Lab 7 (All Sections) Prelab: Verilog Review and ALU Datapath and Control

Name:

Yerania Hernandez

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Yerania Hernandez

1 Objective

The objective of this lab is to gain further experience with Verilog HDL and VCS. In particular, you will explore procedural Verilog, and simulation timing constructs. For this lab you are expected to have some knowledge of Verilog programming. You are advised to review the Verilog lecture videos posted on the course website as needed.

In this lab you will implement various datapath elements, including the register file, sign extender and ALU module with its control block. For this lab you are expected to be familiar with the ALU design (see Chapter C.5 of the textbook, on the CDROM). In addition, you should have a thorough understanding of the MIPS instruction set.

2 Verilog Review

2.1 Procedural Blocks

In digital design there are two types of elements: *combinational elements* and *sequential elements*. Verilog provides mechanisms to model these elements using the *always*, *assign* and *initial* statements:

- Combinational elements can be modeled using *assign* and *always* statements.

- Sequential elements can be modeled using only *always* statements.
- Non-synthesizable constructs can use *initial* statements.

2.1.1 Initial statements

An initial block as the name suggests, is executed only once, at the beginning of the simulation. This is usually useful for writing test benches. If we have multiple initial blocks then all of them are executed in parallel at the beginning of the simulation.

```
initial begin
clk = 0;
reset = 0;
req_0 = 0;
req_1 = 0;
end
```

In the above example, at the beginning of simulation, (i.e., at time zero), all the variables inside the block marked by *begin* and *end* are driven zero.

2.1.2 Always statements

As the name suggests, an *always* block executes periodically throughout the program, unlike *initial* blocks which are executed only once. A second difference is that an *always* block should have a sensitivity list or a delay associated with it. The sensitivity list specifies the conditions which trigger the execution the *always* block.

```
always @ (a or b or sel)
begin
y = 0;
if (sel == 0) begin
y = a;
end else begin
y = b;
end
end
```

As shown above, the @ symbol after reserved word *always* indicates that the block will be executed whenever the condition in parenthesis after symbol @ is satisfied. In this case whenever the variables **a** or **b** or **sel** are modified the *always* block is executed.

Note that *always* blocks cannot drive **wire** data type, but instead can drive **reg** data type. In addition, we note that the statements that need to be executed sequentially must be enclosed inside a begin-end block.

2.1.3 Continuous assignments

The Verilog includes the *continuous assign* construct, which usually represent the combinational logic. For example, consider the following assignment:

```
assign x= y & z
```

This assignment describes an "AND" gate whose inputs are connected to wires y and z and whose output is connected to wire x. In continuous assignments, the left-hand size of the expression must be a wire. As a rule, the register are given values at discrete times, while the nets are always given a value.

The continuous assignments often use the following conditional operator:

```
condition ? expression1 : expression2
```

The condition is the logical expression which has the true or false result. If the result is true, expression1 is evaluated, otherwise expression2 is evaluated. For example, a multiplexor with input signals $in0$, $in1$, output signal out, and control signal s can be described by the following statement:

```
assign out = s ? in1 : in0;
```

2.2 Timing Control

The timing control is an important and useful feature provided by Verilog. Timing control is needed to simulate propagation or switching delay in chips/gates or wires. Timing control is also used in testbench code to simulate timing delays to produce input waveforms.

The Verilog language provides three types of explicit timing control. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The second type of timing control is the event expression, which allows statement execution. The third type is a wait statement which suspends execution until a certain condition is met. Simulated time can *only* progress by one of the following:

1. Gate or wire delay, if specified;
2. A delay control, introduced by the # symbol;
3. An event control, introduced by the @ symbol;
4. The wait statement.

2.2.1 Delay Control (#)

A delay control expression specifies the time duration between initially encountering the statement and when the statement actually executes. For example:

```
#10 A = A + 1;
```

Specifies to delay 10 time ticks before executing the procedural assignment statement. The # may be followed by an expression with variables.

