EE 361
16-Bit Single-Cycle LEGLite Project

Summary: You will implement, in verilog, a simpified version of the **single-cycle LEGv8**, which we will refer to as LEGLite.

LEGLite Description: Its data and address buses are 16-bits wide rather than 32-bits. All instructions and operands are 16-bits, i.e., "halfwords". Memory is byte addressable and is organized as Little Endian. Note that memory addresses of halfwords are divisible by two.

General Purpose Registers (8 of them)

Name	**Reg #**	**Usage**
X0-X6	0-6	General purpose registers
XZR	7	Zero register (always zero valued)

Instruction Formats

Name	Fields					Example
Format	3 bits	3 bits	4 bits	3 bits	3 bits	
R	op	Xm	NA	Xn	Xd	ADD Xd,Xn,Xm
I	op	constant		Xn	Xd	ADDI Xd,Xn,#constant
D	op	constant		Xn	Xd	STUR Xd,[Xn,#constant]
CB	op	constant		NA	Xd	CBZ Xd,#Label

Note: Instruction formats I, D, and CB are all the same, so we could just call them a single format. But we refer to them as different formats to be consistent with LEGv8.

Machine Instructions: In the table LD and ST are load and store, respectively, a 16-bit halfword.

Name	**Format**	**Example**					**Comments**
		3 bits	**3 bits**	**4 bits**	**3 bits**	**3 bits**	
ADD	**R**	**0**	**3**	**0**	**2**	**1**	**ADD X1,X2,X3**
SUB	**R**	**1**	**3**	**0**	**2**	**1**	**SUB X1,X2,X3**
LD	**D**	**3**	**50**		**2**	**1**	**LD X1,[X2,#50]**
ST	**D**	4	**50**		**2**	**1**	**ST X1,[X2,#50]**
CBZ	**CB**	**5**	**(PC-relative offset)/2**		**0**	**1**	**CBZ X1,Label**
ADDI	**I**	**6**	**50**		**2**	**1**	**ADDI X1,X2,#50**
ANDI	**I**	**7**	**50**		**2**	**1**	**AND X1,X2,#50**

The circuit diagram for the single cycle LEGLite is similar to the following circuit diagram for the simplified LEGv8, that was presented in lecture. However, there are differences in the bus widths and shifting. Also, there is no unconditional branch instruction. Also note that this computer must have a reset input that will reset the computer when it is asserted on the next clock transition. Basically, the reset will clear the program counter (PC) to 0.

