RISC-V Functional Coverage Analysis

Brian Humphreys, David Roster
ECE 156A
Mentor: Ahmed

Table of Contents

Tester:

Checker:

Bypass Signals:

Strategy:

Data Bypass Happened (look for bypass signals):

Tester:

Checker:

Required Cover Points:

Adder:

Strategy:

For coverage of the adder portion, we will inject constraints into the verilog file, `vscale_alu.v` that will set a flag variable `COVER_POINT_ALU_UNSIGNED_OVERFLOW` to 1 if the cout portion of the output is 1, which implies overflow. The flag variable will stay 0 if there is no overflow. We will test for both signed and unsigned values simultaneously in the same case statement that is activated when the ALU operation is ADD.

Unsigned Overflow:

Tester:

The following is the vector file `add_unsigned_hex.txt` read into `vscale_hex_tb.v`:

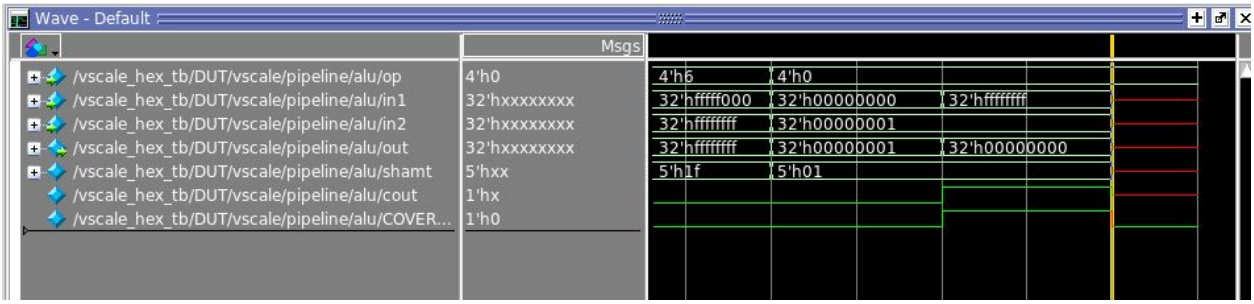
Assembly Coverage:	Hex Representation
addi $3 $0 0 #Initialize $3 lui $3 1048575 #load 0xfffff000 in $3 ori $3 $3 4095 #make $3 equal to 0xffffffff addi $4 $0 1 #initialize $4 to be 1 add $5 $3 $4 #overflow register 5	00000193 fffff1b7 fff1e193 00100213 004182b3

Checker:

The following is a verilog snippet implemented in the case statement of the file `vscale_alu.v`. When ever the ALU is given the command to add, the checker will see if the cout is a 1. If cout is 1 then the flag variable is set:

```
always @(*) begin
   case (op)
     `ALU_OP_ADD : begin

            {cout,out} = in1 + in2;


            assign COVER_POINT_ALU_UNSIGNED_OVERFLOW=0;

            if(cout==1) begin
                    assign COVER_POINT_ALU_UNSIGNED_OVERFLOW=1;
            end
    end
```

The image to the left depict registers $3 and $4 added to cause an overflow in $5.
The following is a wave output of the ALU which made use of registers 3 and 4 as the operands and register 5 as the output. Whenever cout is set to 1, the flag variable at the bottom of the wave list is also set to 1 with a tiny bit of delay:



Signed Overflow:

Tester:

The following is the vector file `add_signed_hex.txt` read into `vscale_hex_tb.v`:

Assembly Coverage:	Hex Representation:
lui $3 262144 # load 0x40000000 into $3 lui $4 262144 # load 0x40000000 into $4 add $5 $3 $4 # adding results in signed overflow	400001b7 40000237 004182b3

