Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

27th February 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 Introduction

Background and terminology for:
- SME
- Machine architecture
- VHDL
- C#

2 Getting started with SME

2.1 Installing SME

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

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`.

3 Basic logic circuits

In this section, we will be looking at some basic combinatorial circuits. 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. Finally, an n-bit adder, by combining a chain of a half adder followed by $n-1$ full adders.

For each of these combinatorial circuits, I state 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

We start by defining some of the common 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.

`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` 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. Each process takes the two bits from the input bus, computes their respective logical function and sends the result out on the output bus.

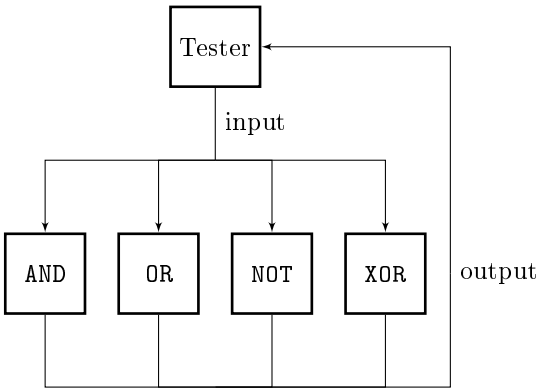

Figure 1: The structure of the test of the logic gates

Testing

To test the four processes, I have made 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 on their output bus. How the processes are connected can be seen in Figure 1.

3.2 Decoder

The first combinatorial circuit we make is the decoder. An decoder takes an n-bit input, and produces an 2^n-bit output, where the bit corresponding to the input numbers value 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

We have previosly made the logic gates that we need, and as such we only need to wire up these logic gate processes as specified by Figure 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

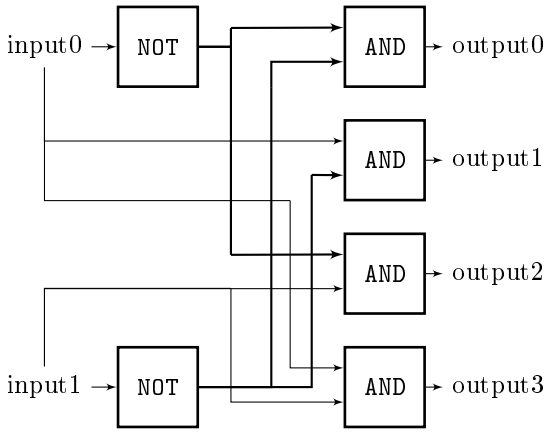

Figure 2: An 2-bit decoder made by `AND` and `NOT` gates.

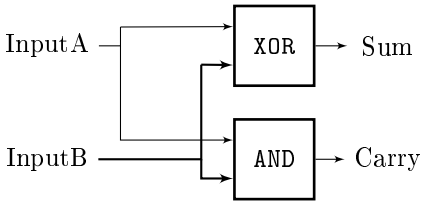

Figure 3: An half adder composed of `XOR` and `AND` gates

`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

I have written both a 2-bit decoder, and an n-bit decoder. They are tested in the same fashion as the logic gates, i.e. a tester process is connected to the input and output busses. The test process inputs every combination of binary values, i.e. 2^n, and checks after each input, that the correct output is set to `1` and `0` respectively.

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.