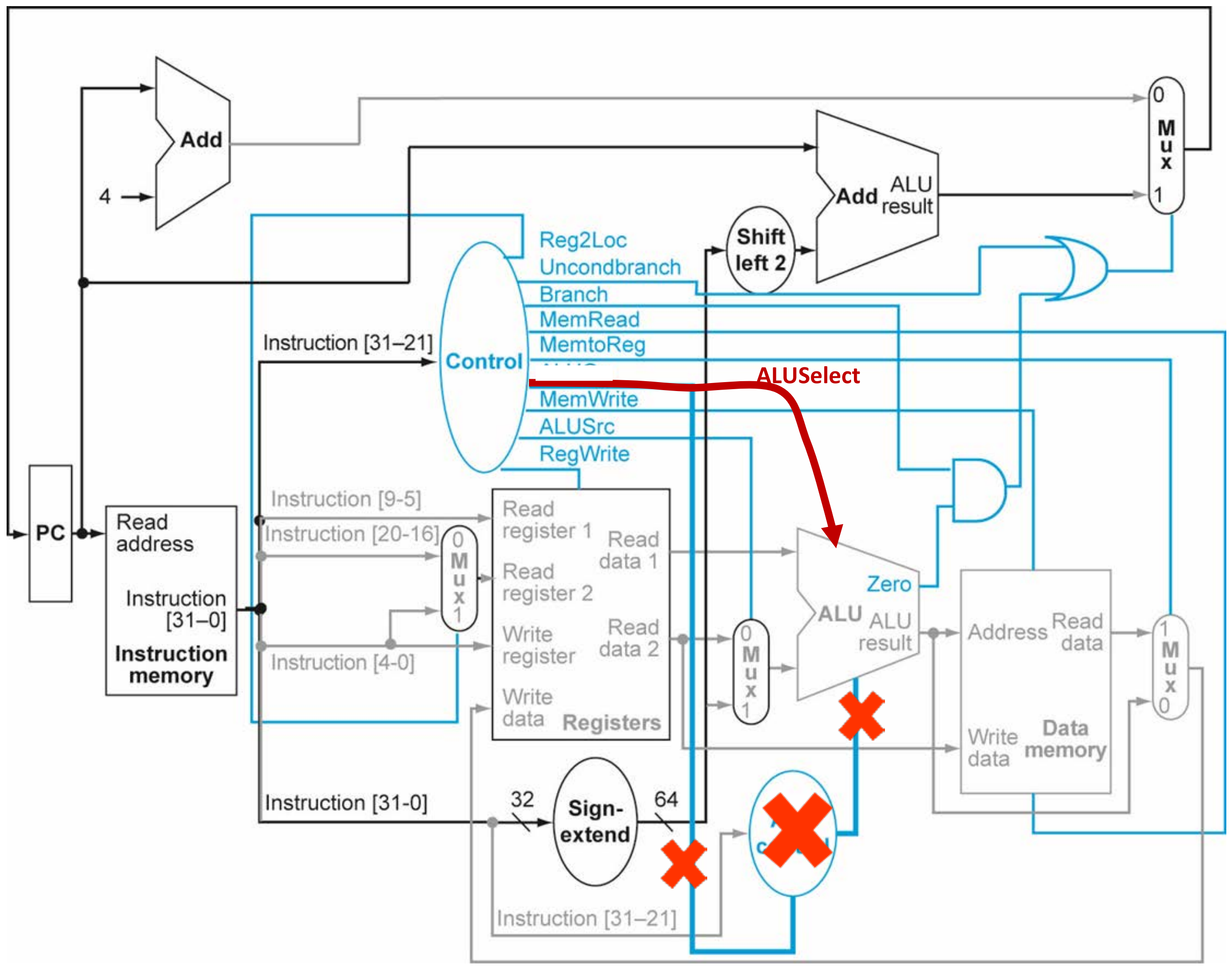

Figure 1. Simplified LEGv8 from lecture notes.

The components of the circuit are

Instruction Memory that has the program	2:1 Multiplexers
Register File	Shifters
Data Memory and I/O	Sign Extenders
ALU	Controller
Adders	Program Counter (PC) and its control logic

Figure 2 explains the PC and its control logic. Circled in the figure is the PC Control Logic.

