Lab 7 : MIPS Datapath Components

Name:

Kraig Orcutt

Sign the following statement:

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

Kraig Orcutt

1 Introduction

The design in this lab will demonstrate the ways in which Verilog encoding makes hardware design more efficient. It is possible to design a 32-bit ALU from 1-bit ALUs. (i.e., you could program a 1-bit ALU that implements a full adder, chain four of these together to make a 4-bit ALU, and chain 8 of those together to make a 32-bit ALU.) However, it is easier (both in time and lines of code) to code it succinctly in Verilog. It is even easier, however, to design a module to be inefficient or incorrect.

2 Pre-requisite

For this lab you are expected to know Verilog programming and understand how to use VCS based upon the last lab and understand the ALU from Chapter C.5 of the textbook.

3 Verilog Review

The following Verilog modules have errors. Explain the errors in each module and provide a fix for each.

1. Logical gates

```
module gates(input      [3:0] a, b,
output reg [3:0] y1, y2, y3, y4, y5);
always @(b)
begin
y1 = a & b;
y2 = a | b;
y3 = a ^ b;
y4 = ~(a & b);
y5 = ~(a | b);
end
endmodule
```

The always should run when both a and b change

always @(a or b)

2.

```
module mux2(input [3:0] d0, d1,
input           s,
output reg [3:0] y);

always @(posedge s)
if (s) y = d1;
else   y = d0;
endmodule
```

The mux output can change when any input changes,

always @(s or d0 or d1)

3. Finite State Machine

```
module FSM(input       clk,
input       a,
output reg out1, out2);
reg  state, nextstate;

// state register
always @(posedge clk, posedge reset)
if (reset)
state <= 1'b0;
else
state <= nextstate;

// next state logic
always @(*)
case (state)
1'b0: if (a) nextstate <= 1'b1;
else nextstate <= 1'b0;
1'b1: if (~a) nextstate <= 1'b0;
else nextstate <= 1'b1;
endcase

// output logic (combinational)
always @ (*)
if (state == 0) out1 = 1'b1;
else            out2 = 1'b1;
endmodule
```

The FSM needs to stay at the current state before it determines the next state

after always @(*), use a blocking assignment to assign nextstate=state

4. Priority Encoder

```
module priority(input      [3:0] a,
output reg [3:0] y);
always @(*)
if       (a[3]) y = 4'b1000;
else if (a[2]) y = 4'b0100;
else if (a[1]) y = 4'b0010;
else if (a[0]) y = 4'b0001;
endmodule
```

Each else if needs to be checked as well. That means that the need to all be if statements instead of else if statements.

if (a[3]) y=4'b1000; else begin
if (a[2]) y=4'b0100; else begin
if (a[1]) y=4'b0010; else begin
if (a[0]) y=4'b0001; else y=4'b0000;

5. And gate with three inputs

```
module and3(input      a, b, c,
output reg y);
reg tmp;
always @ (a, b, c)
begin
tmp <= a & b;
y   <= tmp & c;
end
endmodule
```

tmp and y need to use blocking assignments instead of non blocking ones.

tmp = a&b;
y = tmp&c;

6. Register

```
module floprsen(input              clk,
input              reset,
input              set,
input      [3:0] d,
output reg [3:0] q);

always @(posedge clk, posedge reset)
if (reset)     q <= 0;
else           q <= d;

always @ (set)
if(set) q<=1;
endmodule
```

the last always statement is unnecessary. We can combine the last 2 always statements together.

```
always @(posedge clk, posedge reset, posedge set)
if (reset) q<=0;
  else begin
  if (set) q<=1;
  else q<=d;
```

4 Lab overview

In this Lab, you will use the Verilog hardware description language to implement and verify a 32-bit sign extender module, a register file and the ALU module.

Note: the modules that you implement in this lab and several labs following will be used as building blocks for a complete single cycle processor. It is highly recommended that you implement and test your models properly.

5 32-bit sign extender

A 32-bit extender takes as input a 16-bit immediate value, and based on the control bit, the value is either zero extended or sign extended to a 32-bit value. If the control bit is 1, then the value is zero extended; otherwise, it is sign extended. Type the following Verilog code into a new file and name it "SignExtender.v".

```
module SignExtender(BusImm, Imm16, Ctrl);
output [31:0] BusImm;
input [15:0] Imm16;
input Ctrl;

wire extBit;
assign #1 extBit = (Ctrl ? 1'b0 : Imm16[0]);
assign BusImm = {{16{extBit}}, Imm16};

endmodule
```

Implement a testbench for the module, SignExtender, and simulate the testbench with VCS. Your testbench should be *self-checking*, i.e. it should not only set inputs but it should also check for expected outputs and print a failure message in the event of unexpected output. You may use the testbenches provided in Lab06 as an example of an exhaustive, self-checking testbench.

The module above contains an error, and you are required to find it using you testbench module and correct it. Run the simulation again to ensure the SignExtender module is working properly. At this point, demonstrate your progress to the TA.

5.1 Deliverables

Completion of this part of the lab requires the following deliverables to be turned in to eCampus along with your completed lab form:

- Verilog code for your corrected 32-bit sign extender.
- Verilog code for your testbench which tests all corner cases of the sign extender and catches the error in the provided code.
- A screen shot of the sign extender's input and output waveforms from the VCS DVE with the testbench as stimulus.

6 Register File

Implement a 32x32 register file. You may use behavioral style Verilog programming, but your code must be synthesizable (use only synthesizable constructs). The register file contains 32, 32-bit registers. The file `MiniRegisterFile.v` contains a small register file which is missing several features, but can provide a starting point.