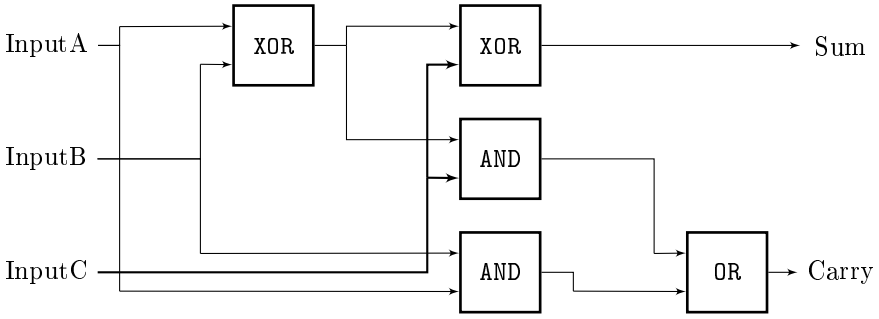

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates

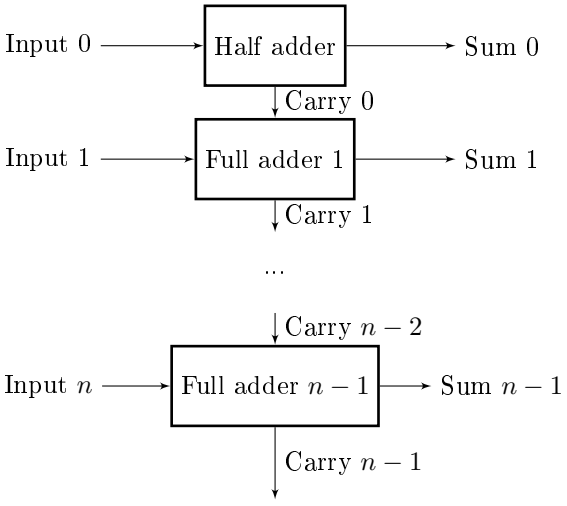

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

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

I have tested both the half adder and the full adder, by giving each every possible input, and for each, compared them to their expected output. For the n-bit adder, I have made a function, that takes an integer as input, sends it along the corresponding input wires. I have also made a similar function, that takes the values from the output wires and packs it into an integer. I start by testing if it can make one of the simplest additions: 2+2. Then I check if it can overflow, i.e. set the Carry $n-1$ to 1. Finally, I run a series of random numbers through the adder, and check if the output is the expected sum of the two numbers.

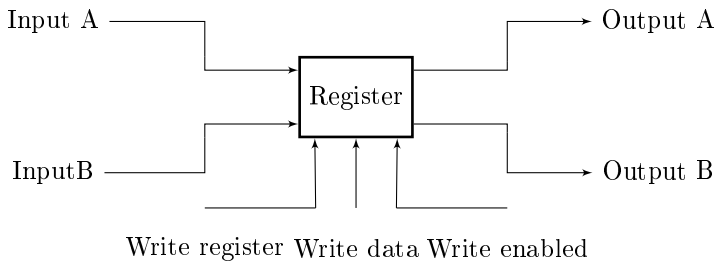

Figure 6: The register file

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 devided into groups based on their usage. This does not matter from a hardware perspective, except for register 0, which is immutable and always 0.

A 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. 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, which is greater than 0, if the `RegWrite` flag has been set. We will explain the flag in more detail in Section 4.3. 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 then processes the read addresses, and outputs the values stored in the array at the given indices.

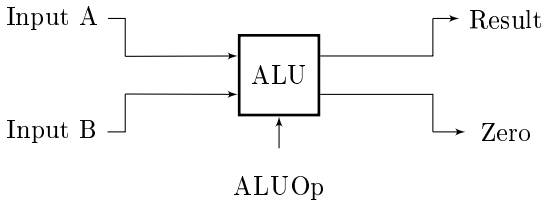

Figure 7: The ALU

Testing

I start by testing if some of the initial values of the register file is correctly set to 0. Then I test if I can write to a register, and whether or not I can read the same value from the same address in the register. Then, I try to write to all of the registers, except for 0, and check whether or not the output that I get, corresponds with the output that I tried to write.

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.

Implementation

To implement this, we start by making an `enum`, so that the code becomes more 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`. The ALU takes the ALU opcode, and based on that it performs its computation. For the needed instructions, the ALU should be able to perform addition, subtraction, logical `AND`, logical `OR` and the comparison less than. How the ALU opcode is encoded is described in section 4.4. Doing this in SME is straightforward, we read from the ALUOp bus, and then we have a `switch`, and for the instructions that we do accept, we take the input from the two input busses, do the computation, and output the result on the resulting bus. Finally, output on the zero bus, whether or not the computation was 0.

Testing

For each of the operations that the ALU can perform, I have tested if the output from the ALU matches the expected value.

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 Two bits 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.

Jump Controls whether or not the instruction is a jump instruction, i.e. if the `PC` register should be changed.

JAL Contrals whether or not the instruction is a `jal` instruction, i.e. that the `PC+4` address should be stored.

LogicalImmediate Controls whether or not the sign extender should be a sign extender, in the case of a numeral immediate, or if it should be a zero extender, in the case of a logical I-format instruction.

Each of the control flags goes to their respective part of the processor.

Implementation

To implement this, I have simply made a `switch` statement on the opcode. Then, for each type of opcode, I set the appropiate flags and the `ALUOpcode`.

Testing

meh - det er gjort dog...

4.4 ALU control

As with the normal Control Unit, we need to produce a signal, based on some input. 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.

Implementation

As with the main control unit, I have just implemented the combinatorial logic from the book.

Testing

Pretty much det samme som før

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, e.g. the control unit only need the 6 most significant bits of the instruction, as this makes up the opcode of the instruction.

Implementation

The implementation is straightforward: We take the instruction comming from the instruction fetch part of the processor, and extract the bits at the indices specified in the book (eller lav egen tegning!)

Testing

Meh, der er lavet en smule, da jeg tester både splitter, register og alu samtidig.

4.6 Sign extend

This component is used for converting values within the immediate field of an instruction, which is 16-bit, to a 32-bit number, maintaining the sign value.

