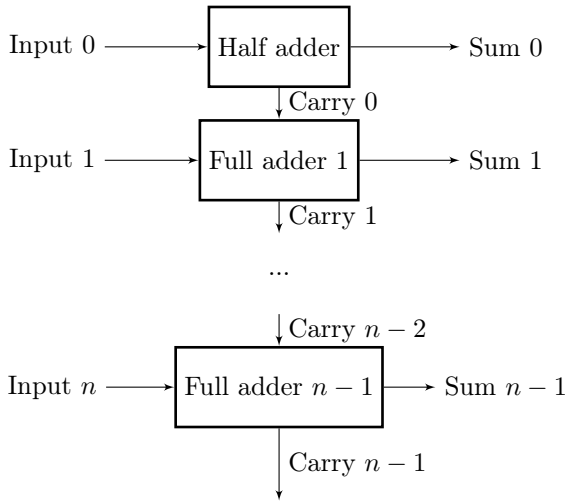

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

Implementation

The half adder and the full adder is made as 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 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 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 also 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. There are two tests: the simplest addition (2+2), overflow, and finally a series of random numbers.

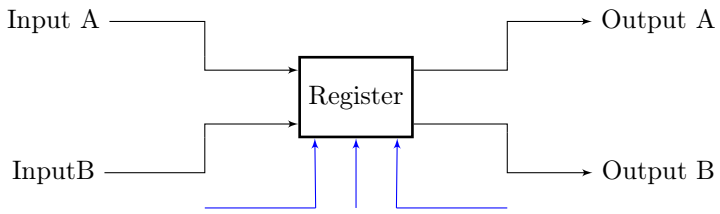

Figure 6: The register file

4 Core components

In this section, we will be looking at the major components of the MIPS processor. For each component, we will go through the theory of the component, then we will look at translating theory into an SME implementation, and the verification of the implementations. The order in which the components are mentioned, is 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 Register file

The Register File is the component that holds values for the processor. It is the first step in a memory hierarchy, and is thus the fastest memory available. There are 32 registers in a 32-bit MIPS processor. The registers are divided into groups based on their usage. This does not matter from a hardware perspective, except for register 0, which is immutable and always 0.

The Register File has 5 inputs: Read Address A, Read Address B, Write Enabled, Write Address and Write Data. It also has two outputs: Output A and Output B. It has two stages: 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 B, 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, and stores it in the register with the address in Write Address.

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. The Register File and its inputs and outputs can be seen in Figure 6.

Implementation

The implementation of the register file in SME is done by the use of an `int` array. We construct a process, that when all of its inputs are ready, it takes the write input, and stores it in the array with the given index, if it is greater than

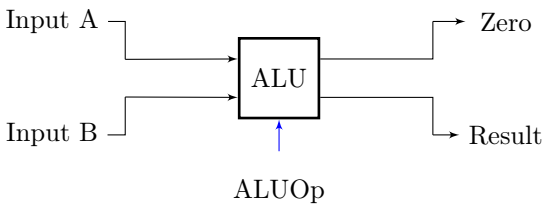

Figure 7: The ALU

0, and if the `RegWrite` flag has been set. In case we do get a 0, we just ignore the request, as this is usually the pattern of a `nop` instruction.

After the process have written to the register, it processes the read addresses, and outputs the values stored in the array at the given addresses.

Testing

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.

4.2 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 7.

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

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 start by implementing the same instructions as the book proposes: `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.

Constructing the ALU process in SME is straightforward, it reads 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. Finally, the ALU should

set the flag on the zero bus, whether or not the computation was 0. How the ALU opcode is encoded is described in Section 4.4.

Testing

Testing the ALU is like the Register File, very trivial. Construct a tester process, which sends values on the Input A, Input B and ALU Opcode busses, reads the values from the Result and Zero busses, and verify that the values are as expected.

4.3 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. 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.

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 can be seen in Figure 8

Implementation & Testing

The book describes the logic needed to implement this unit. However, it only describes the logic for the before mentioned instructions, and is therefore hard to extend. Therefore, we start by making an extendable Control Unit.

As with the ALU, start by having an `enum` for both the opcode and the ALUOp. Then the process should `switch` on the opcode, and set the flags accordingly. This way, adding more instructions is just adding entries in the `enum`s, and adding cases to the `switch`, and then it is up to the VHDL generator to construct the logic.

As with the Register File and the ALU, we test the Control Unit, by having a tester process, which sends input, and verifies the output values.

