CS223 Project Assignment
Simple Processor
Submission Deadline: 27th December 2021 – 08:30 am

For the course project, you are going to implement a simple programmable processor in SystemVerilog. The processor will have a simple architecture but demonstrate your knowledge of designing datapath and controller of a processor. The processor here will only support five instructions which are Load, Store, Add constant, Subtract and Jump.

Overall Architecture

Figure1 illustrates an example of a general-purpose processor. In the control unit, you will have a program counter (**PC**) register to keep track of the next instruction that is going to be executed. Instruction register (**IR**) will fetch that "next" instruction from the instruction memory. The controller FSM will decode the instruction in the IR and send the control signals to the datapath accordingly. There are two additional memory units to register files here, data memory and instruction memory. Data memory is to provide additional space for data since register files offer very limited space, and instruction memory is where the program (instructions) to be run is stored.

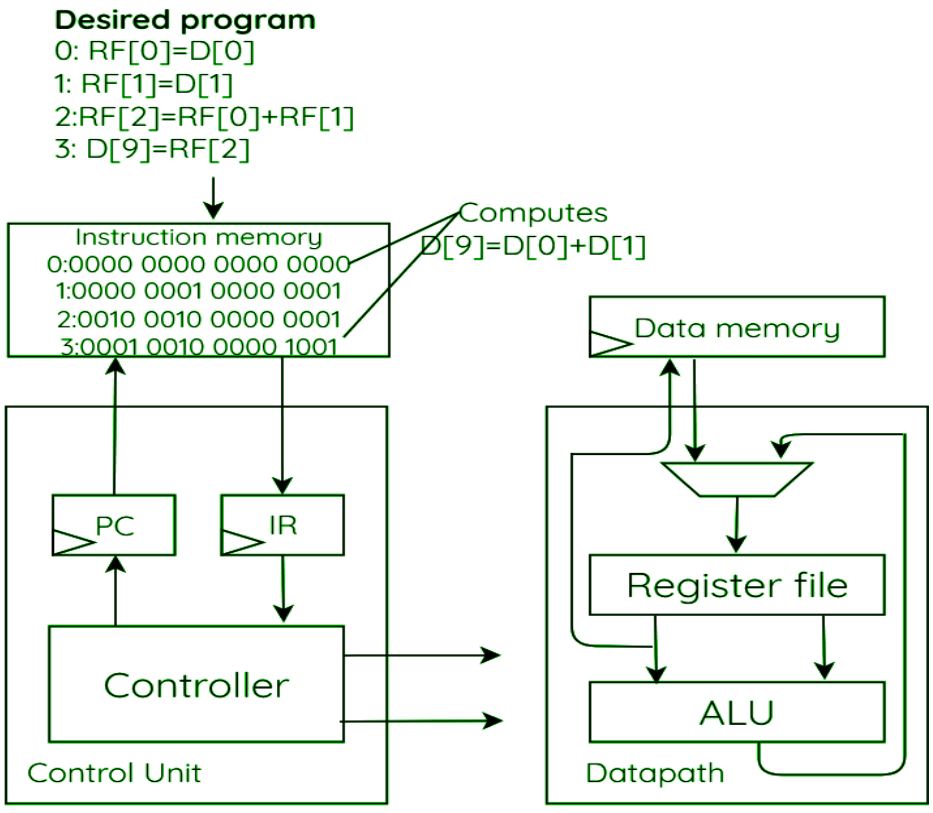

Figure1. Example processor design

Building blocks

1. **PC**: A register that holds the address of the next instruction to be executed.
2. **Instruction memory**: An array of registers that stores the instructions that will be executed (Instruction memory is read-only means that you will hard code your instruction inside of it and its content doesn't change during the execution of the instructions)
3. **Register file**: An array of registers that keeps data.
4. **ALU**: a module that performs the arithmetic operations
5. **Data memory**: An array of registers that keeps data. (Additional space to store data)

Instruction Set

In the following, we define how the instructions are represented using 12-bit and what each instruction is actually supposed to do. The processor should be able to execute 4 different types of operations: **Load**, **Store**, **Sub**, **Jump** and **add constant**. The most significant 3-bits of each instruction are used to identify the operation type(load, store, sub, jump or add constant) and the rest of the bits can be interpreted differently according to the instruction.