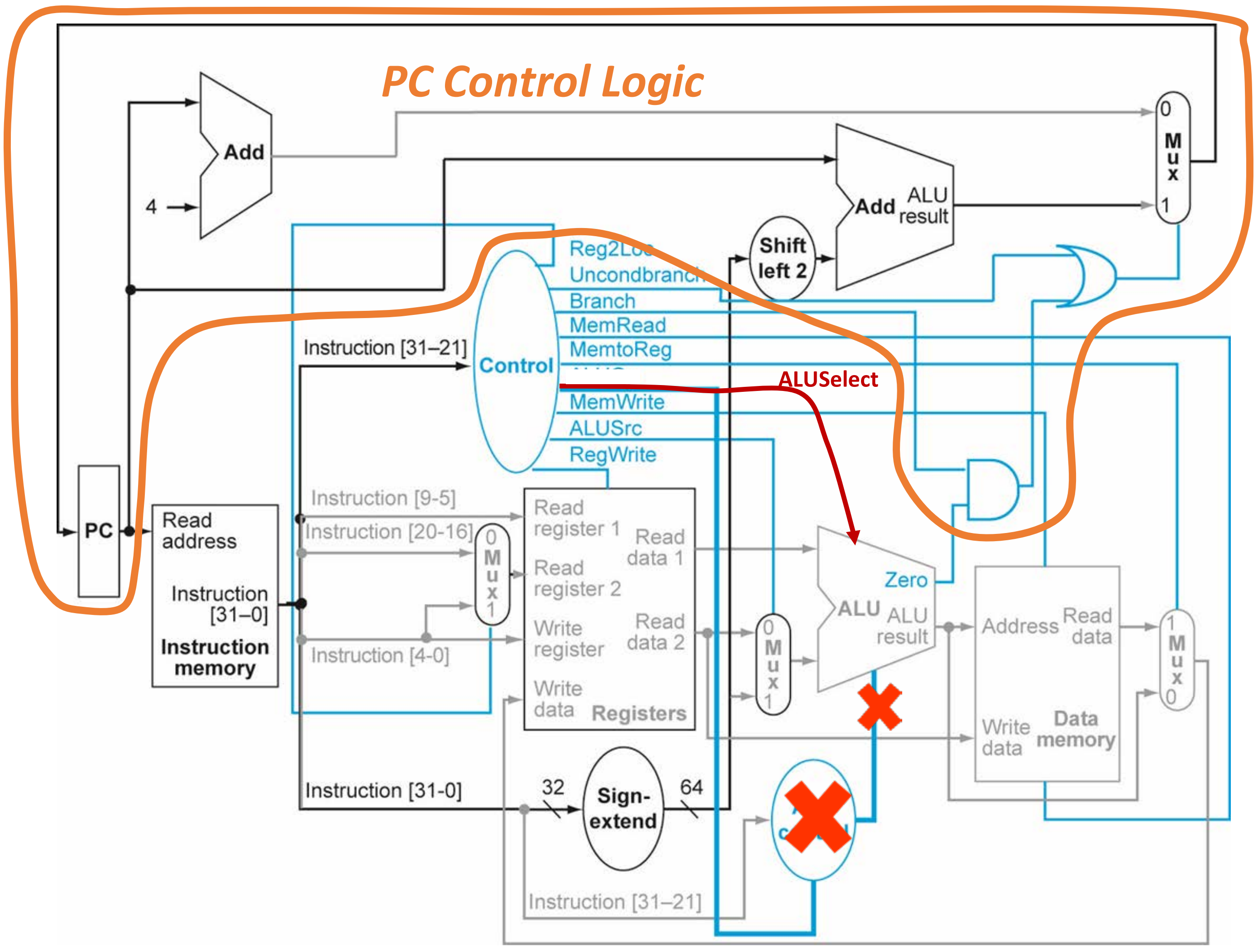

Figure 2. Schematic showing the PC control logic.

Figure 3 is a block diagram of the PC control circuit. You can implement this as a single verilog module. This simplifies the design since you have less components. Note that figure does not show the clock input.

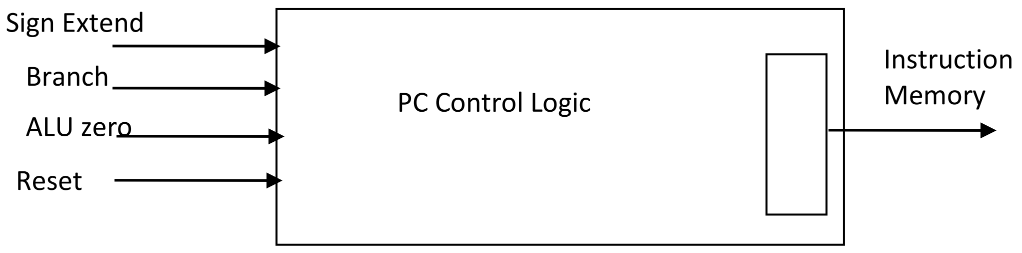

Figure 3. Block diagram of the PC control logic.

