CS223 Laboratory Project
Single-cycle processor

Groups: Each student will do the project individually. **Group size = 1**

Important dates:
Report submission and project upload: 21.12.2020 Monday, until **09:00 AM**.
You need to upload your design files in order to make your project demonstration.
Demo presentation day: 21, 22, 25, 26 and 27 December 2020 during lab time.

Project summary:

The purpose of this project is to build a simplified **single-cycle processor** (Single-cycle processor: an instruction is fetched from memory, it is executed, and the results are stored all in a single clock cycle.) on Basys 3 board that would be capable of performing several types of instructions. The processor you will design will perform several functions depending on the 16-bit instructions on 8-bit data. Your design should include instruction memory, data memory, a register file and an arithmetic logic unit (ALU). The instruction memory should be a read-only memory which would have the instructions of a sorting program that would sort the values in an array. Data memory should be able to hold 16x8-bit data. Instructions to be executed should be either picked from the instruction memory, or from the switches on Basys3. Depending on the instruction, processor should do addition, comparison, load/store, swap and branch operations.

Instructions:

The processor will work on 16-bit instructions where the most significant 3-bits will be used as the opcode (abbreviated from operation code, also known as instruction code is the portion of a machine language instruction that specifies the operation to be performed.). Depending on the opcode, the controller will decide what to do, how to interpret the rest of the instruction, and send the necessary control signals back to datapath.

Opcode	Function
000	Store Value
001	Load Value
010	Addition
011	Reserved
100	Reserved
101	Branch if equals

110	Reserved
111	Reserved

Table 1: Instruction Set

Store Value:

If the opcode is 1'b000, the next bit will decide if the value to store should come immediately taken from instruction or from a register in the register file. If the value is 1, the next 4-bits will be used as the write address for data memory, and the remaining least significant 8 bits will be used as the data value to be written. For example, instruction 000_1_0101_00000011 can be translated as "write value 3 to address 5 of the data memory," and therefore, address 5 of the data memory must be 5 after this instruction is executed.

[15:13] **Opcode**=000	[12] **Immediate bit**=1	[11:8] **Write Address**	[7:0] **Write Data (unsigned)**

Table 2: Store Instruction Immediate

Else if the immediate bit value is 0, next 4-bits will be ignored, and the remaining 8-bits will be used as write address of memory and read address from register file:

[15:13] **Opcode**=000	[12] **Immediate bit**=0	[11:8] **Ignored (don't care)**	[7:4] **Write Address**	[3:0] **Read Address**

Table 3: Store Instruction

Ex: An instruction 000_0_XXXX_0010_0011 will put the value hold by register 3 to address 2 of the data memory.

Load Value:

Load value will work similar to store value but in reverse direction. If the immediate bit after the opcode is 0, it will read the value from read address of the data memory, and put it into the register in the register file whose address is given. If the immediate bit is 1, it will load directly the value in instruction's least significant 8-bit to the given register.

[15:13] **Opcode**=001	[12] **Immediate bit**=1	[11:8] **Write Register Address**	[7:0] **Write Data (unsigned)**	
[15:13] **Opcode**=001	[12] **Immediate bit**=0	[11:8] **Ignored (don't care)**	[7:4] **Write Register Address**	[3:0] **Read Data Address**

Table 4: Load Instruction

Ex: 001_0_XXXX_0010_0011 will mean that write the value of address 3 in data memory into 2nd register in register file.

Addition:

The format for addition is as follows:

[15:13] **Opcode**=010	[12] **Ignored (don't care)**	[11:8] **Write Register Address**	[7:4] **Read Register Addr 1**	[3:0] **Read Register Addr 2**

Table 5: Addition Instruction

Ex 1: Instruction: 010_X_0000_0000_0001 will add the values in Register 0 and Register 1 and put the result in Register 0, similar to "rf[0] += rf[1];" in a high level language.

Ex 2: Instruction: 010_X_0000_0001_0002 will add the values in Register 0 and Register 1 and put the result in Register 0, similar to "rf[0] = rf[1] + rf[2];" in a high level language.