Implementation

To implement this, I just take the value from an input bus, which contains a `short` value, and send it along the output bus, which contains an `int` value.

Testing

Jeg har jo ikke testet dem individuelt...

4.7 Instruction memory

The instruction memory consists of three parts: the `PC` register, which is the address of the current program line. The program memory, which is just like the memory unit, a chunk of memory. And an `PC` incrementer, which adds word length to the `PC` after every instruction.

Implementation

The PC is just a clocked process, that forwards its input. The program memory, is like the register file, but with less connections. It only has a single input and a single output bus. It takes the address given on the input bus, extracts the instruction from the instruction memory, and sends the instruction on its output bus.

The incrementer just adds 1 to its input. In the book, it states that it should be 4 added to the program counter. However, since we are working in C#, we can just use an `int` array, and instead of adding 4, we just add 1, as this will increment where in memory we are pointing.

Testing

It has been tested! IF has a program that it pushes through the processor.

4.8 Jump control

The jump control is the part of the processor, that based on the instruction changes the value of the PC register. There are two ways to jump: by branching and by jumping.

Implementation

We start by implementing the control for the branching logic. We need to support 2 branch instructions in our basic processor: branch equal and branch not equal. Therefore we will need two signals: `Branch` and `BranchNot`. The `Branch` is used to indicate that the instruction should change the PC register. The `BranchNot` is used to choose whether or not the `Zero` flag from the ALU should be `1` or `0` for the instruction to branch. In both instructions, we compute the subtraction of the two input registers, and if the result is `1` they are equal, otherwise they are not equal. The `Zero` signal is then send to an `AND` gate along with the `Branch` signal. bla bla decoder.....

Then we need to implement the jump unit. This is a little bit easier, as we do not depend on anything from the ALU. The problem is that we need to change the decoder to accept a new format: the J-format. The J-format consists of an opcode, and a 26-bit address. The address is left shifted by 2, to ensure word alignment, and the upper 4 bits of the PC is added to the final address. We also need another control signal: `Jump`. We use this to multiplex between the newly computed address and the output from the previous mux (branch or pc+4).

Testing

`nop`

4.9 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 our case 32 bit.

Implementation

We are going to need two signals: `MemRead` and `MemWrite`. Furthermore, we need an address input, an data input and an data output. On each clock, we first check if we should read, in which case we output the word at the given adress, if we should write, in which case we store the word from the data bus at the given address.

Testing

aoeuaoeuaoeuaoeuoaeuaoeue

4.10 Write back

The final stage is the writeback, where we take the computed result, and store it in the appropriate registers.

Implementation

(Måske som førnævnt??) Since there is a cycle from the register file, through the alu and memory, back to the register file, we need to have a clocked buffer. As such, after each instruction, the result is stored in this buffer, and on a clock tick, the value in hold is put on its output bus to the registers.

Testing

meh

4.11 Adding additional instructions

To add additional instructions, we need to add additional circuitry and entries in the `enum`s. We will go through implementing the MIPS core instruction set.

We start with the remaining R format instructions (Except for shifting, multiplication and division). This is straightforward, as we only need to add additional entries to the `enum`s, add the additional cases in the `switch` in the ALU control and in the `switch` in the ALU.

For shifting, we need to extract extra information from the instruction in the Splitter: the `shamt` field. Following this, we add an additional control signal, and multiplex on whether it should be the value read from the registers or the `shamt`, which should be the B input for the ALU.

Then we have the remaining arithmetic I format instructions. This is also straightforward, as we only need to extend the `enum`s, and add additional entries to the `switch` in the ALU control.

Next we need the `jal` instruction. Here we need circuitry to store the contents of the `PC` register. To do this, we send the `PC` to the `EX` stage, and multiplex on the output from the ALU and the `PC`, with the `JAL` control signal.

Next we need the `jr` instruction. We add an additional signal, and multiplex on the value read from register, and the normal `PC+4`.

Then we have the `mult`, `multu`, `div` and `divu` instructions. These are all R format, so we follow the same procedure. However, instead of sending the result on the ALU result bus, we store it in two additional registers, which we keep in

the ALU: `HI` and `LO`. Following this, we can also easily add the instructions `mfhi`, `mflo`, `mthi` and `mtlo`, which moves data to and from the two new registers.

Finally, we have the `bne` instruction. We add an additional signal, and multiplex the `NOT`ed `zero` signal from the ALU as the signal for the branching `AND` gate.

4.12 Test programs

Snak om at bruge MARS!

Now that we have our single cycle MIPS processor, we are ready to throw actual instructions at it. We start by collecting all of the smaller test instructions into a full test program. SE APPENDIX! (og lav det for den sags skyld)

Then we have a Quicksort written in MIPS assembly.