DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2017

Lab 6: Finite State Machine Controller for the "Simple RISC Machine"

handin deadline is 7:59 PM the evening before your lab section the week of Oct 23-27

1 Introduction

In this lab you extend your datapath from Lab 5 to automate the process of executing instructions on your datapath. If you did not complete Lab 5 you can use someone else's Lab 5 solution as a starting point for this lab, but you should first test their code to make sure it works. If you use someone else's Lab 5 code as a starting point for Lab 6 you must explicitly mention this fact in your CONTRIBUTIONS file (**NOTE: for Lab 6 a CONTRIBUTIONS file is required for all submissions–even if you are working alone and even if you wrote all submitted code yourself**).

The two key additions you will make to the datapath in this lab are: One, adding a finite state machine controller to automate the process of setting control inputs to the datapath; and, two, adding an instruction register to control the finite state machine. In the lab procedure outlined in Section 3 you implement these two additions together. The rest of this introduction walks through a sequence of *preliminary* designs for extending the Lab 5 datapath. Then, Section 2 introduces the six instructions you need to implement for this lab. Section 3 specifies more precisely the changes you need to make in Lab 6. Sections 4, 5 and 6 describe the lab marking scheme, submission and demo procedures.

1.1 Controlling the datapath with a finite state machine

Recall that to execute "`ADD R2, R5, R3`" on the datapath in Lab 5 required four clock cycles. The datapath control inputs can be set easier and faster using a finite state machine instead of the slider switches. Figure 1 illustrates a preliminary state machine design for implementing "`ADD R2, R5, R3`". The inputs to the state machine are *reset*, *s* and the clock (not shown). The outputs of this state machine are all the inputs for the datapath implemented in Lab 5. To keep Figure 1 from getting cluttered we only show a subset of all outputs in each state. Any datapath input not shown as an output of a given state is 0 in that state. For example, in Figure 1 when in state *GetA* the datapath input *loadc* is assumed to be 0 even though it is not explicitly shown inside the circle representing state *GetA*. Later, in Section 3.2, you will design your own state machine. When you do, remember that in your Verilog you need to set all outputs in every state—including those outputs that are zero or for which you don't care what the value is.

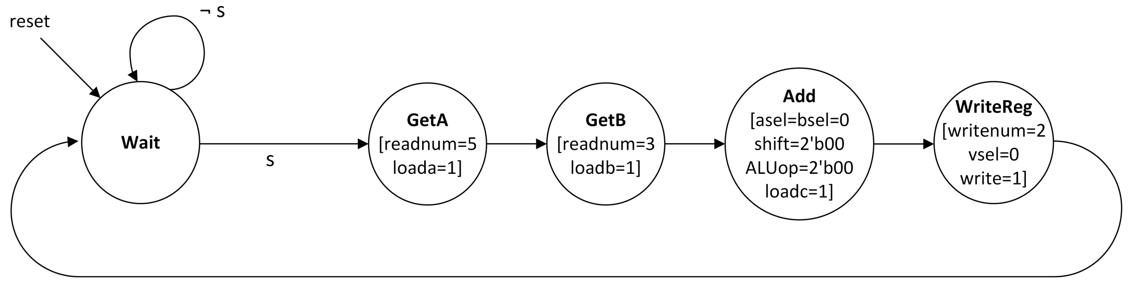
reset
¬ s
Wait
s
GetA [readnum=5 loada=1]
GetB [readnum=3 loadb=1]
Add [asel=bsel=0 shift=2'b00 ALUop=2'b00 loadc=1]
WriteReg [writenum=2 vsel=0 write=1]

Figure 1: Finite state machine for addition.

In Figure 1, after reset the state machine waits for a start signal *s* in state *Wait*. Here a value of 1 on *s* indicates we should execute the sequence of four steps that implements "`ADD R2, R5, R3`", where each step takes one clock cycle (e.g., one rising edge of *clk*). As illustrated in Figure 2 during *Cycle n*, when in state *Wait* and *s* is 1 the rising edge of *clk* causes the state to change to *GetA* which causes the control input *readnum* to be set to 5 and *loada* is set to 1. During *Cycle n+1*, after the following rising edge of the clock the contents of register R5 are copied to Register A. At the same time the state changes to *GetB*. You might be worried changing the state to *GetB* at the same time as reading the contents of R5 would cause a problem, but it does not. Why? Recall that at a rising edge of the clock a D flip-flop copies its D input to the output Q. This copying process actually takes a short amount of time. Similarly, the output of a combinational logic block does not change exactly when the input changes but instead after a very short delay. Both of these small delays are reflected in Figure 2 where you will notice *state* changes slightly after the rising edge of *clk* and the outputs *readnum*, *loada* and *loadb* change slightly after the change in *state*. Thus, after the rising edge of *clk loada* and *readnum* stay equal to 1 and 5, respectively, just long enough that R5 will be correctly copied to A *before* the change in *readnum* during *Cycle n+1* can cause a problem. You do not see these delays in ModelSim because ModelSim performs what is known as *functional simulation*. In functional simulation these delays are assumed to be non-zero but infinitesimally small. Note that in the example in Figure 2 we assume the value hexadecimal 13 was stored in register R5 at some point between *Cycle 0* and *Cycle n*.