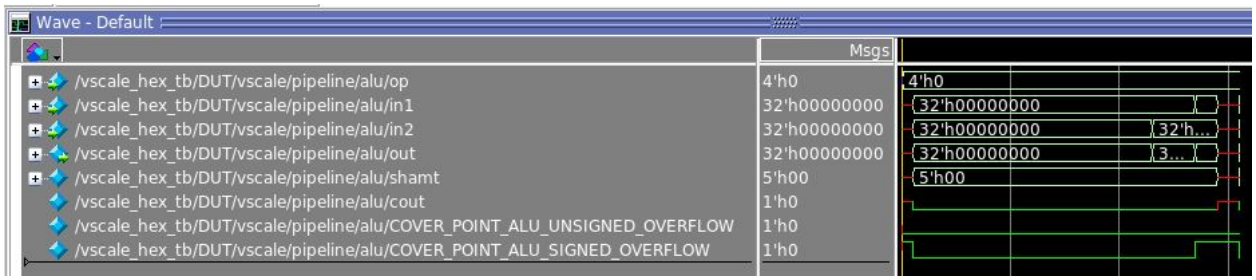
Checker:

The following is a verilog snippet implemented in the case statement of the file `vscale_alu.v`. At first, the flag variable is set to 1. When ever the ALU is given the command to add, the checker will see if the most significant digits of the operands do not equal or if the most significant digits of the operands and the result are all the same. If either of these conditions are true, then the flag variable is unset.

```
always @(*) begin
   case (op)
     `ALU_OP_ADD : begin

            {cout,out} = in1 + in2;


            assign COVER_POINT_ALU_UNSIGNED_OVERFLOW=0;
            assign COVER_POINT_ALU_SIGNED_OVERFLOW=1;

            if(cout==1) begin
                    assign COVER_POINT_ALU_UNSIGNED_OVERFLOW=1'b1;
            end if((in1[31] != in2[31]) || ((in1[31] == out[31]) && (in2[31] == out[31]))) begin
                    assign COVER_POINT_ALU_SIGNED_OVERFLOW=1'b0;
            end
     end
```

The Register snippet to the left depicts Registers $3 and $4 being added and stored into $5 causing an overflow with the correct lessor bits.

In this test instance, the assembly code causes a signed overflow by adding 0x40000000 to itself. This is a positive number since the most significant digit is a 0. When added to itself, it will become 0x80000000 which is negative 0 in signed binary notation. The wave for causes an overflow for signed binary but now overflow for unsigned binary as seen in the waveform below.



Subtractor:

Strategy:

The strategy here is essentially the same as the signed and unsigned add. We are going to test for results that should be negative in the ALU case statement that is activated when the operation is SUB.

Negative Result:

Tester:

The following is the vector file `sub_neg_result_hex.txt` read into `vscale_hex_tb.v`:

Assembly Coverage:	Hex Representation:
addi $3 $0 10 # load 10 into $3 addi $4 $0 100 # load 100 into $ 4 sub $5 $3 $4 # subtract bigger from smaller	00a00193 06400213 404182b3

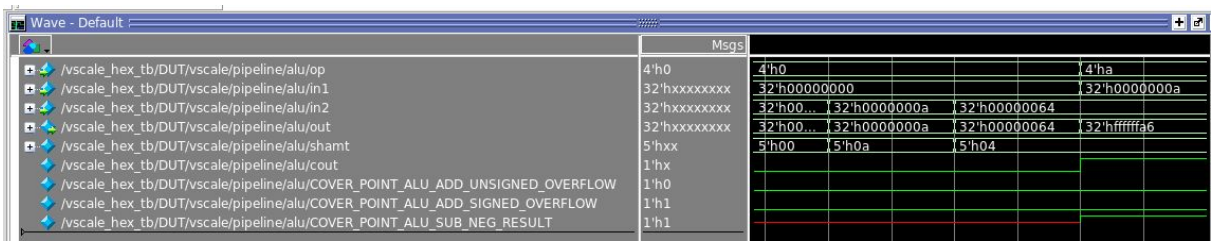
Checker:

