CPE 233 Hardware Assignment 5

Branch Condition Generator and Branch Address Generator

Report by:
Ethan Vosburg (evosburg@calpoly.edu)

February 14, 2024

Table of Contents

1 Project Description **3**

2 Structural Design **3**

2.1 Branch Address Generator Elaborated Design 3
2.2 Branch Condition Generator Elaborated Design 4
2.3 Otter Memory Module Signal Table 5

3 Synthesis Warnings **5**

3.1 Branch Address Generator Synthesis Warnings 5
3.2 Branch Condition Generator Synthesis Warnings 6

4 Verification **6**

4.1 Verification Methodology and Scope 6
4.2 Bounded Model Checking 7
4.3 Coverage Verification 8

5 Source Code **9**

5.1 Branch Address Generator Source Code 9
5.2 Branch Condition Generator Source Code 11

6 Conclusion **13**

1 Project Description

In this project, the branching hardware for the Otter CPU was made. The branch condition generator was created to check the condition given two different source registers. This then resulted in an output of whether or not the values were equal, less than, or less than unsigned. The branch address generator was created to decide where to branch when a branch condition is met. In other words, this unit would modify the program counter given the program counter, j type immediate, b type immediate, i type immediate, and the source register 1. These modules were then formally tested using the SymbiYosys suite and proved to be functional.

2 Structural Design

2.1 Branch Address Generator Elaborated Design

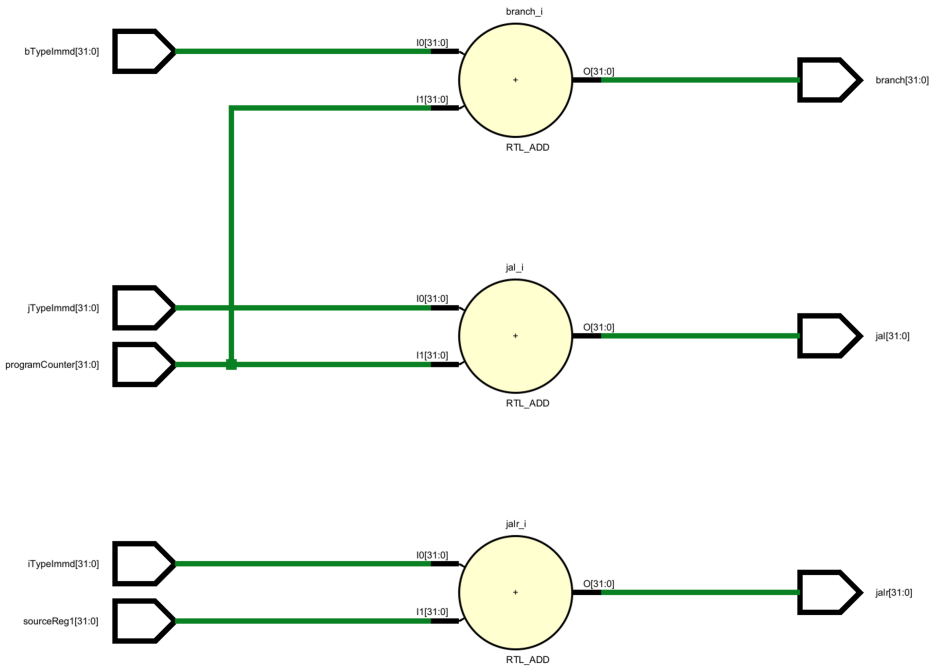

Figure 1: Branch Address Generator Elaborated Design

The branch address generator combines signals to output a new location for the program counter to jump to. From the elaborated design you can see this process as the different immediate values and registers are combined to get the respective outputs.