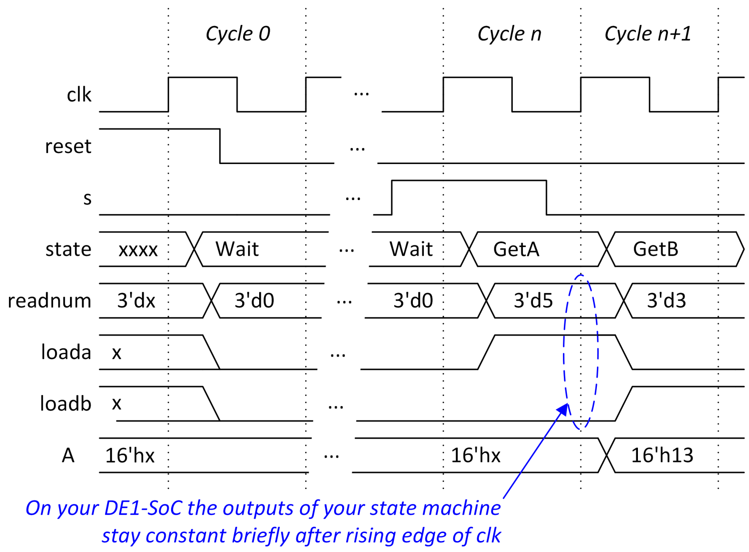

Figure 2: Register A updated on same rising edge of clk that state changes from GetA to GetB.

1.2 The instruction register: Supporting more than one instruction

In the above example the state machine was restricted to only ever executing "`ADD R2, R5, R3`", albeit as many times as we set *s* to 1. What if we want to be able to use different registers besides R2, R5, and R3 while performing addition? We can introduce some *programmability* by adding a partial *instruction register* as illustrated in Figure 3. The bottom half of this figure shows our state machine augmented with a 9-bit instruction register. This instruction register contains three 3-bit fields: `Rd`, `Rn` and `Rm`. Each 3-bit field can specifying the name of one of the eight registers inside the register file inside the datapath. The top of this figure shows a revised state machine that implements "`ADD Rd, Rn, Rm`". For example, to execute "`ADD R2, R5, R3`" we would set `Rd=3'b010`, `Rn=3'b101`, and `Rm=3'b011`. The output of the state machine have been modified: Instead of directly specifying *readnum* and *writenum* the state machine now outputs a *name select* signal called "*nsel*". The signal *nsel* is used as the select input of a three input multiplexer. This multiplexer selects which of the 3-bit values `Rd`, `Rn` or `Rm` should be driven to *readnum* and *writenum*. We can use the same multiplexer for *readnum* and *writenum* because no single state in this state machine both reads and writes the register file.

Next, suppose we wish to execute different types of instructions. For example, suppose we want to be able to execute "`MOV R3, #42`" as well as "`ADD R2, R5, R3`"? We can do this by extending the instruction