In this coverage test, I wanted to see if the ALU subtract method could take in two positive operands, subtract the bigger from the smaller value, and then output the proper 2's complement number. The following snippet of verilog code shows the checker that made sure the conditions were correct for the cover point flag to be set to 1.

```
`ALU_OP_SUB : begin
        {cout, out} = in1 - in2;
        assign COVER_POINT_ALU_SUB_NEG_RESULT = 0;

        if((in2 > in1) && out[31]==1) begin
                assign COVER_POINT_ALU_SUB_NEG_RESULT = 1;
        end
end
```

The registers to the left display $4 being subtracted from $3 and storing the result in $5. The resulting value is the hex 2's comp representation of -90.

In this instance, we should see the negative variable flag be set to 1 when the registers are subtracted and that is exactly what we observe in the waveform below:



Multiplier:

Strategy:

The strategy here is similar to the ALU operations as mentioned above, except that the verilog checker will be placed into a different file. Since MUL and DIV operations are not performed in the ALU but instead a separate component, we will navigate into the file where these operations take place; namely, `vscale_mul_div.v`. Our checker is similar to the above cover points in that the checker will be placed in the case statement where MUL is performed.

Zero Output:

Tester:

The following is the vector file `mul_result_zero_hex.txt` read into `vscale_hex_tb.v`:

Assembly Coverage:	Hex Representation:
addi $3 $0 55 # set $3 as arbitrary value mul $5 $3 $0 # multiply $3 by 0, store to $5	03700193 020182b3

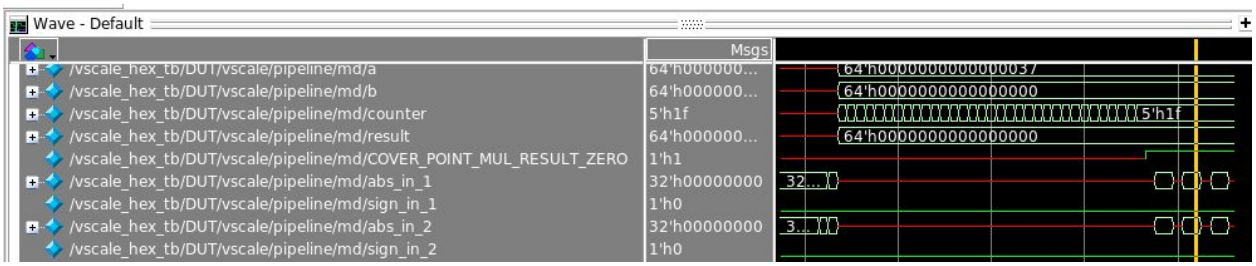
Checker:

In this cover point, all we are checking for is whether a number multiplied by zero is zero. The number can be any number, even zero. We only care about the result so this is what we check for in the case statement. The following verilog snippet depicts the checker in the file `vscale_mul_div.v`.

The premise is simple. If the output is 0, set the cover point flag to 1. If the result is not 0, keep the flag as 0. The register shows the result in $5.

```
s_setup_output : begin
   result <= {`XPR_LEN'b0,final_result};
   assign COVER_POINT_MUL_RESULT_ZERO = 0;
   if(result == 0) begin
        assign COVER_POINT_MUL_RESULT_ZERO = 1;
   end
end
```

Once the counter finished, the multiplication was finished. And once the the MUL operation was finished, the result was ready to be checked for 0 output. In our case, the flag was set, which means the checker works!