2.2 Branch Condition Generator Elaborated Design

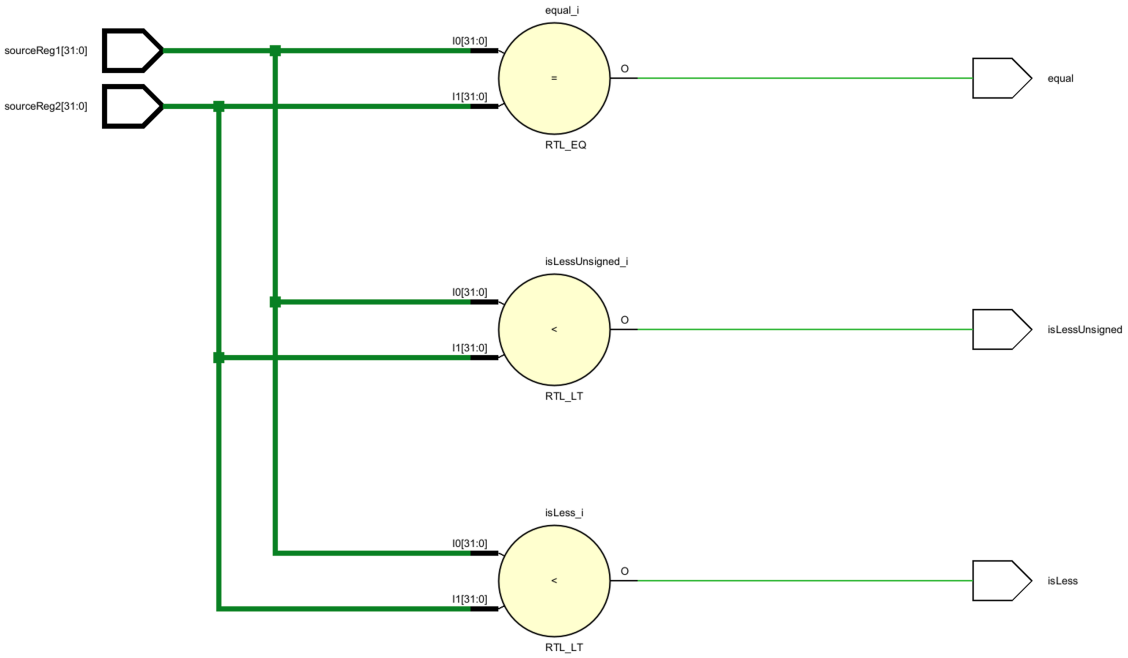

Figure 2: Branch Condition Generator Elaborated Design

The branch condition generator compares the two source registers and then outputs different one-bit values that represent different comparisons. This information can then be used to determine if a branch should be made.

2.3 Otter Memory Module Signal Table

Signal Name	Size (Bits)	Purpose
MEM_CLK	1	This is the clock input for the module
MEM_RDEN1	1	This is the read enable for the instruction
MEM_RDEN2	1	This is the read enable for the data
MEM_WE2	1	This is the write enable
MEM_ADDR1	14	This is the instruction word memory address that connects to the PC
MEM_ADDR2	32	This is the data memory address
MEM_DIN2	32	This is the data that will be saved
MEM_SIZE	2	This determines the size of the memory that will be saved (0-Byte, 1-Half Word, 2-Word)
MEM_SIGN	1	This determines whether or not the memory is signed
IO_IN	32	This gets data from the IO of the hardware
IO_WR	1	This tells the hardware whether data is being read or written
MEM_DOUT1	32	This outputs the instruction that is requested by the PC
MEM_DOUT2	32	This outputs the data that is requested by the data address

Table 1: Otter Memory Module Signal

3 Synthesis Warnings

3.1 Branch Address Generator Synthesis Warnings

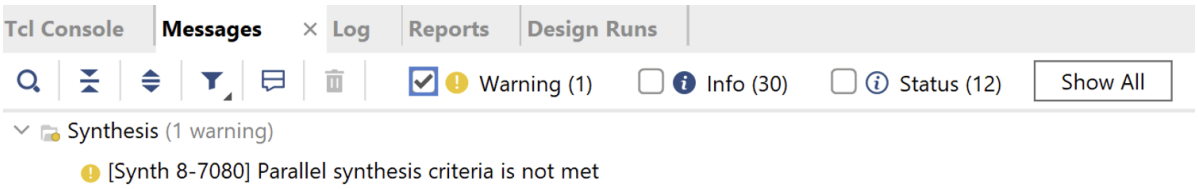

Figure 3: Branch Address Generator Synthesis Warnings

3.2 Branch Condition Generator Synthesis Warnings

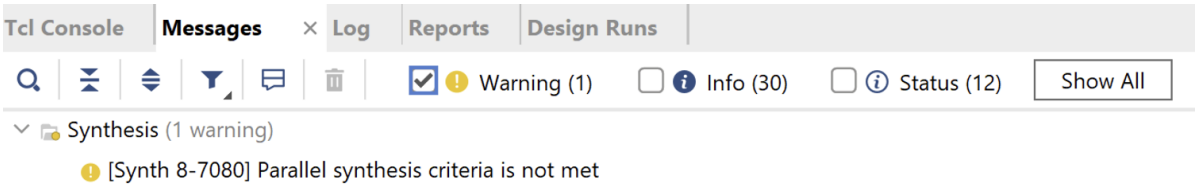

Figure 4: Branch Condition Generator Synthesis Warnings

4 Verification

In this section, the verification methodology and scope of the verification is explained. A different type of verification was used in this project that allowed for an increased level of confidence when testing the modules.