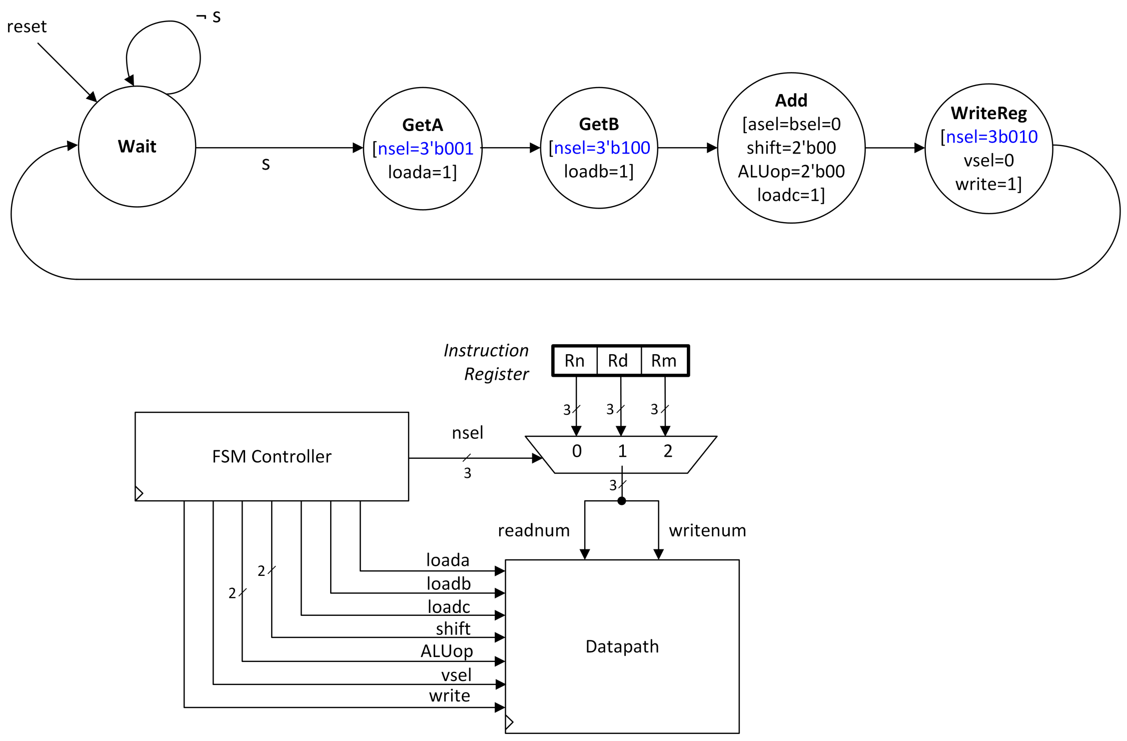

Figure 3: A partial instruction register.

register. As shown in the bottom part of Figure 4 the instruction register now includes a single-bit *opcode* field specifying the operation the instruction should perform addition (*opcode=0*) or move immediate (*opcode=1*). The instruction register now also includes some bits, called *Immediate*, that are used to specify the value to copy into the register named by `Rd` during the MOV instruction. For example, for "`MOV R3, #42`" Immediate would contain the value 42 represented as a binary number. The top part of Figure 4 shows we extended our state machine to include a state *Decode* whose role is choose between performing the sequence of four steps required for an `ADD` instruction versus the single step required for this `MOV`. To enable the correct transition out of *Decode* note that we made the value of *opcode* an input to the state machine. We also connected the 16-bit *Immediate* field of the new instruction register to the *datapath_in* portion of the datapath.

Our design can now support execution of multiple types of instruction and the registers used by an instruction can be varied after the hardware is built. At this point the instruction to execute is *encoded* with 26-bits: One bit for the opcode, 9-bits total for the three 3-bit *register specifiers* and 16-bits for the constant value. For example, the operation "`ADD R2, R5, R3`" would be encoded as:

```
0 010 101 011 0000000000000000
```

Different computer *instruction set architectures* (ISAs), such as x86 and ARM, represent a given operation, such as addition, using a different encoding (pattern of 1's and 0's). For ARM processors, each instruction is encoded in 32-bits. For x86, different instructions may be encoded using a different number of bits (between 8 and 120 bits). For Lab 6 through 8 we provide an encoding for the Simple RISC Machine ISA in which each instruction is encoded in just 16-bits. The portion of the Simple RISC Machine ISA that you will implement in Lab 6 is shown in Table 1, which is explained below. While you may want to come up with your own encoding, using this encoding will save you time later because it will enable you to use an *assembler* program we will provide you in Lab 7. Also, all students are encouraged to enter their computer designs in the Lab 8 computer design competition. All entries in that competition will be