XOR:

Strategy:

The strategy here is to yet again, place a checker in the ALU file `vscale_alu.v` under the XOR case and to check for results that are either all 1's or all 0's. The all zero result should arise when the two values being XORed are exactly the same value. The all ones result should appear when when the two values being XORed are 1's complement of each other. The verilog snippet will be placed below in this section since I implemented both the zero and all ones checker together. Let's see what we get from our checkers.

```
`ALU_OP_XOR : begin
        out = in1 ^ in2;
        assign COVER_POINT_ALU_XOR_ZERO_RESULT = 0;
        assign COVER_POINT_ALU_XOR_ALL_ONES_RESULT = 0;
        if(out == 32'h00000000) begin
                assign COVER_POINT_ALU_XOR_ZERO_RESULT = 1;
        end if(out == 32'hffffffff) begin
                assign COVER_POINT_ALU_XOR_ALL_ONES_RESULT = 1;
        end
end
```

Zero Result:

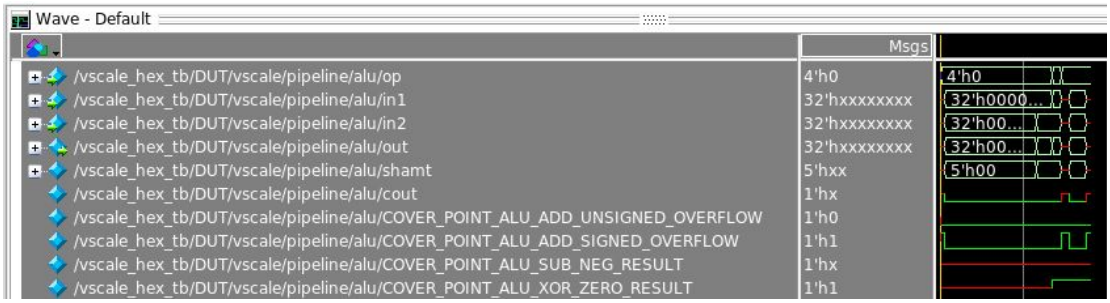
Tester:

The following is the vector file `mul_result_zero_hex.txt` read into `vscale_hex_tb.v`:

Assembly Coverage:	Hex Representation:
addi $3 $0 240 addi $4 $0 240 xor $5 $4 $3 # XOR same values to get 0	0f000193 0f000213 003242b3

Checker:

The values 240 are loaded into two registers that are then XORed. The result is as expected. The flag at the bottom of the waveform informs us that the XORed value has been set to 0, verifying our results. The only thing that may seem out of the ordinary here is the spikes in the signed overflow flag. This occurs only because the flag is set to 1 as default until it is proved to not be overflown.



All Ones Result:

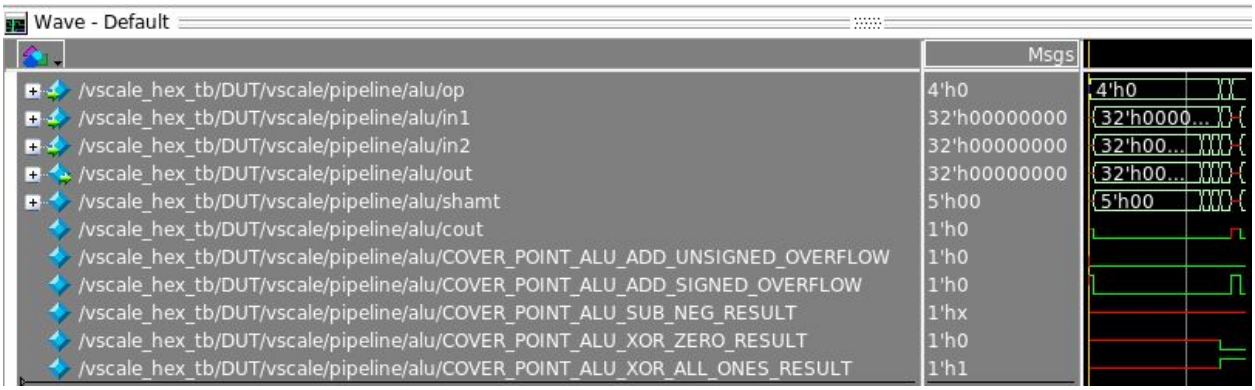
Tester:

The following is the vector file `mul_result_zero_hex.txt` read into `vscale_hex_tb.v`:

Assembly Coverage:	Hex Representation:
addi $3 $0 3855 addi $4 $0 240 xor $5 $4 $3 # XOR 1's comp #'s to get all 1's	0f000193 0f000213 003242b3

Checker:

The result of this XOR all ones checker, as can be seen below, works as expected. Two numbers, 0x000000f0 and 0x00000f0f are ones complements of each other and the result should be all ones. This expectation is met and can be seen in the register $5 which has resulted in 0xffffffff and the waveform which displays the all ones checker as active. Again, there is an odd jump in the signed overflow flag here for a moment but it corrects itself.