4.1 Verification Methodology and Scope

Using SymbiYosys, formal verification could be implemented to rigorously test the respective modules. With this testing, the desired function of the code is defined in a separate system verilog file and then that is used to drive an engine to try and break the module that is under test. In this case, two of the three available modes for testing were used Bounded Model Checking(BMC) and Cover.

Bounded Model Checking is the most basic of the formal testing suit and allows attempts to break the module by using as many different combinations of inputs and outputs as possible. These are not random though, they have mathematical backing to try and break the module that they are testing.

Cover checking test for coverage of an individual case. This tells the engine to try and figure out how to get to the state that is defined in the code. This is a different process and is more akin to a precise test as you are checking for one thing rather than trying to break a module by any means.

A general verification process starts with the creation of the module that will be tested. The next step in this process is to generate a formal file that describes the expected behavior of the module that you want to test. It is worth noting that if you want to emphasize test-driven development, this step can be done before the actual programming of the module. The final step is to set up a configuration file that will be used to run the respective tests and set up different criteria. Combined all of these steps allow you to use the SymbiYosys engine to formally verify.

4.2 Bounded Model Checking

For bounded model checking an `assert` statement was used to tell the verification engine to try and break the logic in the test file. An example of this can be seen below in Listing 1.

Listing 1: System Verilog Arithmetic Logic Unit Test Case File

```
always @(*) begin
    // Check for the jal branch verification
    assert(jal == programCounter +jTypeImmd);

    // Check for the branch address verification
    assert(branch == programCounter + bTypeImmd);

    // Check for the jalr  verification
    assert(jalr == sourceReg1 + iTypeImmd);
end
```

This code will tell the engine what the desired behavior of the module code should be and then will try to break the code by checking different aspects of the inputs and outputs. If a failure is found then a waveform is outputted with the specific conditions needed to break the module.

To see this in action, let's introduce a bug into the `BranchAddressGen.sv` file by changing how the jal line generates its value. In this case, a simple `+ 1` was added to bring the code out of specification.

Listing 2: Branch Address Generator Introduced Bug

```
    // Generate the jal address
    assign jal = jTypeImmd + programCounter + 1;
```

Now running this through the verification engine, we get the waveform seen in Figure 5.

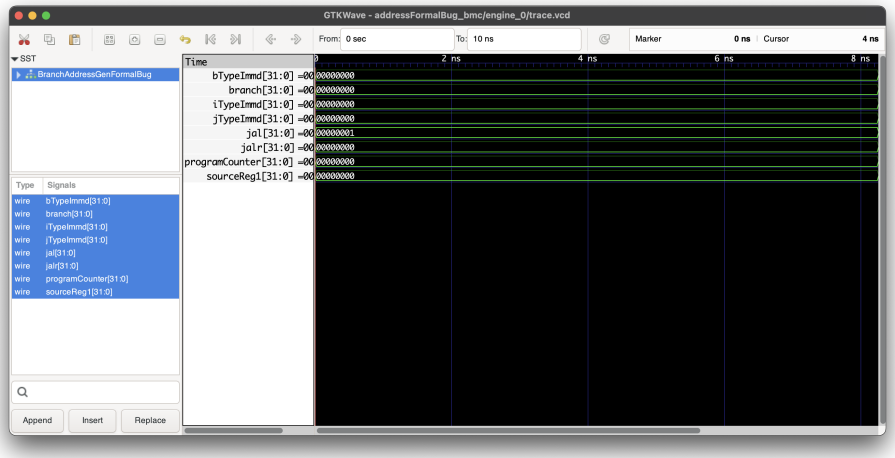

Figure 5: Branch Address Generator Introduced Bug Waveform

Examining Figure 5 we can see that when all zeros are input into the system, we have an output of 1 on the `jal` line which is not what we specified in our testbench when we refer to line 3 of Listing 1. The engine was able to find this and then error out and show us where the problem is. We can then go to that part of the code and fix that section. The engine was able to break our module and showed that the module that we set up did not work as we described.