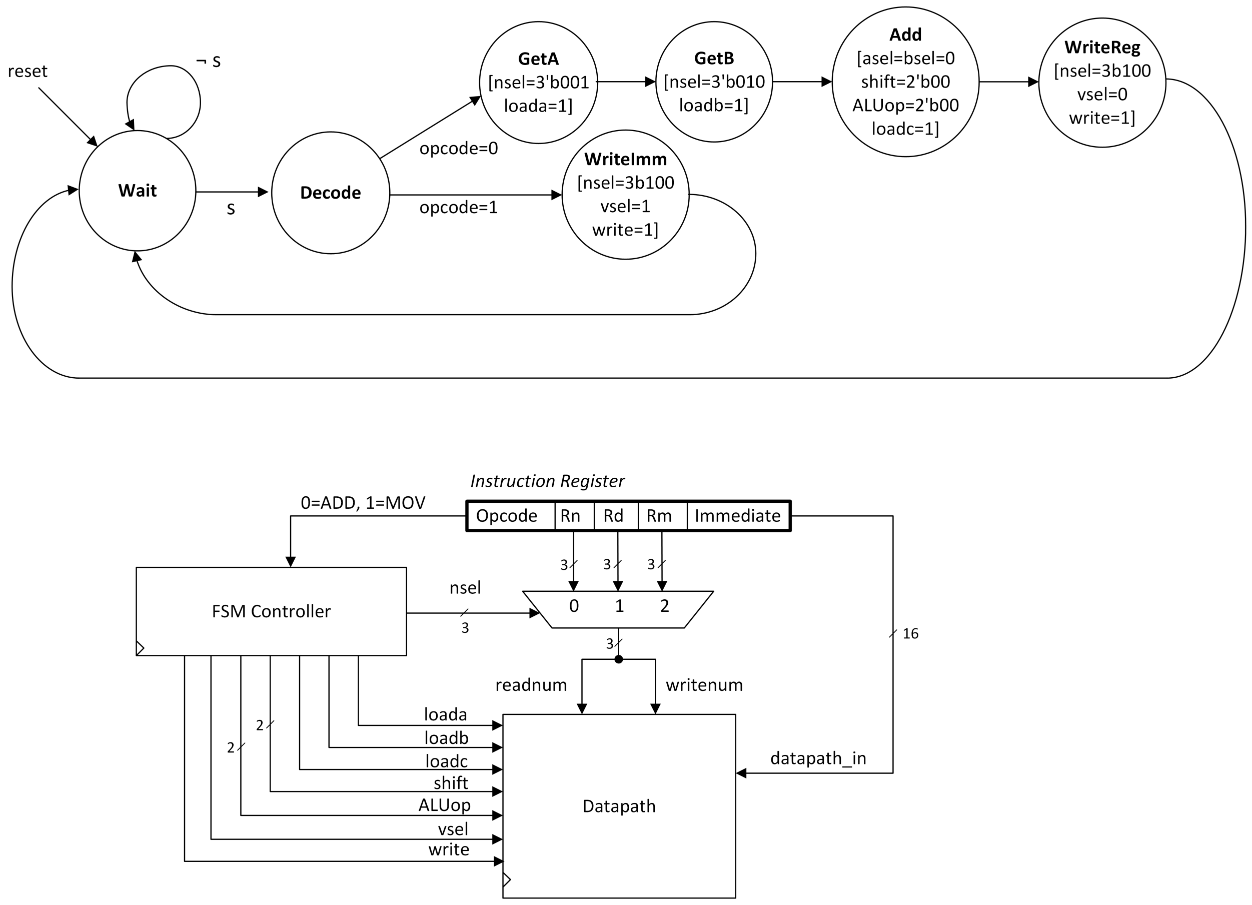

Figure 4: Supporting both MOV and ADD instructions.

ranked by performance, where performance is measured as one divided by the time in seconds it takes to run a test program. However, for your design to be ranked in this competition it *must* follow the encoding provided in the Lab 6 to 8 handouts. An alternative would be to provide your own assembler program (that you would need to create) for converting from the assembly syntax we provide into your own instruction encoding. However, finally, in an attempt to ensure uniform and objective grading of this lab, we will use an auto-grader for computing a portion of your mark. This auto-grader requires you follow the encoding in Table 1 along with requirements specified in Sections 3 and 4.

2 The Simple RISC Machine Instruction Set Architecture

This section introduces the six instructions you will implement in Lab 6. In Lab 7 and Lab 8 you will implement additional instructions.

2.1 Assembly Syntax and Encoding

Each row of Table 1 defines an instruction of the Simple RISC Machine ISA that you will implement in Lab 6. Table 1 is divided into three main sections described below.