2.2.2 Timescale

In Verilog there are no default units of for the "ticks", instead they are typically defined explicitly in the testbench for each particular design with a "timescale" Verilog directive thus:

```
‘timescale time_unit base / precision base
```

This command specifies the time units and precision of time tick delays:

time_unit The amount of time a tick delay of 1 represents. Must be 1 10 or 100.

base The time base for each unit. Must be: s ms us ns ps or fs.

precision Represents how many decimal points of precision to use relative to the time units.

Example timescale with 1ns units and 10ps of precision:

```
‘timescale 1 ns / 10 ps
```

2.2.3 Events

The execution of a procedural statement can be triggered with a value change on a wire or register, or the occurrence of a named event.

Some examples

```
@r begin                    //Controlled by any value change in r
A = B & C;
end

@ (posedge clock2) A = B & C;      //Controlled by positive edge of clock2

@ (negedge clock3) A = B & C;      //Controlled by negative edge of clock3

forever @(negedge clock)  //Controlled by negative egde
begin
A = B & C;
end
```

Note: In the forms using *posedge* and *negedge*, the must be followed by a 1-bit expression, typically a clock.

2.2.4 User-defined events

Verilog also provides features to name an event and then to trigger the occurrence of that event. We must first declare the event:
event event6;
To trigger the event, we use the -> symbol:
->event6; To control a block of code for this event, we use the @ symbol:

```
@(event6) begin
<some procedural code>
end
```

We assume that the event occurs in one thread of control, i. e., concurrently, and the controlled code is in another thread. Several events may be or-ed inside the parentheses (after @ symbol).

2.2.5 Wait Statement

The wait statement allows a procedural statement or a block to be delayed until a condition becomes true.

```
wait (A == 3)
begin
A = B & C;
end
```

2.3 Blocking and non-blocking Assignments

In the previous sections we saw the assignments which use the "=" operator. Such assignments are referred to as *blocking assignments*. With such assignments, the next statement after the assignment statement does not begin executing until the assignment has finished. The blocking assignments have the form

```
left-hand side =[control] right-hand side;
```

For example, consider the following assignment with intra-assignment delay:

```
y = #10 x + z;
```

The execution of this statement requires the following steps:

1. Evaluation of the right-hand side and saving the result. In this case, the value of $x + z$ will be evaluated;
2. Wait for the specified delay - in this case 10 ticks;
3. Perform the assignment.

The execution of this statement includes waiting for 10 time ticks. That is, the next statement will execute only after 10 ticks. Thus, this statement blocks the next one until the waiting is over.

A blocking statement may wait for a specific event to occur. For example, consider the following statement:

```
y = @(posedge clk) y + z;
```

This statement first evaluates the right-hand side expression $(y + z)$ and then waits for rising edge of clk and performs the assignment.

Non-blocking assignments use the "`<=`" operator. Such assignments do not block the following statement even in case of intra-assignment wait. Non-blocking assignments have the following form:

```
left-hand side <=[control] right-rand side;
```

For example consider the following assignment with intra-assignment delay:

```
y <= #10 x+z;
```

The execution of this statement requires the following steps:

1. Evaluate the right-hand side and save the result. In this case, the value of $x + z$ will be evaluated;
2. Put the assignment on the waiting list to be executed at $t + 10$ time units, where t is the current time.
3. Continue the next assignment at time t.

2.4 System Tasks

Verilog provides a means for executing system level operations on the host machine during simulation via system tasks. All system tasks use the form `$<keyword>`. A commonly used system task is `$display`, which prints variables, strings, and expressions to the terminal window when invoked. The syntax for `$display` follows very closely to that of *printf* in C. Another useful system task is `$monitor`, which continuously monitors the signals specified in the parameter list and prints to the terminal whenever any of these signals change. `$monitor` needs to only be invoked once and can be toggle on and off by other system tasks not mentioned here. Another useful system task is `$time`, which provides the current simulation time when invoked. Finally, for exiting simulation, Verilog provides the `$finish` system task.

3 ALU Datapath

In this lab, you will design the 32-bit Arithmetic Logic Unit (ALU) as described in Chapter C.5 of the textbook. Your ALU will become an important part of the MIPS microprocessor that you will build in later labs, so it is advised to do this lab carefully.

