Implementing a MIPS processor using SME

Carl-Johannes Johnsen (grc421)

3rd February 2017

Abstract

asoenuthnsoaeuhtt

Contents

1 todo introduction?

1.1 Software requirements for using SME

SME is a library for `C#`. Therefor we need to setup a development environment for `C#`. We will be using `mono` URL

Now that we have `mono` installed, we need to download and run `SME`. We start by cloning the project from github: URL!. Then, we add the project in `monodevelop`. Before building the project, we need to generate a few files (`SME.Render.VHDL/Entity.tt` and `SME.Render.VHDL/TopLevel.tt`, which is done by right clicking the file, and choosing 'Tools > Process T4 Template'. Finally, we can open one of the example programs, and press f5 to build and run.

1.2 Logic gates

To start with, I am going to implement the 4 basic logic gates: `AND`, `OR`, `NOT` and `XOR`, as these are the basic building blocks of a processor.

I start by describing the busses. I have one input bus, which has two fields: `Bit1` and `Bit2`, and an output bus, which has four fields, one for each gate. Note that these interfaces has to be public

Then I define the processes for each of the logic gates. They are very simple, as they just take the two inputs, applies their logic operation, and puts the output on the designated lane on the output bus.

Finally, I wrote a test process, which feeds input to the gate processes, stores the output from the gate processes, and finally verifies the collected results, which is:

Bit1	Bit2	AND	OR	NOT	XOR
false	false	false	false	true	false
false	true	false	true	true	true
true	false	false	true	false	true
true	true	true	true	false	false

1.3 Decoder

A decoder takes an n-bit input, and has 2^n outputs. Exactly one of the outputs are `1` on any given input. If the value of the input is `0`, then bit0 of the output is set to `1`, and the rest is set to `0`. If the value of the input is 124, then bit124 is set to `1`, and the rest is set to `0`.

I have implemented a 2-bit encoder, by having 2 `NOT` gates, and 4 `AND` gates:

1.4 Scalable Decoder

The previous decoder was a hardcoded decoder, i.e. that I had manually defined all of the busses and the processes. Since SME requires that everything is known at compile time, we have to define everything statically. This results in an exponential amount of work when making a larger decoder, where the work is primarely defining the buses, each has to have an unique name, and defining the processes, as each has to read from a different bus, or in another case, read from a different wire on the bus.

To solve this, I have used T4 Templates to generate the source code files for me, since the network is just a bunch of busses, `NOT` gates and `AND` gates.

1.5 Half adder

A half adder is a circuit, which takes 2 inputs, each one bit, and adds them together, outputting a sum bit and a carry bit.

1.6 Full adder

A full adder is a circuit, which takes 3 inputs, each one bit, and adds them together, outputting a sum bit, and a carry bit. The difference between a half adder and a full adder is the extra input, which is designated to be the carry from the previous full adder in a chain of adders.

1.7 32 bit adder

Now that I have made scalable `SME` networks, an half adder and a full adder, I can make a 32 bit adder. The 32 bit adder starts with an half adder connected to the first bit of the first input number and to the first bit of the second input number. The output of the half adder is the first bit of the output. The remainder of the adder consists of 31 full adders, where we say that the ith full adder is connected to the ith bit of both of the input numbers, to the carry from the $i-1$th adder, and outputs the ith bit of the result, and the ith carry. The last carry is the flag indicating whether or not the addition produced an overflow.

I have made the implementation using Templates, and as such, we can also use this to produce an n-bit adder, just by changing the `BitWidth` variable of each of the template files.