The first column, labeled "Assembly Syntax", is a human-readable textual representation of each instruction. In assembly format each Simple RISC Machine instruction starts with an opcode mneumonic. For example, in "`MOV Rn,#<im8>`" the opcode mneumonic is `MOV`. We use this representation because it is easier to remember that `MOV` means "move a value from somewhere to a register" than it is to remember what "`110`" means. Here, "`110`" is the value under the heading "opcode" in the next column of Table 1. Returning to the Assembly Syntax column we see that the registers used by each instruction are also given a mneumnic representation. For example, for “MOV Rn,#<im8>” the instruction will move a value into the register Rn, where Rn can be R0, R1, ... R7. Recall these are the names of registers inside the register file that you built in Lab 5. Some instructions also include other information when written in assembly format using the Assembly Syntax. For example, in “MOV Rn,#<im8>” the part “#<im8>” represents an 8-bit binary number encoding a value between -128 and 127 (in decimal). The 8-bit value is stored inside the instruction itself. Values used by a program that are encoded inside instructions like this are typically referred to as an “immediate operand” in most instruction set architectures. In “MOV Rd,Rm{,<sh_op>}” the portion “{,<sh_op>}” is an optional shift operation (use “LSL#1”, “LSR#1”, or “ASR#1” for <sh_op>). This part of the instruction indicates what you want the shifter you built in Lab 5 to do when executing the instruction.

When programming in assembly syntax in Lab 7 and 8 you will write instructions using the notation in the first column of Table 1, but you replace Rn, Rm, and Rd with the names of specific registers between R0 and R7. Similarly, you will replace <im8> with a signed number between -128 and 127 and replace <sh_op> with one of LSL#1, LSR#1, or ASR#1.

The last column, labeled “Operation”, specifies precisely what the instruction should do using the datapath from Lab 5. For example, for “MOV Rn,#<im8>” this column contains “R[Rn] = sx(im8)”. The “sx()” part tells us that the 8-bit immediate value #<im8> should be sign-extended to 16-bits. That means we interpret the 8-bit value as a 2’s complement number and if the most significant bit of the 8-bit value, (bit 7) has the value 1, then we consider the 8-bit number to be negative (see Slide Set 6). When converting the number to 16-bits, we fill in the upper 8-bits with all 1’s. Similarly, if bit 7 was 0, we would fill in the upper 8-bits with all 0’s. The portion “R[Rn] =” indicates that this sign extended value should be placed into the 16-bit register identified by Rn. Note that the steps specified under the “Operation” column can take more than one clock cycle. The notation used in this part of the table is summarized as follows:

- Rn, Rd, Rm are 3-bit register number specifiers.
- im8 is an 8-bit immediate operand encoded as part of the instruction.
- <sh_op> and sh are 2-bit immediate operands encoded as part of the instruction. The value of sh is used to control the shifter you built in Lab 5.
- sx(f) sign extends the immediate value f to 16-bits.
- sh_Rm is the value of Rm after passing through the shifter connected to the Bin input to the ALU.
- status is a 3-bit status register. The three bits are called Z, V and N which stand for zero, overflow and negative, respectively.
- f(x) is a 3-bit value, corresponding to the three bits called Z, V and N in the status register, indicating whether x is zero, whether the calculation caused an overflow (see Slide Set 6), and/or whether x is a negative 2’s complement value (see Slide Set 6).
- R[x] refers to the 16-bit value stored in register x.

Finally, the set of columns under the heading *“Simple RISC Machine” 16-bit encoding* in Table 1 specify exactly how an instruction in assembly format should be converted into the 1’s and 0’s that will be placed into the instruction register. For example, the instruction “ADD R4, R0, R1” is encoded as the 16-bit value:

```
101 00 000 100 00 001
```

Bits 15, 14, and 13 of each instruction is a special “operation code” or “opcode” that identifies the basic operation the instruction performs. Thus, the first three bits above, 101, represents the “opcode” used for the instruction “ADD R4, R0, R1”. The opcode specifies the type of operation performed by the instruction. Instructions with opcode 101 are defined in the Simple RISC Machine ISA to be “ALU instructions”. ALU instructions read two registers perform an operation on the values in these registers using the ALU and then write the result back to a register in the register file. For such instructions the next two bits, Bit 12 and Bit