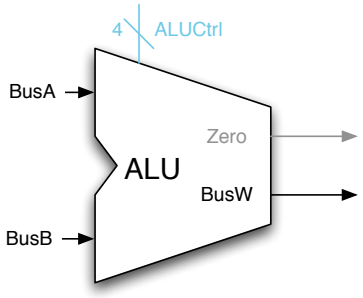


Fig. 1: ALU Block

3.1 ALU and ALU CONTROL Blocks

As you know, the ALU is the most crucial element in any processor data-path. Almost every instruction uses ALU for some processing. For example, ADD and ADDI use the ALU for addition, SUB for subtraction, OR and ORI for logical OR, etc.

As shown in Figure 1, an ALU module takes two 32-bit inputs (`BusA` and `BusB`), performs an operation as specified by the 4-bit `ALUCtrl` field, and outputs the result on the 32 bit output bus `BusW`. When the value on `BusW` is 0, the `Zero` signal is set to 1, otherwise it is set to 0.

The basic operations performed by an ALU are:

1. Addition
2. Subtraction
3. OR
4. AND
5. Set on less than (compare and set)

In this lab, we will be extending the ALU to implement 2 additional operations:

1. Shift left
2. Shift right

The shift left operation takes two values, A and B, each of which is 32-bits. A is shifted left B bits ($A << B$), and written to W. Shift right works in an similar manner but shifts to the right instead of the left.

4 Questions

1. The goal of the following model is generate a clock waveform that has the clock high 4 time units and low 4 time units, with the first rising edge at time 20. Complete the missing lines.

```
module q2;
reg clock;
initial begin
clock = 0;
...
...
...
end
end
endmodule
```

```
#20 clock = 1; //time 20 rising edge
always @(*)
#4 clock = ~clock; //at every 4 units must change
```

2. The following model produces a 4 bit output that counts the number of rising clock edges which have occurred. The output should be produced on the falling edge of the clock. Complete the missing fields.

```
module Counter (count, clock);
input clock;
output count;
reg [3:0] ....., count_reg;

initial count_reg = 0;

always @(posedge clock)
.....;

always @(....)
.... = count_reg;
endmodule
```

```
reg [3:0] count, count_reg;

always @(posedge clock)
count_reg = count_reg + 1;

always @(negedge clock)
count = count_reg;
```

3. Consider the following code snippets:

(a)
```
A = 1;
B = 2;
A = B;
B = A;
```

What will be the values in A and B after the above code is executed?

A = 2
B = 2

(b)
```
A = 11;
B = 22;
A <= B;
B <= A;
```

What will be the values in A and B after the above code is executed?

A = 22
B = 11

4. What will the following code display to the terminal?

```
module test1(i1,i2);
input i1, i2;
reg [31:0] a;
always @(i1) begin
#1 a = 1;
#5 a = a + 1;
#5 a = a + 2;
end
initial begin
$monitor ($time, "\t a = %d", a);
a = 2;
end
endmodule

module testtest1();
reg i1, i2;
test1 t1(i1,i2);
initial begin
i1 = 1;
#2 i1 = 0;
#10 $finish;
end
endmodule
```

```
0  a = 2
1  a = 1
6  a = 2
11 a = 4
```

5. What are the values of the variable *out* in the following code

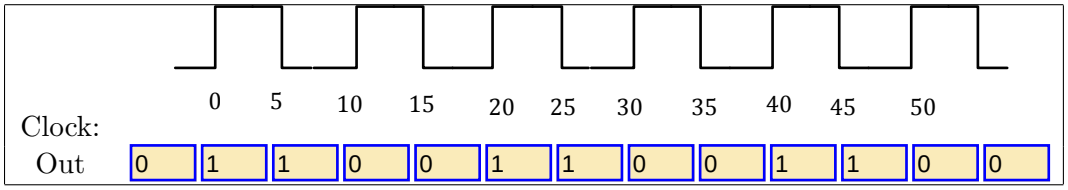
```
module test();
reg clock;
reg out;

initial begin
clock = 1'b1;
forever #5 clock = !clock;
end

inital begin
out = 1'b0;
#50 $finish;
end

always @(posedge clock) begin
#5 out = !out;
end

endmodule
```



6. Assume an MxN register file where M is the number of registers and N is the number of bits in each register. The register file can read 2 registers and optionally write a single register in each clock cycle on a clock edge if write is enabled. This results in 2 data outputs and 6 inputs.