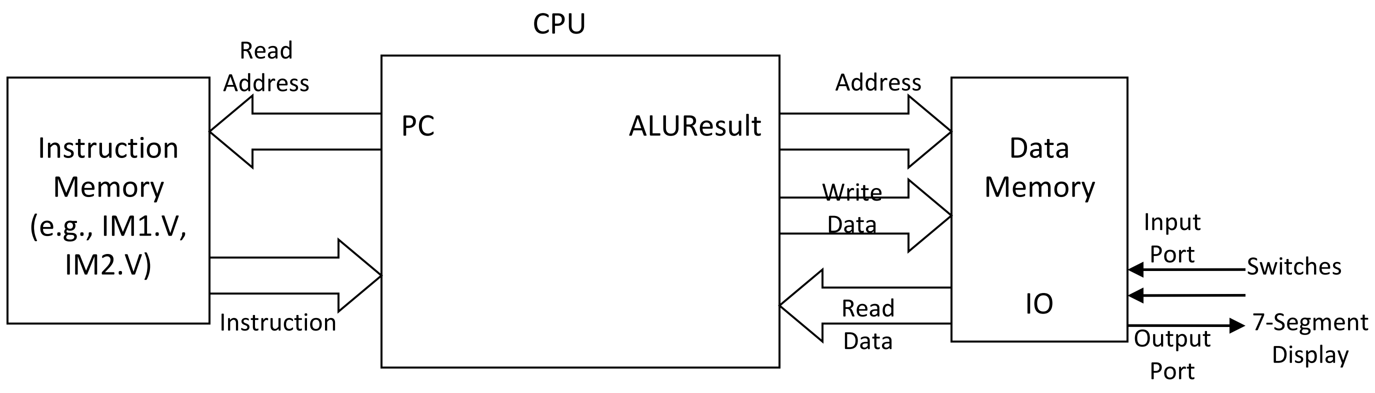

Figure 4. Block diagram of the computer, clock not shown.

Figure 4 shows a block diagram of the computer. There is the CPU, instruction memory, and data memory, which includes I/O. Note that the CPU in Figure 4 is basically all the components in Figure 2 except the instruction memory and data memory.

Notice that the Data Memory/IO block has

- Input port at address 0xfff0. It is connected to sliding switches sw0 and sw1: sw0 is bit 0 and sw1 is bit 1
- Output port at address 0xfffa. It is connected to a 7 segment display.

The Data Memory is 128 memory cells, each with 16-bits. The addresses of the memory cells are 0, 2, 4, ..., 254.

The Data Memory/IO Unit (DMemory_IO module) works as follows. If the address is between 0 and 254 then it works like ordinary memory. If the address is equal to 0xfff0 then then it outputs the values of the two switches in the last two bits of data. If the address is equal to 0xfffa then it will store the data written to it and output it to the 7-segment display output port.

The following is the portion of the module that controls reading data from the unit.

```
assign mem_rdata = memcell[addr[7:1]];  // data from memory
assign io_rdata = {14'd0,io_sw1,io_sw0}; // data from switches

always @(addr or mem_rdata or io_rdata or read)
      begin
      if (read == 0) rdata = 0;
      else // read = 1
            begin
            if (addr >= 0 && addr < 256)     rdata = mem_rdata;
            else if (addr == 16'hfff0)       rdata = io_rdata;
            else rdata = 0; // default
            end
      end
```

The following is the portion of the module that controls writing data to the unit.

```
always @(posedge clock)
      if (write == 1 && addr == 16'hfffa)
          io_display <= wdata[6:0];

always @(posedge clock)
      if (write == 1 && addr[15:8] == 0) memcell[addr[7:1]] <= wdata;
```

Assignment:
There are three stages to this homework. Each stage takes about a week. DON'T FALL BEHIND.

- Stage 1: This stage is to test components of the computer. There's nothing to submit. The files for this stage are in the folder HwLEGLite-Stage1. This stage is straight forward but it will take time. Do this right now.
- Stage 2: You will implement a computer so that it runs a simple program, which multiplies two integers by using a loop. The computer should be able to run the instructions: ADD, ADDI, and CBZ. The files for this stage are in the folder HwLEGLite-Stage2
- Stage 3: This stage improves the computer from stage 2. In particular, it can also run LD, ST, and ANDI. The program that it will run will access the IO ports. The files for this stage are in the folder HwLEGLite-Stage3

For each of these stages are verilog files for component and testbenches. Read the testbenches to understand them. They have comments to guide you.