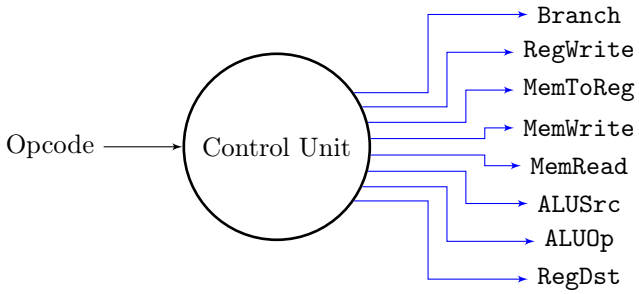

Figure 8: The Control Unit

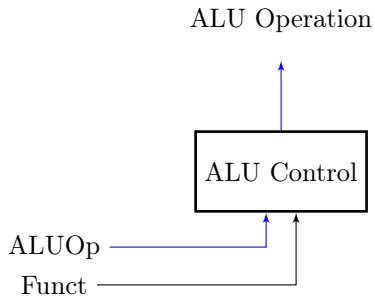

Figure 9: The ALU Control

4.4 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, and it computes its output based on these. The ALU Control can be seen in Figure 9

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

Implementation & Testing

As with the main Control Unit, the book describes usable logic, but it does not correspond to our own ALUOp from the Control Unit, or the needed ALU Operation codes for the ALU. As such, we should do the same as in the Control Unit. We start by checking whether or not the opcode from the instruction indicates that the instruction is an R-format. If this is the case, we need to `switch` on the `funct` field of the instuction, otherwise `switch` on the opcode from the ALUOp.

It is tested in the same manner as the Control Unit.

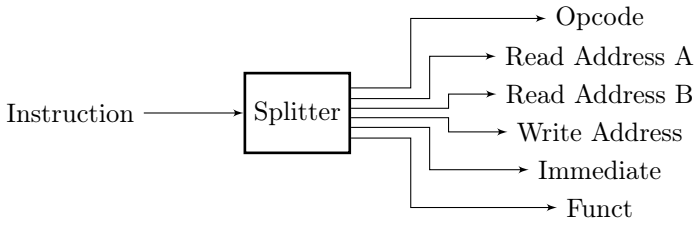

Figure 10: The Splitter

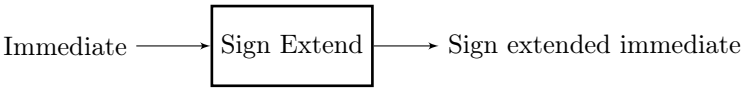

Figure 11: The Sign Extend

4.5 Splitter

This is a very simple component, but it is still used in the decoding step of the processor. 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 can be seen in Figure 10

Implementation & Testing

The implementation is straightforward: We take the instruction comming from the instruction fetch part of the processor, and extract the bits at the indices, by using C# bit hacking. Finally, output the extracted values on the respective output busses.

As before, testing is performed by constructing a tester process, which sends input, and verifies the output.

4.6 Sign Extend

The Sign Extend is used for extracting values from the instruction. 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 11

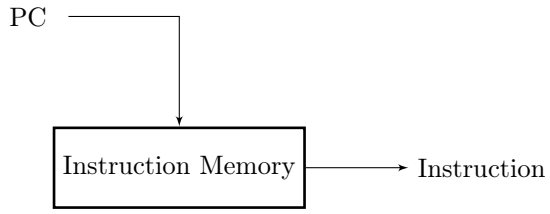

Figure 12: The Instruction Memory

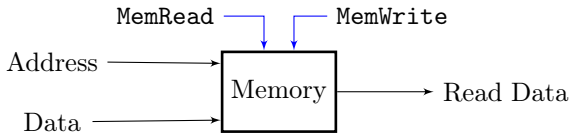

Figure 13: The Memory Unit

Implementation & Testing

The SME proces takes the 16-bit immediate, and outputs it on its 32-bit output bus. C# handles the sign extension for us.

It is tested in the same manner as the previous components.

4.7 Instruction Memory

The Instruction Memory is the part of the processor, which holds the program. It has a chunk of memory, and for each clock, it outputs the value at the given address. It has one input, the program counter (PC), and one output, the read instruction. The Instruction Memory can be seen in Figure 12

Implementation & Testing

Upon receiving all of its inputs, the SME process should read the address from the PC bus, and output the value stored in the memory at the read address.

Usually the memory should be a `byte` array. However, since we are working with C#, we can just use a `int` array for simplicity, as we do not have to worry about word alignment.

The Instruction Memory is tested in the same manner as the previous components.