   What are the different input and outputs of the register file? Include the width of each port. Express appropriate values in terms of M and N.

Inputs:
clock, enable
[M:0] rd_Reg1, rd_Reg2, wr_Reg
[N:0] wr_Data

Outputs:
[N:0] data_Reg1, data_Reg2

7. What is the operation that your ALU performs with the following instructions?

Instruction	ALU Action
LW (Load Word)	Add to calculate 32-bit address
SW (Store Word)	Add to calculate 32-bit addresss
BEQ (Branch of Equal)	Subtraction
SLT (Set on Less Than)	Subtraction
SLL	Multiplies by order of 2
SRL	Multiplies by order of 2 and sign extends

8. ALU Control
   The ALU control block takes 2 inputs

   - The function field from an R-TYPE instruction (bits [5:0] of the instruction).
   - ALUOP(2 bits) from the main control unit

   The output of the ALU control unit is a 4-bit signal, which directly controls the ALU. Give the ALU Ctrl values for the following operations:

   Note: For Shift operations (shift left and shift right), you may used any unassigned value. For the purposes of this lab, use 0011 for shift right, and 0100 for shift left.

Operation	ALU Control lines
Addition	0010
OR	0001
AND	0000
Subtract	0110
Set on Less Than	0111
NOR	1100
SRL	0100
SLL	0011

9. How would you extend the datapath in Figure 2 to implement the SLL and SRL instructions. **Note:** The ALU needs the first port to contain the value to shift and the second to contain the amount to shift by. Pay special attention to the data path for these values.

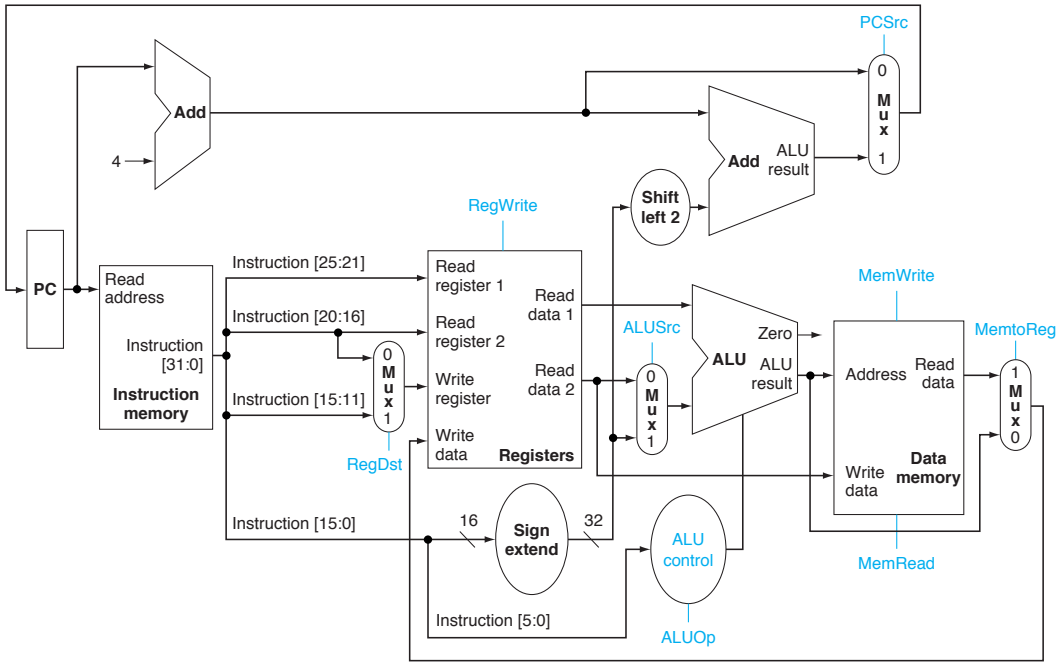


Fig. 2: ALU Datapath

This data path needs to include the shamt field instruction [10:6] to the ALU, in order to determine the shift amount. The ALU will then identify if the operation is SLL or SRL based on the ALU op field. As a result the ALU control needs to be modified in order to recognize the SLL and SRL opcodes, which can be accomplished by adding a control unit since it has the signal for the ALU op codes.

