Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

7th March 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 Introduction

Background and terminology for:
- SME
- Machine architecture
- 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

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

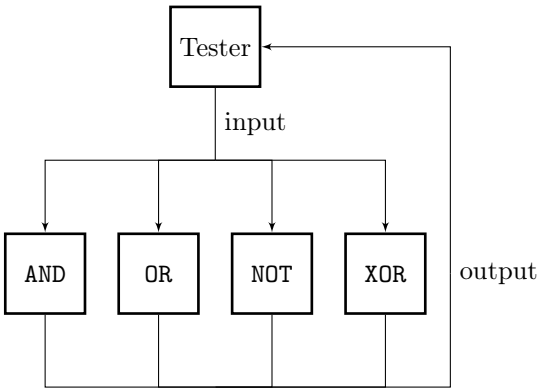

Figure 1: The structure of the test of the logic gates

Testing

To test the four processes, 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 on their 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 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

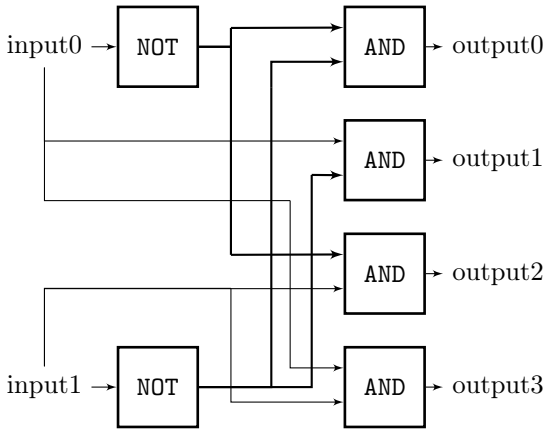

Figure 2: An 2-bit decoder made by `AND` and `NOT` gates.

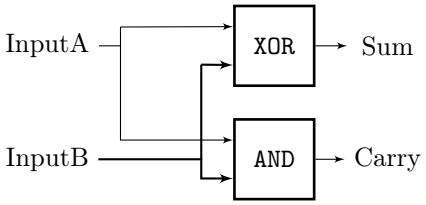

Figure 3: An half adder composed of `XOR` and `AND` gates

`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

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.

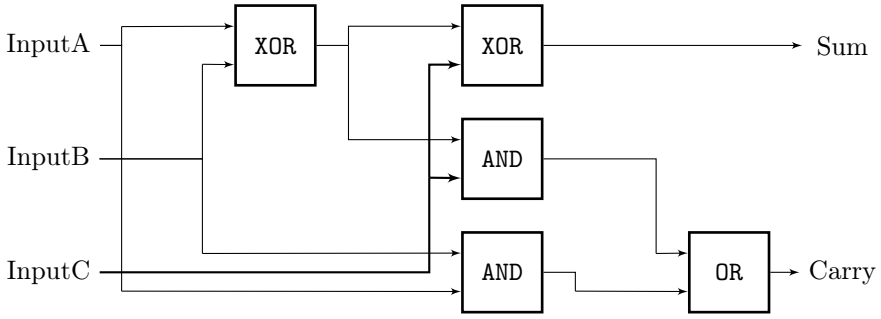

Figure 4: A full adder composed of `AND`, `OR` and `XOR` gates

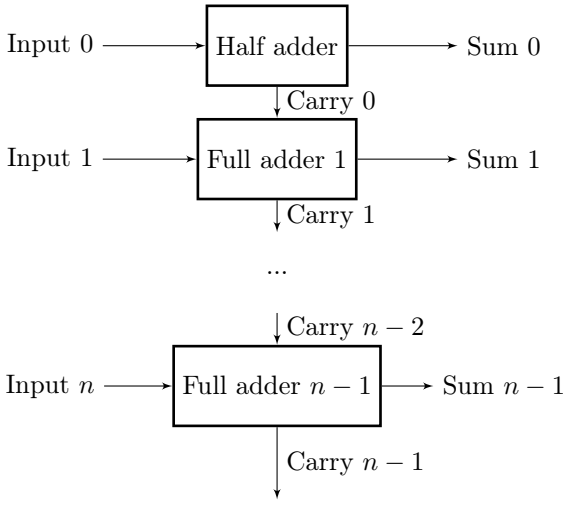

Figure 5: An n-bit adder composed of a half adder, and $n-1$ full adders. Note: Input A and B are both inside the inputs for simplicity.

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.

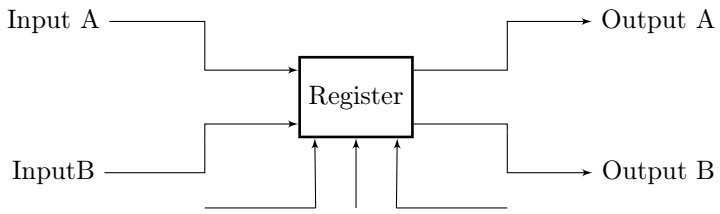

Figure 6: The register file

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

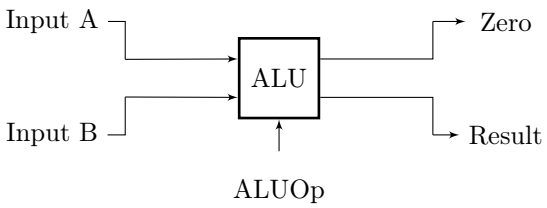

Figure 7: The ALU

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.

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 `enums`, 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.

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.

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.

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

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.

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, and one output, the read instruction.

Implementation & Testing

Upon receiving all of its inputs, the SME process should read the address from the Program Counter 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.

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 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.

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

5 Single cycle MIPS processor

aoeu

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