4.3 Coverage Verification

Switching gears, let's look at the cover command in the formal verification suit. This command is used to specify a state that we are looking for and then try to achieve that state in the verification of the code. This is useful when testing to see whether certain conditions are achievable in your code. For example in the snippet below we wish to see if it is possible to reach a state where both the signed and unsigned less-than lines are set.

Listing 3: System Verilog Formal Verification Code for Branch Condition Generator

```
1     // Test if both isLessUnsigned and isLess can be reached at the same time
2     cover(isLessUnsigned == 1'b1 && isLess ==1'b1);
```

When running this kind of test, a waveform is generated upon success. When the engine is able to reach the state, it will output what steps it took to get there. The waveform for this specific code is shown below in Figure 6.

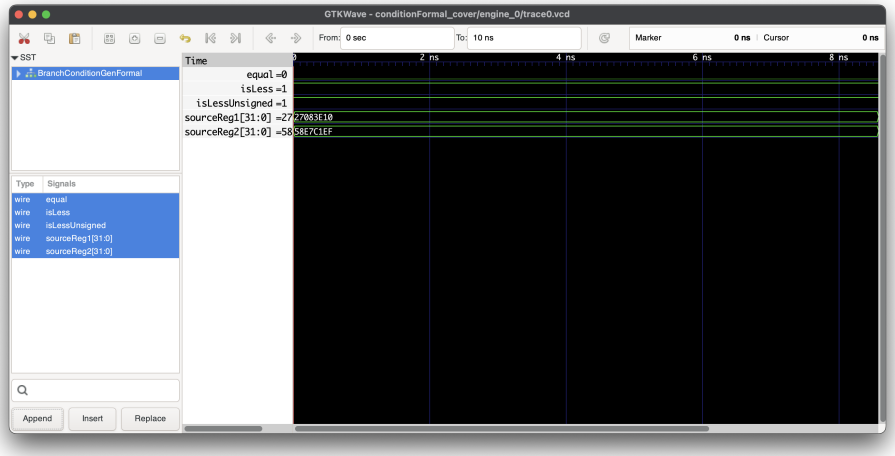

Figure 6: Branch Condition Generator Cover Test Waveform

This is useful when trying to make sure a critical path can be taken in your module. When designing a complex design, it is possible to introduce unexpected ways to reach a state and this helps with that. You can have the engine try to get to the state and check is that is a valid path. This kind of testing informs you on how to better build your modules.

5 Source Code

5.1 Branch Address Generator Source Code

Listing 4: System Verilog Code for Branch Address Generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly SLO
// Engineer: Ethan Vosburg
//
// Create Date: 02/07/2024 05:56:05 PM
// Module Name: BranchConditionGen
// Project Name: Branching Hardware
// Target Devices: Basys 3
// Description: This module will compare two 32-bit signed and unsigned numbers
// and then output the result of the comparison to three different outputs
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////


module BranchAddressGen(
    // Inputs
    input [31:0] programCounter,
    input [31:0] jTypeImmd,
    input [31:0] bTypeImmd,
    input [31:0] iTypeImmd,
    input [31:0] sourceReg1,

    // Outputs
    output logic [31:0] jal,
    output logic [31:0] branch,
    output logic [31:0] jalr
);

    // Generate the jal address
    assign jal = jTypeImmd + programCounter;

    // Generate the branch address
    assign branch = bTypeImmd + programCounter;

    // Generate the jalr address
    assign jalr = iTypeImmd + sourceReg1;

endmodule
```

Listing 5: System Verilog Formal Verification Code for Branch Address Generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly SLO
// Engineer: Ethan Vosburg
//
// Create Date: 02/9/2024 09:56:05 PM
// Module Name: BranchAddressGenFormal
// Project Name: Branching Hardware
// Target Devices: Basys 3
// Description: This module will formally verify that the branch address
// generator module will work correctly.
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////


`include "BranchAddressGen.sv"

module BranchAddressGenFormal(
    // Inputs
    input [31:0] programCounter,
    input [31:0] jTypeImmd,
    input [31:0] bTypeImmd,
    input [31:0] iTypeImmd,
    input [31:0] sourceReg1,

    // Outputs
    output logic [31:0] jal,
    output logic [31:0] branch,
    output logic [31:0] jalr
);


BranchAddressGen BranchAddressGen(
    .programCounter(programCounter),
    .jTypeImmd(jTypeImmd),
    .bTypeImmd(bTypeImmd),
    .iTypeImmd(iTypeImmd),
    .sourceReg1(sourceReg1),
    .jal(jal),
    .branch(branch),
    .jalr(jalr)
);


always @(*) begin
    // Check for the jal branch verification
    assert(jal == programCounter +jTypeImmd);

    // Check for the branch address verification
    assert(branch == programCounter + bTypeImmd);

    // Check for the jalr  verification
    assert(jalr == sourceReg1 + iTypeImmd);
end

endmodule
```