Branch if equals:

The format for branch if equals is as follows:

[15:13] **Opcode=**101	[12:8] **Jump Addr Instruction Mem**	[7:4] **Read Reg Addr 1**	[3:0] **Read Reg Addr 2**

Table 6: Branch if Instruction

This instruction tells the processor to jump to a certain instruction if the 2 given values are equal. For the previous instructions, after the processor executes them, the next instruction to fetch for processor will be the instruction that just comes after in the instruction memory. However for branch if equals instruction, if the values whose addresses are given in "Read Reg Addr 1" and "Read Reg Addr 2" are equal, the next instruction to fetch will be the instruction whose address is given in "Jump Addr Instruction Mem," else, the processor will continue to execute the next instruction in order.

Ex: Instruction: 101_00010_0000_0001

- if (rf[0] == rf[1])
  goto Jump Addr Instruction Mem
  else
  continue with next instruction

Stop Execution (Wait)

[15:13] **Opcode**=111	[12:0] **Ignored (don't care)**

Table 7: Halt Instruction

This instruction will stop the whole execution and processor will halt at this point.

Sample code to do multiplication assuming data is in main memory.

IM[0]	Load value from DM[0] to RF[0]	001_0_0000_0000_0000
IM[1]	Load value from DM[1] to RF[1]	001_0_0000_0001_0001
IM[2]	Load 0 immediately to RF[2]	001_1_0010_00000000
IM[3]	Load 0 immediately to RF[3]	001_1_0011_00000000
IM[4]	Load 1 immediately to RF[4]	001_1_0100_00000001
IM[5]	Branch to IM[9] if RF[2] == RF[1]	101_01001_0010_0001
IM[6]	Add RF[3] += RF[0]	010_0_0011_0011_0000
IM[7]	Add RF[2] += RF[4]	010_0_0010_0010_0100
IM[8]	Branch to IM[5] if RF[0] == RF[0]	101_00101_0000_0000
IM[9]	Store RF[3] to DM[3]	000_0_0000_0011_0011

Table 8: Multiplication sample code

Instruction Memory, Data Memory and the Register File:

The instruction memory should be able to hold 32 instructions and will be read-only memory. Therefore the program you write will be hardcoded in the instruction memory. Both data memory and the register file will hold 16x8 bit data, and they can have at most 2 read ports and a single write port. Notice that all the computational instructions except swap operation manipulate data on register file, so you need to get the data you want to process into registers first. The register file will take the next instruction as an input from the instruction memory (or directly from switches if middle pushbutton is pressed), and place the components of the instruction (such as opcode or DataAddress) to corresponding registers. The opcode will be sent to controller module as an output so that controller will be able to send the necessary control signals such as writeEnable, ALU opcode, jumpEnable. Both memory and register file contents will be zero on reset state, and contents will be stored with load and store instructions.

There are two ways to update memory in data memory. One is through store instruction, and the other is through swap instruction.

For the register file, updates can occur through load instruction, but also through addition and comparison instructions, as the result of these operations are also written into specified registers.

Interface:

Push buttons, switches and seven-segment display will be used to interact with the processor. Seven-segment display will be used to show the address and the data in memory with a separator in middle (Figure 1). Right and left pushbuttons will be used to circulate in data memory. Assume initially seven-segment displays address 0 in the first digit and the correlated value in hex in the last 2 digits, pressing right button should display address 1 and the correlated value, whereas pressing left button should display address 0xF and the correlated data. The upper pushbutton will be used for reset. Middle pushbutton will be used to execute the instruction entered on switches whereas the down pushbutton will be used to execute the next instruction on instruction memory. The instruction to be executed next should be displayed in LEDs.

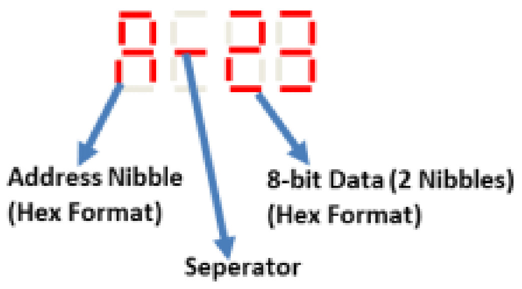

Figure 1: Memory Display

Report:

Write a report that will contain the following:

1) Block diagram of your controller/datapath and explanation.

2) Detailed state diagram of the controller and explanation.

3) You are required to write SystemVerilog code for:

- The implementation of the single-cycle processor instruction set given in Table 1.
- Push button, switch and seven-segment display interfaces in order to interact with the processor.
- For the following code, workout the code in single-cycle processor instruction set and show that it works in your processor (don't care overflows and operate with only unsigned data):

$$rf[15] = rf[rf[0]\&0x0E] * rf[rf[1]\&0x0E];$$