4.8 Memory unit

The memory unit is the main memory. The CPU can either read or write to the memory unit. The addresses for the memory are word sized, i.e. in the 32-bit processor, the word size is 32 bit. The Memory Unit has four inputs: Address, Data, `MemRead` and `MemWrite`. It has a single output: Read Data. In one clock, the Memory Unit either reads or writes. The Memory Unit can be seen in Figure 13

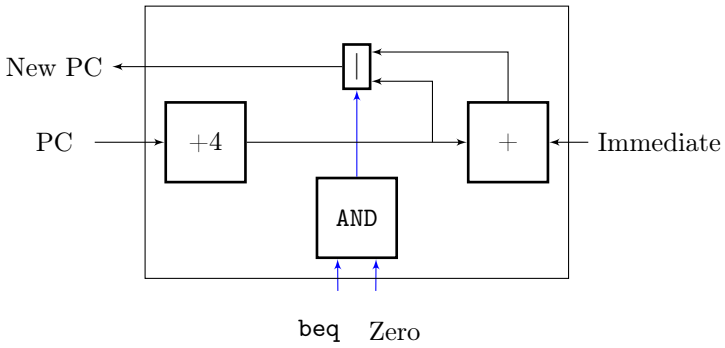

Figure 14: The Jump Unit

Implementation & Testing

As with the Instruction Memory, we are going to need a chunk of memory. However, this time we do want to construct it using a `byte` array, as later programs expect this. As such, the SME process needs to pack four bytes into an `int` value.

In each clock, 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. Then 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.

The Memory Unit is tested in the same manner as the Register File.

4.9 Jump Unit

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. The Jump Unit and its connections can be seen in Figure 14

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.

Implementation & Testing

The book describes the logic for both of the initial requirements. To add the normal program traversel, the Jump Unit should take the input PC, and increment it by 4, such that it points to the next instruction. To add support for branching, it should take the incremented PC, and add it with the sign extended immediate. The `beq` signal and the Zero signal should go to an `AND` gate, which should be a control signal for a multiplexor. The incremented PC, and the PC with the added immediate should go into inputs for the mux. If the control signal into the multiplexor is `0`, then the incremented PC should be the output, otherwise the PC with the added immediate. The output from the multiplexor should also be the output for the Jump Unit.

The testing is performed in the same manner as with the other components.

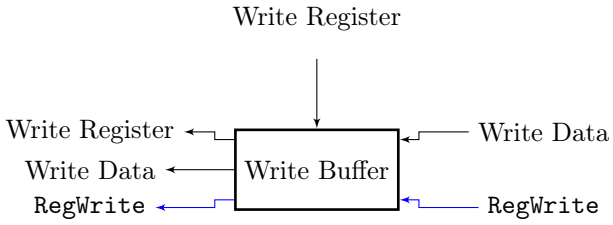

Figure 15: The Write Buffer

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.

Implementation & Testing

Usually in the single cycle MIPS processor, there is nothing special in the Write Back. 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 takes the Write Data, Write Register and `WriteEnabled` as input, and produces the same output. On each clock, it should output its stored values, and store its input values. The Write Buffer can be seen in Figure 15

Testing is performed in the same manner as the previous components.

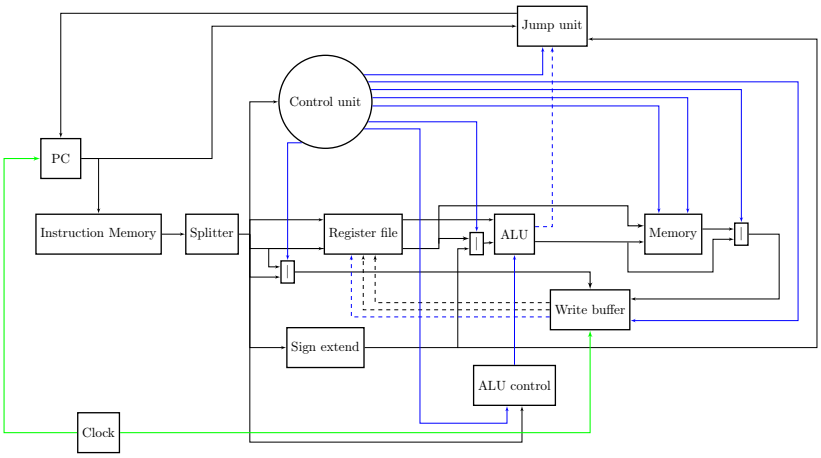

Figure 16: Simple single cycle MIPS processor. The units with | indicate multiplexors. The PC Unit is just a simple register.

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 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 the Write Buffer and the PC register should be clocked processes. The wiring og 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 be testing these. The program and the different parts of the instructions can be seen in Table 2.

Inserting this program into the Instruction Memory is straightforward: just create an `uint` array, which is initialized to the hex values from 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.

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.

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.

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 in the ALU Control and the ALU. For the unsigned instructions, it is important to remember to cast before making the computation.