Below is a specification of the register file;

- Buses A, B, and W are 32 bit wide.
- When RegWr is set to 1, then the data on Bus W is stored in the register specified by Rw, at negative clock edge.
- Register 0 must always read zero.
- Data from registers (as specified by Ra and Rb) is sent on Bus A and Bus B respectively, after a delay of 2.
- Writes to the register file must have a delay of 3.
- The Register File module should have the following interface:

```
module RegisterFile(BusA, BusB, BusW, RA, RB, RW, RegWr,
Clk);
```

- Test your code against the provided `RegisterFileTest.v` testbench.

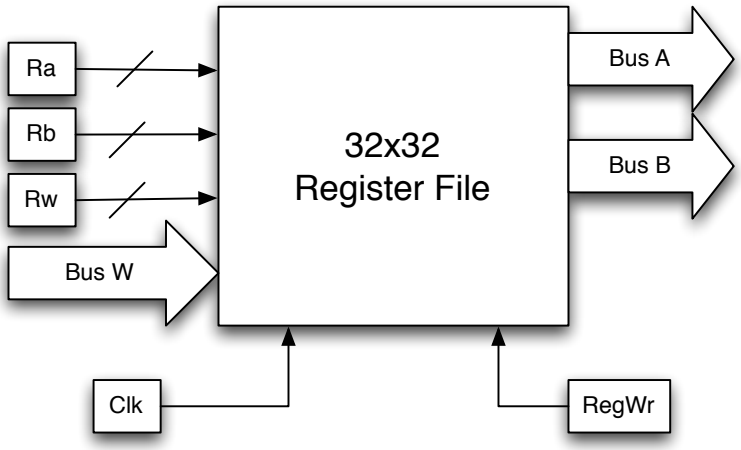

Fig. 1: Register File

Write a testbench to test your code for the following inputs of `Ra`, `Rb`, `Rw`, `BusW` and `RegWr`. Initialize the register file to the following data.

Register	**Value**	**Register**	**Value**
0	0x00000000	**8**	0x00000008
1	0x00000001	**9**	0x00000009
2	0x00000002	**10**	0x000000A
3	0x00000003	**11**	0x000000B
4	0x00000004	**12**	0x000000C
5	0x00000005	**13**	0x000000D
6	0x00000006	**14**	0x000000E
7	0x00000007	**15**	0x000000F

All the other registers should have a value of zero. Specify the values on `BusA` and `BusB` along with the registers (if any) that will be modified. Put `N/A` if none are modified:

Ra	Rb	Rw	RegWr	Bus W	Bus A	Bus B	Modified
0	1	0	0	0x00000000			
2	3	1	0	0x00001000			
4	5	0	1	0x00001000			
6	7	A	1	0x00001010			
8	9	B	1	0x00103000			
A	B	C	0	0x00000000			
C	D	D	1	0x0000ABCD			
E	F	E	0	0x09080009			

6.1 Deliverables

Completion of this part of the lab requires the following deliverables to be turned in to eCampus along with your completed lab form:

- Verilog code for your new register file.
- Verilog code for your testbench which tests the register file according to the specification above.
- A screen shot of the register file's input and output waveforms from the VCS DVE with the testbench as stimulus.

7 ALU Block

Design an ALU that provides support for implementing the following instructions: **and**, **sub**, **or**, **xor**, **slt**, **addi**, **ori**, **sll**, **slt (ignoring the overflow)**. All operations must have a delay of 20 before the output is available. Use non-blocking assignments as shown in the provided template.

Your ALU should implement the following operations (which are needed to support the instructions in the pre-lab).

Opperation	ALU Control Line
AND	0000
OR	0001
ADD	0010
SLL	0011
SRL	0100
SUB	0110
SLT	0111
ADDU	1000
SUBU	1001
XOR	1010
SLTU	1011
NOR	1100
SRA	1101
LUI	1110

Note* Several instructions have subtle requirements. The `lui` instruction supposed to take the least significant 16 bits of `BusB`, and place them in the most significant bits of `BusW`, with the lower 16 bits of `BusW` set to 0. `slt` is a signed instruction, but in performing comparisons, verilog assumes unsigned numbers. One way to work around this is to invert the most significant bit before comparisons. For example, if A and B were both 6 bit signed numbers, one can perform the comparison in the following manner:

```
if( { ~A[5], A[4:0] } < { ~B[5], B[4:0] } )
```

For `sra`, the >>> operation performs an arithmetic shift, but only if the the value is signed. Again, verilog assumes values are unsigned, so it needs to be told otherwise. The following example does an arithmetic shift on A by 5:

```
$signed(A) >>> 5;
```

You should use the provided ALU code template `ALU.v`. You will need to extend it for your own purposes. The ALU module has the following interface:

```
module ALU(BusW, Zero, BusA, BusB, ALUCtrl);
```

- Ports `BusA`, `BusB`, and `busW` are 32 bits wide.
- `ALUCtrl` is 4 bits wide, supporting up to 16 functions.
- Port `Zero` is a boolean variable that is 'true' (1) when `BusW` is 0, and 'false' (0) otherwise.

You may use behavioral style Verilog programming, but your code must be synthesizable (make sure to show your TA). Test your ALU using the test bench file ALUTest.v

from eCampus. Make sure to add the test vectors from the prelab to the test bench.

Note: You will implement the ALU control block in a future lab.

7.1 Deliverables

Completion of this part of the lab requires the following deliverables to be turned in to eCampus along with your completed lab form:

- Your Verilog code for the ALU and control block. Ensure your program is clearly commented.
- A file with waveform trace from DVE (print to file) when running the ALUTest.v testbench. Be sure to set it to require enough pages that the full signal values are readable in the wavefore trace. Postscript format is fine.