Grading: The Homework is worth 30 points according to the following

Complete	**Points**
Partially working Stage 2	15
Completely working Stage 2	24
Completely working Stage 3	30

Submission Instructions:

Submit a single Windows folder (zipped), which should have

1. All your verilog files including the testbench
2. Also include either
    a. IM2.V if you have Stage 3 completely working or
    b. IM1.V if don't have State 3 working but you have Stage 2 partially or completely working
3. Project file that will run the testbench
4. README file which includes
    a. Your name
    b. Description of what you have completed (and not completed). For example
        i. "I have completed Stage 3. I have included IM2.V"
        ii. "I have completed Stage 2, I didn't get Stage 3 to work completely. I have included IM1.V"
        iii. "I didn't get Stage 2 to work completely but I have it partially working. I have included IM1.V"
        iv. "I didn't get anything to work in Stage 2"
    c. Description of what you have in this folder including a list of all the files including all verilog and project files.

Zip the folder and submit it through laulima as an attachment on the due date indicated in the assignment.

Description of 3 Stages:

Stage 1: Here are the files in HwLEGLite-Stage1 folder (note some files may need some editing)

- Parts.V: This has
    - Register file
    - ALU
    - 16-bit 2:1 multiplexer
    - 16-bit 4:1 multiplexer
    - Data memory
        - The data memory has 128 memory cells that are 16-bits
        - Memory starts from address 0.
        - Memory is byte addressable, each 16-bit word has an address that is divisible by 2
        - The word addresses are 0, 2, 4, ...
        - There are two IO ports
            - Output port to a 7-segment display at address 0xfffa.
            - Input port from two switches at address 0xfff0.
    - There are three testbenches for this file
        - testbench-Parts-CombCirc.V: Tests the combinational circuits.
        - testbench-Parts-Rfile.V: Tests the register file
        - testbench-Parts-Dmem.V: Tests the data memory and IO

Create three projects, each having one of the testbenches and Parts.V. Run each project and make sure Parts.V works. Read the testbenches to understand them. There are comments to guide you. You can also write your own testbenches or modify the existing ones.

Be sure to look over the verilog files to understand the circuits, so read the comments.

There is nothing to turn in for this stage.

The next two stages are to build a working processor that can run some of the instructions.

Stage 2: In the Hw9-Stage 2 folder there are the following files (some files may need editing, i.e., buggy)

- LEGLiteSingle.V: This has the LEGLite processer module. It's incomplete (actually it's basically empty) and you must complete it.
    - It must implement the ADD, ADDI, and CBZ instructions
- testbench-LEGLiteSingle-Stage2.V: This is the test bench for LEGLiteSingle.V. It has instantiations of the
    - LEGLiteSingle module
    - Data memory (with IO)
    - Instruction memory IM1.V. This is a program that multiplies 3 by 5 using a loop. It uses the ADD, ADDI, and CBZ, instructions. The following is the program

```
L0:     ADDI  X2,XZR,#3         # X2 = 3, X2 is used as a counter
        ADD   X4,XZR,XZR        # Clear X4, which will contain the final product
L1:     CBZ   X2,L0       # If counter = 0 then we've completed the multiply and we can start all over
        ADDI  X4,X4,#5  # Increment X4 by 5
        ADDI  X2,X2,#-1 # Decrement counter
        CBZ   XZR,L1      # Continue looping
```

- Note: We can check if the LEGLite is working correctly by viewing the ALU output. For example, if the instruction is ADDI X4,XZR,#3 then the ALU output = 3. Then if the next instruction is ADDI X4,X4,#3, the ALU output is 6.

  Note that the ALU output is the write data output of the LEGLite which goes to data memory. So by viewing the write data output from the LEGLite we will view the ALU output.

- To create a project you need the following files:
    - IM1.V
    - LEGLiteSingle.V
    - LEGLite-Control.V (described below)
    - LEGLite-PC.V (described below)
    - Parts.V
    - testbench-LEGLiteSingle-Stage2.V.