Listing 6: SymbiYosys Config File for Branch Address Generator

```
[tasks]
bmc

[options]
bmc: mode bmc
depth 10

[engines]
smtbmc boolector

[script]
read -formal -sv BranchAddressGenFormal.sv
prep -top BranchAddressGenFormal

[files]
BranchAddressGen.sv
BranchAddressGenFormal.sv
```

5.2 Branch Condition Generator Source Code

Listing 7: System Verilog Code for Branch Condition Generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly SLO
// Engineer: Ethan Vosburg
//
// Create Date: 02/07/2024 05:56:05 PM
// Module Name: BranchConditionGen
// Project Name: Branching Hardware
// Target Devices: Basys 3
// Description: This module will compare two 32-bit signed and unsigned numbers
// and then output the result of the comparison to three different outputs
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////


module BranchConditionGen(
    // Inputs
    input [31:0] sourceReg1,
    input [31:0] sourceReg2,

    // Outputs
    output equal,
    output isLess,
    output isLessUnsigned
    );

    // Flag equal out as high if reg1 is equal to reg2
    assign equal = (sourceReg1 == sourceReg2) ? 1'b1 : 1'b0;

    // Flag isLess is comparison is valid
    // Note: Numbers are interpreted as unsigned by default
    assign isLess = ($signed(sourceReg1) < $signed(sourceReg2)) ? 1'b1 : 1'b0;

    // Flag isLessUnsigned if comparison is true
    assign isLessUnsigned = (sourceReg1 < sourceReg2) ? 1'b1 : 1'b0;
endmodule
```

Listing 8: System Verilog Formal Verification Code for Branch Condition Generator

```
‘timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: Cal Poly SLO
// Engineer: Ethan Vosburg
//
// Create Date: 02/9/2024 09:56:05 PM
// Module Name: BranchConditionGenFormal
// Project Name: Branching Hardware
// Target Devices: Basys 3
// Description: This module will formally verify that the branch condition
// generator module will work correctly.
//
// Revision:
// Revision 0.01 - File Created
//
//////////////////////////////////////////////////////////////////////////////////


‘include "BranchConditionGen.sv"

module BranchConditionGenFormal(
    // Inputs
    input [31:0] sourceReg1,
    input [31:0] sourceReg2,

    // Outputs
    output equal,
    output isLess,
    output isLessUnsigned
);

BranchConditionGen BranchAddressGen(
    .sourceReg1(sourceReg1),
    .sourceReg2(sourceReg2),
    .equal(equal),
    .isLess(isLess),
    .isLessUnsigned(isLessUnsigned)
);

always @(*) begin
    // Check for equal condition
    if (sourceReg2 == sourceReg1) begin
        assert(equal == 1’b1);
    end

    // Check for less than
    if ($signed(sourceReg1) < $signed(sourceReg2)) begin
        assert(isLess == 1’b1);
    end

    // Check for less than unsigned
    if (sourceReg1 < sourceReg2) begin
        assert(isLessUnsigned == 1’b1);
    end

    // Test if both isLessUnsigned and isLess can be reached at the same time
    cover(isLessUnsigned == 1’b1 && isLess ==1’b1);
end
endmodule
```

Listing 9: SymbiYosys Config File for Branch Condition Generator

```
[tasks]
bmc
cover

[options]
bmc: mode bmc
cover: mode cover
depth 10

[engines]
smtbmc boolector

[script]
read -formal -sv BranchConditionGenFormal.sv
prep -top BranchConditionGenFormal

[files]
BranchConditionGen.sv
BranchConditionGenFormal.sv
```

6 Conclusion

Both the branch address generator and branch condition generator hardware were constructed and formally verified. The goal of this hardware was to make decisions on where to branch when a machine code instruction requires branching. Both modules were found to be working as described. Unique to this assignment, formal verification was added and applied to the modules to ensure that they were working properly. Because of this behavioral specification, there was no need for traditional simulation. All code for this assignment can be found here.