JUMP:

Strategy:

There is a file in the verilog simulation called `vscale_PC_mux.v` which handles jump and branch cases. In this cover point, we want to test for jumps (JAL and JALR) so I have placed flag assigns whenever these case statements are triggered as seen below.

```
always @(*) begin
   case (PC_src_sel)
     `PC_JAL_TARGET : begin
        base = PC_DX;
        offset = jal_offset;
             assign COVER_POINT_JUMP_OCCURED = 1;
     end
     `PC_JALR_TARGET : begin
        base = rs1_data;
        offset = jalr_offset;
             assign COVER_POINT_JUMP_OCCURED = 1;
     end
```

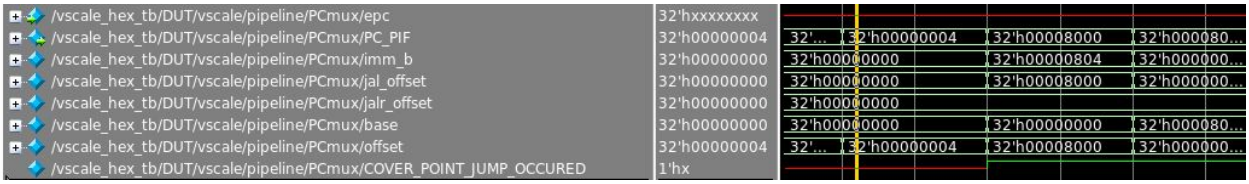
A jump happened (Check for PC value):

Tester:

Assembly Coverage:	Hex Representation:
jal $5 8	000082ef

Checker:

In this case, we use a JAL command with an arbitrary base address and an offset of 8 words. In the waveform shown below, it can be seen that the jump flag has been triggered and the offset has been set to 8. The blue variable box confirms this.



Bypass Signals:

Strategy:

In this section, my goal is to cause a data hazard. This can happen when you load a value into a register and then immediately try to perform an operation with the contents in that register. We are going to attempt to cause this and see if our microprocessor can handle this. The file `vscale_ctrl.v` as a variable that is set whenever data bypass occurs. We are going to provoke this by adding values from register that were just loaded.

```
// Hazard logic

assign load_in_WB = dmem_en_WB && !store_in_WB;

assign raw_rs1 = wr_reg_WB && (rs1_addr == reg_to_wr_WB)
  && (rs1_addr != 0) && uses_rs1;
assign bypass_rs1 = !load_in_WB && raw_rs1;

assign raw_rs2 = wr_reg_WB && (rs2_addr == reg_to_wr_WB)
  && (rs2_addr != 0) && uses_rs2;
assign bypass_rs2 = !load_in_WB && raw_rs2;

always @(*) begin
     if(bypass_rs1 ==1 || bypass_rs2 ==1 )begin
            assign COVER_POINT_DATA_BYPASS_HAPPENED=1;
     end
end
```

Data Bypass Happened (look for bypass signals):

Tester:

Assembly Coverage:	Hex Representation:
addi $3 $0 10 addi $4 $0 12 # add value into $4 add $5 $4 $3 # possible data hazard at $4	00a00193 00c00213 003202b3

Checker:

The values loaded in could have been virtually any value. The real test here is to see if the value is completed stored in a register before being used by the very next instruction. The following shows the input signals and all of the additional inputs that determine if bypass is reached.