10. The ALUOP takes 3 values:

    `00` Override Add (AKA Implicit Add) (Used by load and store and `addi`).

    `01` Override Subtract (used by `beq`).

    `10` Use Function Field (used by R-Type instructions).

    `11` Override OR (Used by `ori`)

    Complete the following table for mapping the `ALUOP` and `FUNCT` fields to `ALUCTRL` signals.

Instruction	ALUOP (2 bits)	Funct Field Instruct[5:0]	ALU operation	ALU Control (4 bits)
LW	00	$XXXXXX$	add	0010
SW	00	$XXXXXX$	add	0010
BEQ	01	$XXXXXX$	subtract	0110
ADD	10	100000	addition	0010
SUB	10	100010	subtraction	0110
AND	10	100100	AND	0000
OR	10	100101	OR	0001
ORI	11	XXXXXX	OR	0001
SLT	10	101010	Subtract	0111
SLL	10	000000	Multiply by order	0011
SRL	10	000010	Divide by order of	0100

Note: You will use the above mapping to implement the ALU control block during inlab.

Test #	Test	ALUCtrl (4 bits)	A (32 bits)	B (32 bits)	W (32 bits)	Zero
1	ADD $0, 0$	2	0x00000000	0x00000000	0x00000000	1
2	ADD $0, -1$	2	0x00000000	0xFFFFFFFF	0xFFFFFFFF	0
3	ADD $-1, 1$	2	0xFFFFFFFF	0x00000001	0x00000000	1
4	ADD $FF, 1$	2	0x000000FF	0x00000001	0x00000100	0
5	SUB $0, 0$	6	0x00000000	0x00000000	0x00000000	1
6	SUB $1, -1$	6	0x00000001	0xFFFFFFFF	0x00000002	0
7	SUB $1, 1$	6	0x00000001	0x00000001	0x00000000	1
8	SLT $0, 0$	7	0x00000000	0x00000000	0x00000000	1
9	SLT $0, 1$	7	0x00000000	0x00000001	0x00000001	0
10	SLT $0, -1$	7	0x00000000	0xFFFFFFFF	0x00000000	1
11	SLT $1, 0$	7	0x00000001	0x00000000	0x00000000	1
12	SLT $-1, 0$	7	0xFFFFFFFF	0x00000000	0x00000001	0
13	AND 0xFFFFFFFF, 0xFFFFFFFF	0	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0
14	AND 0xFFFFFFFF, 0xCAFEBABE	0	0xFFFFFFFF	0xCAFEBABE	0xCAFEBABE	0
15	AND 0x00000000, 0xFFFFFFFF	0	0x00000000	0xFFFFFFFF	0x00000000	1
16	AND 0x12345678, 0x87654321	0	0x12345678	0x87654321	0x02244220	0
17	OR 0xF0F0F0F0, 0x0000FFFF	1	0xF0F0F0F0	0x0000FFFF	0xF0F0FFFF	0
18	OR 0x12345678, 0x87654321	1	0x12345678	0x87654321	0x97755779	0
19	SLL 0x12345678,0x2	3	0x12345678	0x00000002	0x48D159E0	0
20	SLL 0x80000000,0x3	3	0x80000000	0x00000003	0x00000000	1
21	SRL 0x00000001,0x3	4	0x00000001	0x00000003	0x00000000	1
22	SRL 0x00001234,0x6	4	0x00001234	0x00000006	0x00000048	0

Tab. 1: ALU Test Vectors

11. During the in-lab, you will be implementing the ALU block. As part of the design process, you need to develop an appropriate set of test vectors to verify basic functionality. Complete Table 1 (submit hardcopy or via email) to verify that all 7 ALU operations work as designed. Note that all values are expressed in **hexadecimal**. You will use this table to test the ALU Verilog code that you will implement during the in-lab **(make sure to keep a copy of your submitted answers)** .