- LEGLite-Control.V: This has the Controller module of LEGLite. This will be instantiated in the LEGLite module.
    - LEGLite-Control.V: This has an incomplete Controller module. Shown below is the function tables for the ALU for your reference. This is straight forward if you understand how the LEGLite datapath works.
    - There's a sample testbench testbenchControl (it may be buggy). You can make your own testbench too.

ALU function table

alu select	**Operation**
0	add
1	subtract
2	pass input 1 to output ALUresult
3	or
4	and

- LEGLite-PC.V: This has the PC control logic described earlier (see Figures 3 and 4). This is a component of MIPS-L and will be instantiated in the LEGLite module.
    - LEGLite-PC.V: has an incomplete PCLogic module. Currently, it only increments PC by 2 and resets to 0. You should also have it operate for conditional branch CBZ. This is straight forward (really).
    - testbench-PC.V: this is a test bench for PCLogic module, which is the module that does PC Control.

Suggestion 1: LEGLite may be difficult to debug, especially since the circuit is not trivial and you are still learning about verilog and the Xilinx Webpack tools. In addition, the LEGLite is a circuit that forms a "loop" of circuitry. Then if there is a bug, the error can circulate throughout the circuit, which makes it difficult to locate the origin of the bug.

It's helpful to build the LEGLiteSingle in stages. For example, you can start with testing the PC control logic. Next, you can add the the Instruction memory. Then set "default" inputs to specific values (e.g., zero), and subsequently run this incomplete system to check if it works. Then you can add other components such as the sign extender, register file, and ALU.

Suggestion 2: To debug a module, it is often very helpful to display signals. However, when simulating a module, the signals are displayed only at the testbench module rather than the particular circuit module you are debugging. To access signals that are completely internal to the circuit module, you can do the following.

Suppose you have a circuit module called X, and you'd like to probe its internal signals, signal1 and signal2. You can modify it for debugging purposes as follows. Add to module X an extra output called "probe". This will be used to output 6 bits of information out of the module, which is 4 bits of "signal1" and 2 bits of "signal2".

```
module X(y2, y1, reset, clock, x1, x2, probe)
{
...
output  [5:0] probe;
...
assign probe[5:2] = signal1;
assign probe[1:0] = signal2;

}
```

Now the testbench module that instantiated X (e.g., the testbench) can use probe to display the two signals.

Stage 3: Implement LEGLiteSingle so that it can run LD and SW instructions as well as the instructions of Stage 2. It should run Instruction Memory IM2.V. The files are in the folder LEGLite-Stage3. It also includes the testbench.

The IM2.V program interacts with the I/O ports:

- Input switch 0 at address 0xfff0
- Output port connected to the seven segment display at address 0xfffa

It will continually check switch 0. If the switch = 0 then it outputs "0" to the seven segment display. If the switch = 1 then it outputs "1" to the seven segment display.

Description of IM2.V

The program is an infinite loop. It first does the following

- Set X3 to 0xfff0. This is a reference point to the I/O ports. For example, note that
    - LD X4,[X3,#0] will load a value from the input switches into X4
    - ST X4,[X3,#0] will store the value in X4 into the 7 segment display

Next it checks the input switch 0. If it's 0 then the computer will display "0" on the 7 segment display. Otherwise, it displays "1" on the display.

```
#  Initialization Phase
L0:     ADDI   X3,XZR,#0xfff0       # X3=0xfff0, the addr to sw0/display
        LD     X5,[X3,#0]           # X5 = input switch value
        ANDI   X5,X5,#1             # Mask all bits except bit 0 (sw0)
        CBZ    X5,Disp0             # if bit = 0 then X4 = pattern "0" else X4 = pattern "1"
        ADDI   X4,XZR,0110000       # X4 = bit pattern "1"
        CBZ    XZR,Skip
Disp0:  ADDI   X4,XZR,1111110       # X4 = bit pattern "0"
Skip:
        ST     X4,[X3,#10]          # 7-segment display = bit pattern
        CBZ    XZR,L0               # Repeat loop
```