Load –000 xx r2r1r0 d3d2d1d0: This instruction specifies a move of data from the data memory location (D) whose address is specified by bits [d3d2d1d0] into the register file (RF) whose address location is specified by the bits [r2r1r0]. For example, the instruction "000 00 001 0010" specifies a move of data memory location 0010, D[2], into register file location 001 (or RF[1]) – In other words, that instruction represents the operation RF[1] = D[2]. Notice that in load instructions, there are 2 redundant "don't care" bits, so instructions "000 00 001 0010" and "000 11 001 0010" should both do the same thing as they only differ in those bits.

Store –001 xx r2r1r0 d3d2d1d0: This instruction specifies a move of data in the opposite direction as the instruction load, meaning a move of data from the register file to the data memory. So, "001 00 001 0100" specify D[4] = RF[1]. Similar to the load instruction, the 3th and 4th most significant bits are redundant.

Sub– 010 wa2wa1wa0 rb2rb1rb0 ra2ra1ra0: This instruction specifies a subtraction of two register-file specified by [rb2rb1rb0] and [ra2ra1ra0], with the result stored in the register file specified by [wa2wa1wa0]. For example, "010 010 000 001" specifies the instruction RF[2] = RF[0] - RF[1]. Note sub is an ALU operation.

Add constant – 101 wa2wa1wa0 rb2rb1rb0 ca2ca1ca0 : This instruction specifies a sum operation of one register-file specified by [rb2rb1rb0] and constant value ca2ca1ca0, with the result stored in the register file specified by [wa2wa1wa0]. For example, "101 010 000 001" specifies the instruction RF[2] = RF[1] + 001. Note that this is also again an ALU operation.

Jump - 111 ca2ca1ca0 rb2rb1rb0 ra2ra1ra0 : This instruction is a simple representation of conditional statement in C programming language. It compares the value of the registers in the register-file specified by [rb2rb1rb0] and [ra2ra1ra0]. If they are **equal** the address of the next instruction will be the address of the current instruction plus (ca2ca1ca0%(instruction memory size)):

$$PC = (PC_{next} + current\ address(c_{a2}c_{a1}c_{a0}))\%8$$

Datapath:

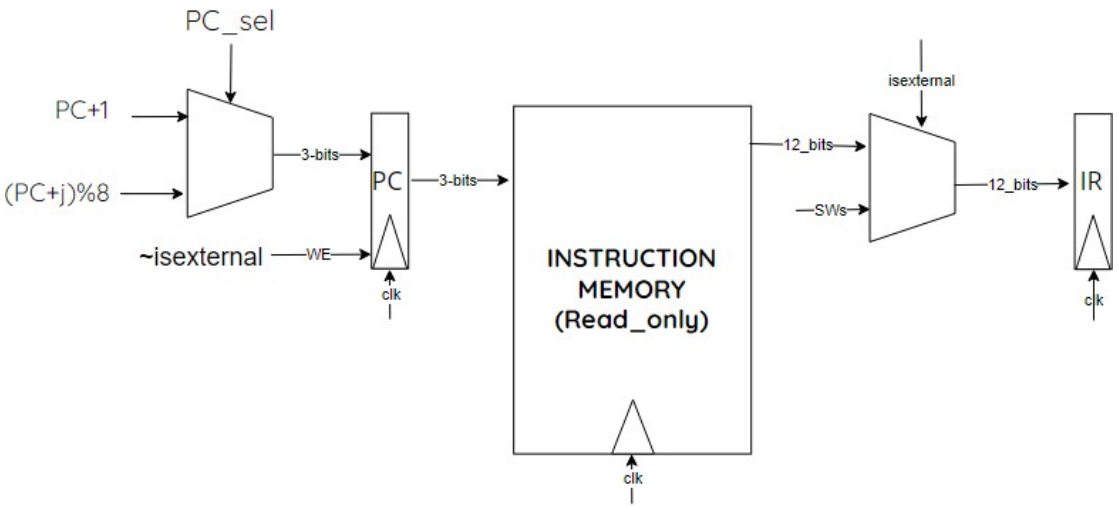

Figure 2: Datapath of fetching instructions from instruction memory to instruction register

