Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

10th February 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 Introduction 3

2 Getting started with SME 4

2.1 Installing SME 4

2.2 Writing first SME program 4

2.3 C# and bit hacking 4

2.4 Translating first SME program into VHDL 4

2.5 Running and verifying VHDL 4

3 Basic logic circuits 5

3.1 Basic logic gates 5

3.2 Decoder 6

3.3 Adder 7

4 Core components 10

4.1 Register file 10

4.2 ALU 10

4.3 Control Unit 11

4.4 ALU control 11

4.5 Instruction memory 12

4.6 Jump control 12

4.7 Memory unit 12

4.8 Write back 12

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.

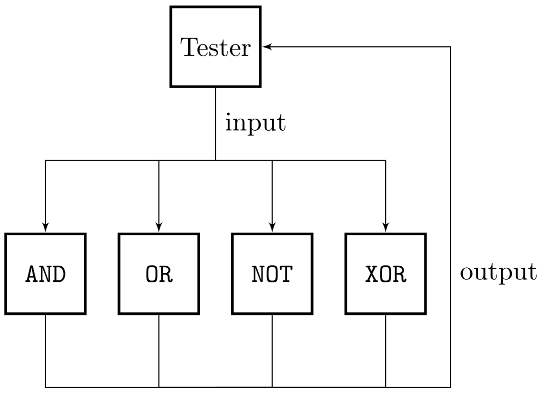

Figure 1: The structure of the test of the logic gates

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

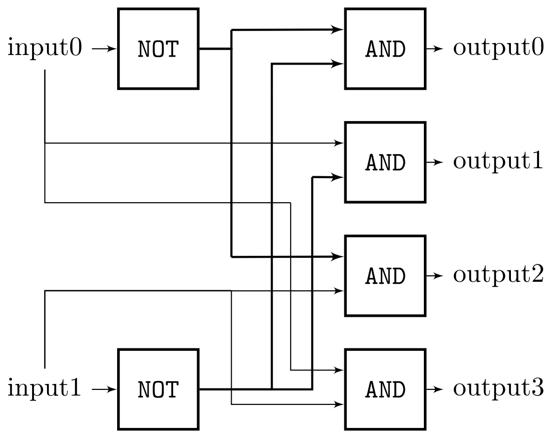

Figure 2: An 2-bit decoder made by `AND` and `NOT` gates.

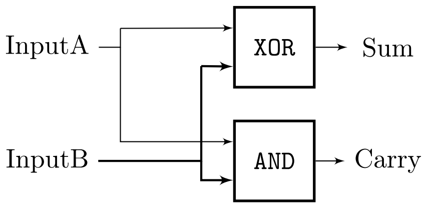

Figure 3: An half adder composed of `XOR` and `AND` gates

`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.

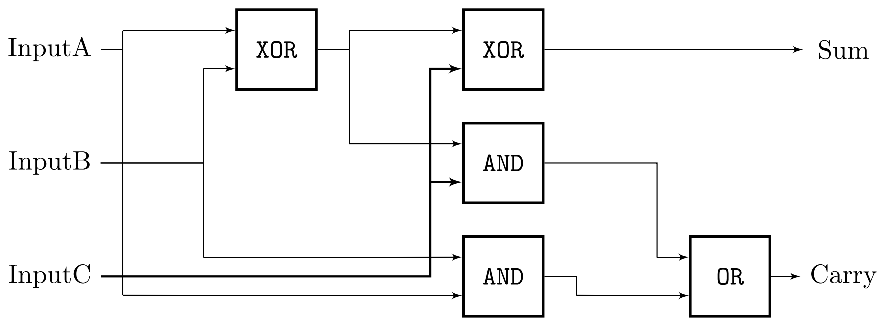

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates

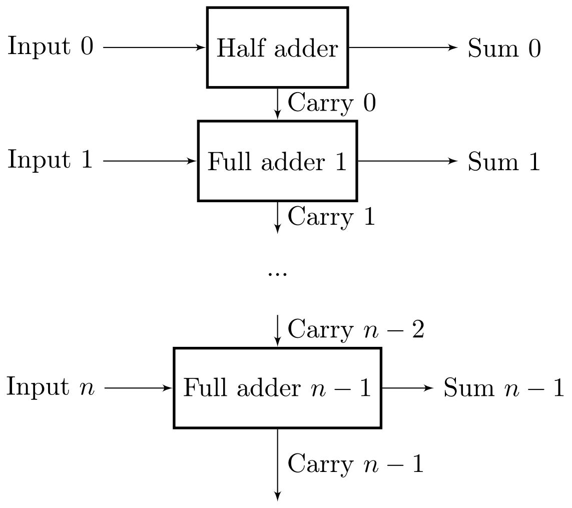

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

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.

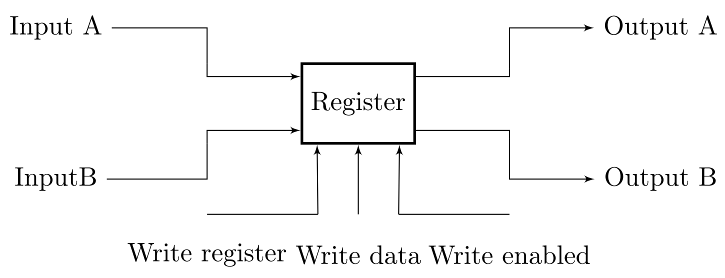

Figure 6: The register file

4 Core components

The components are mentioned in order of implementation.

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 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 written value. This is easy to fix in the single cycle processor, as we just need to write before reading.

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 wrote.

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.

I follow the approach from the book and have implemented the basic processor operations: `add`, `sub`, `and`, `or`, `slt`, `sw`, `lw` and `beq`. The encoding of the ALU opcode is described in Section 4.4

I have tested each of the eight operations, and by looking at the results from the ALU, the results look sane.

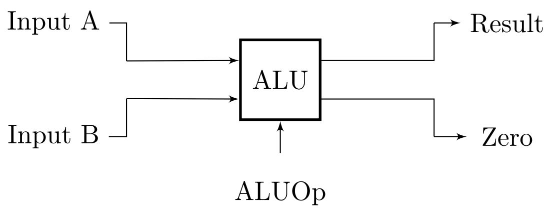

Figure 7: The ALU

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 seven 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.

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

4.4 ALU control

The ALU control is used for generating the 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. 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.

4.5 Instruction memory

4.6 Jump control

4.7 Memory unit

4.8 Write back