Testing these new instructions is straightforward: just append them to the simple program.

Immediate instructions

The next instruction we want to add is the `addi` instruction, as this would allow us to feed values into the processor without hardcoding it. While we are doing this, we should also add `addiu`, `slti`, `sltiu`, `andi`, `ori` 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, all we need to do, is extend the `enum`s and the `switch`es. The affected units are: The Control Unit, the ALU Control and the ALU.

Once the processor have been extended, we can prepend instructions at the beginning of the program, in order to insert initial values into the Register File, and then it is just adding each instruction, and verifying the output 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 first 26 bits to the Jump Unit. Then the Control Unit should be extended, both in its `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.

Branching instructions

Then we are going to add the `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 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 `jr`, `jal` and `jalr` instructions, 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 then compute the new address in the same manner as usual, and 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 `sll`, `slr`, `sra`, `sllv`, `srlv` and `srav` instructions, 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 `mult`, `multu`, `div` and `divu` instructions. 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.

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 are going to construct some low level `C` code, which should be easy translatable into assembly. 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.

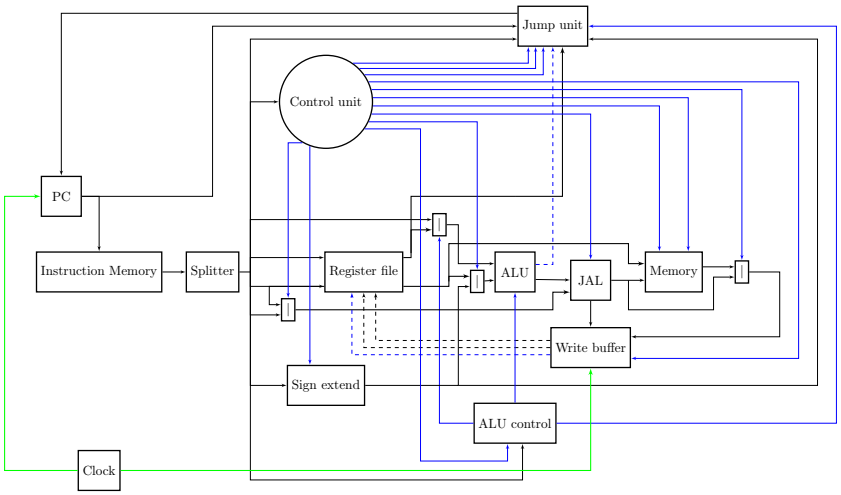

Figure 17: Full single cycle MIPS processor. The units with | indicate multiplexors. Black wires indicate data wires. Blue wires indicate control wires. Green wires indicate clock

Quicksort

There are three parts of the Quicksort program: Loading data into memory, the partition function and the quicksort function. We are going to use Hoare partition as our partition function, and the quicksort function as described in the algorithm book.

```
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  }
9
10 int partition(int *a, int p, int r) {
11     int x, i, j, *addr1, *addr2, val1, val2;
12     x = *(a + p);
13     i = p - 1;
14     j = r + 1;
15     while (true) {
16         do {
17             j--;
18             addr1 = a + j;
19             val1 = *(addr1);
20         } while (val1 > x);
21         do {
22             i++;
23             addr2 = a + i;
24             val2 = *(addr2);
25         } while (val2 < x);
26         if (i < j) {
27             *(addr1) = val2;
```

```
28              *(addr2) = val1;
29          } else {
30              return j;
31          }
32      }
33  }
34
35  void quicksort(int *a, int p, int r) {
36      if (p < r) {
37          int q = partition(a, p, r);
38          quicksort(a, p, q);
39          quicksort(a, q+1, r);
40      }
41  }
42
43  int main() {
44      int arr[6];
45      load(arr);
46      quicksort(arr, 0, 5);
47  }
```

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 disk at the time. We are going to represent the three pegs as an array, which is three times the size of the tower.

```
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   }
9
10  void tower(int num, int **from, int **to, int **aux) {
11      int *t, *f;
12      if (num == 1) {
13          t = *to;
14          f = *from;
15          f--;
16          *t = *f;
17          t++;
18          *f = 0;
19          *to = t;
20          *from = f;
21      } else {
22          tower(num-1, from, aux, to);
23          t = *to;
24          f = *from;
25          f--;
26          *t = *f;
27          t++;
28          *f = 0;
29          *to = t;
30          *from = f;
31          tower(num-1, aux, to, from);
```

```
32      }
33  }
34
35  int main() {
36      int num = 5;
37      int *arr = int[num*3];
38      init(num, arr, arr+num, arr+(2*num));
39  }
```

6 Pipelining

aoeu

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`