As you can see in Figure 1, there are 2 important registers in the Controller module, which are PC and IR. PC is responsible for keeping the address of the next instruction that is going to be executed. When an instruction is executed, PC is incremented so that it will point to the next instruction in the program (Instruction memory). IR is the register where the instruction to be executed is fetched before being decoded. Instruction Memory(IM) is the memory where the program in machine code is being held. Since the instruction set consists of 12-bit instructions, IR and Instruction Memory should also hold 12-bit values. IM should have 8 slots, therefore PC should be 3 bits (to specify the address). You should write some instructions into the instruction memory yourselves to test your

processor. IM should be a read-only memory, which means that it won't have any writeAddress/writeData ports but only readAddr and readData ports. Since it is a ROM, you need to write your instructions hard coded on your code on Vivado. The processor should also be able to take instructions from switches. Therefore we have an extra input named isexternal. If isexternal is 1, 12 of the switches should be used to define an instruction and on the clock edge, IR should fetch the instruction defined by switches instead of the pointed instruction in IM. In this case, the PC shouldn't also be incremented. That is why the write-enable bit of the PC register is isexternal invert.

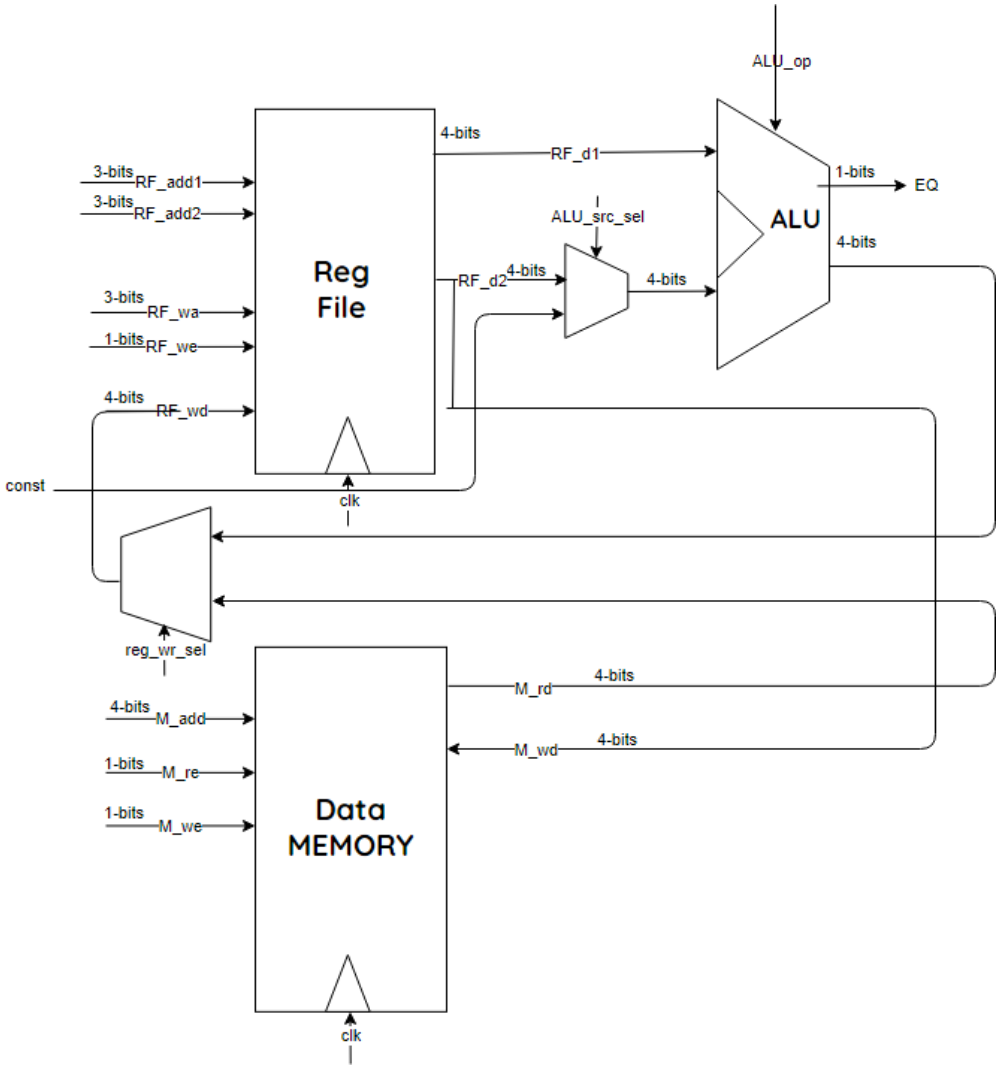

Figure 3. Datapath from decoded instruction to writeback into memory or register file Bits in the block diagram

RF_ad1: source register address 1

RF_ad2: source register address 2

RF_wa: destination register address

RF_we: Register file write enable

RF_wd: Register file write data

RF_d1: Register file source register data 1

RF_d2: Register file source register data 2

M_add: Memory address

M_re: Memory read enable

M_we: Memory write enableM_rd:

Memory read data

M_wd: memory write data

Const: Constant value coming from the instruction

EQ: If RF_d1 and RF_d2 are **equal** EQ is **1** otherwise EQ equals to **0**

After decoding the instruction the corresponding operation will be executed.

1. For **sub** operation, RF_ad1 and RF_ad2 will be read from the register file and go tothe ALU for calculation. Then the result will be written into the register file.
2. For **load** operation, data will be read from the data memory and the result will be written into the destination register in the register file.
3. For **store operation**, the value in the source register will be written in the memory cells defined in the instruction.
4. For add **constant operation**, the value in the source register will be added to the constant value defined in the instruction and the result will be written in the destination register in the register file.
5. For **Jump** operation, the RF_d1 and RF_d2 will be compared in the ALU. If they areequal, the selection of the PC will change to read PC+j where j is Ins[8:6] (**ca2ca1ca0**).

Data memory

The data memory would work just like a register file but only differ in memory size and ports. It should have 16 slots, each holding 4-bit data. Unlike the register file, it should only have 1input for

address selection, 4 bit (M_add), which will be used both for reading and writing to memory (depending on the enable signals). It has two separate enable inputs for write (M_we) and read (M_re). The write data is 4-bit data shown as M_wd. The read data is alsoa 4-bit output shown with M_rd in the figure 3. In the initial state, data memory should have values:

D[0] = 0x0, D[1] = 0x1, ... D[14] = 0xE, D[15] = 0xF.

Controller

The controller can be modeled as a state machine with states responsible for mainly fetching the instruction, decoding the instruction and executing the operation. In Fetch state, the next instruction should be written to IR register. In the next clock cycle, the instruction in IR should be decoded and next state should be determined according to the most significant 3 bits of the instruction. In other words, according to the three most significant bits of the instruction (opcode), we should move to one of the five states Load, Store or Add constant, Subtraction or Jump. And finally, after we are done with that instruction, we go back to the Fetch state, waiting for next pushbutton press to execute the next instruction. Please note that moving from one state to another should be synchronized by the clock signal. The important thing about the controller of your processor is that, you should decide what set of control signals should be enabled or disabled in either state of the controller. To make it clear for you, let's give an example. Assume after decoding the incoming instruction, you found out that the instruction is a **load** instruction. Respectively, you will set your next state as Load state. In the Load state, in order for your datapath to work properly, the controller should give the right values to the datapath. For the case of Load instruction, it should set the following signal as below:

M_add =d3d2d1d0 / M_re = 1 / M_we = 0 / RF_we = 1 / RF_wd = r2r1r0

User Interface

- Left pushbutton will be used to execute the next instruction in the instruction memory. Toavoid pressing multiple times a debouncer is needed. The processor should wait idle if no button is pressed.
- Right pushbutton will be used to execute the instruction defined by switches. It is the signal isexternal. Here also, a debouncer is needed.
- 12 rightmost switches on Basys3 will be used to provide user-defined instruction.
- SevenSegment Display will be used to show ALU inputs A,B in leftmost 2 digits and ALU result on the rightmost digit in hexadecimal. The remaining digit in between can be

turned off.

- Leftmost switch on Basys3 for reset.

Project Report

In the project report, you need to submit the following:

a) Cover Page

b) Controller High-Level State Machine Diagram

c) Controller Block Diagram

d) Controller/Datapath Top Module Block Diagram

c) Testbenchs (This is optional, mainly for partial points if Basys3 doesn't work properly)