Assembly Syntax (see text)	“Simple RISC Machine” 16-bit encoding																Operation (see text)
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Move Instructions	*opcode*			*op*		*3b*			*8b*								
`MOV Rn,#<im8>`	1	1	0	1	0	`Rn`			`im8`								`R[Rn] = sx(im8)`
`MOV Rd,Rm{,<sh_op>}`	1	1	0	0	0	0	0	0	`Rd`			`sh`		`Rm`			`R[Rd] = sh_Rm`
ALU Instructions	*opcode*			*ALUop*		*3b*			*3b*			*2b*		*3b*			
`ADD Rd,Rn,Rm{,<sh_op>}`	1	0	1	0	0	`Rn`			`Rd`			`sh`		`Rm`			`R[Rd]=R[Rn]+sh_Rm`
`CMP Rn,Rm{,<sh_op>}`	1	0	1	0	1	`Rn`			0	0	0	`sh`		`Rm`			`status=f(R[Rn]-sh_Rm)`
`AND Rd,Rn,Rm{,<sh_op>}`	1	0	1	1	0	`Rn`			`Rd`			`sh`		`Rm`			`R[Rd]=R[Rn]&sh_Rm`
`MVN Rd,Rm{,<sh_op>}`	1	0	1	1	1	0	0	0	`Rd`			`sh`		`Rm`			`R[Rd]= ~sh_Rm`

Table 1: Assembly instructions introduced in Lab 6 (this table is explained in Section 2)

11, specify the ALUop input to the ALU. In this example these bits have the value `00`, which corresponds to the addition operation for the ALU you designed in Lab 5. The next three bits indicate the middle register (called Rn). In our example, this is R0, so the next three bits are 000. The next three bits encode the destination register that will be written by the instruction. In our example, this is R4. For ALU instructions the next two bits are the “shift” input to the shifter in your Lab 5 datapath. The final three bits specify the other register that is read by the instruction. In this example, that is R1, which is encoded as 001.

Rn, Rd, Rm are 3-bit numbers that refer to one of the eight 16-bit registers inside of the register file. in the first column “`{,<sh_op>}`” is an optional shift operation (use “`LSL#1`”, “`LSR#1`”, or “`ASR#1`” for `<sh_op>`); `sx()` means sign extend (described below); `sh_Rm` is the 16-bit value resulting from shifting Rm using the code “`sh`” (bits 3:4) as input to the shifter from Lab 5.

2.2 Instruction Descriptions

Next, we briefly summarize the operation of each instruction. The first instruction in Table 1, “`MOV Rn, #<im8>`”, takes bits 0 to 7 of the instruction (labeled “`im8`”) and “sign extends” these bits to a 16-bit value. Recall that in 2’s complement the most significant bit is a 1 if the number is negative and it is 0 if the number is positive. We can take an 8-bit positive number (with bit 7 equal to zero) and make a 16-bit positive number with the same value by simply concatenating 8 bits that are all zero. Similarly, we can take an 8-bit negative number (with bit 7 equal to 1) and make a 16-bit negative number with the same value by concatenating 8-bits with all 1’s. E.g., consider sign extending the number 3 from 8-bits to 16-bits:

```
8-bit representation of 3      16-bit representation of 3
       00000011                       0000000000000011
```

Similarly, consider sign extending -3 from 8-bits to 16-bits:

```
8-bit representation of -3      16-bit representation of -3
       11111101                        1111111111111101
```

After performing this sign extension, the MOV instruction writes the resulting 16-bit sign value to one of the eight 16-bit registers inside the register file. It identifies which of the 8 16-bit registers inside the register file to write using the 3-bit 8 to 10 of the instruction (labeled Rn). Recall with 3-bits we can uniquely identify 8 things since $2^3 = 8$. We return to discuss the second version of MOV further below.

The second MOV instruction, “`MOV Rd, Rm{,<sh_op>}`” reads Rm into datapath register B and then sets asel=1 to select the 16-bit zero input for the Ain input to the ALU. Since ALUop is “00” the ALU adds the zero on Ain to the shifted value of Rm on Bin and places the result in register C. The result is then written to Rd.

The next four instructions in Table 1 are called ALU instructions because their main purpose is to use the ALU you built in Lab 5. Such instructions are the main “workhorses” of any general-purpose computer design. The `ADD Rd,Rn,Rm{,<sh_op>}` instruction reads the contents of register Rm, optionally shifts the value one bit to the left (for example, "`ADD Rd,Rn,Rm, LSL#1`"), one bit to the right without sign extension ("`ADD Rd,Rn,Rm, LSR#1`") or with sign extension ("`ADD Rd,Rn,Rm, ASR#1`"). Then adds the result to Rn and places the sum in Rd. For example, if R0 contains 25 and R1 contains 50, then after executing the instruction "`ADD R2, R1, R0`" the contents of R2 would be 75.

The ADD instruction reads register Rn into register A and reads register Rm into register B. Bits 11 to 12 of the instruction register are directly fed to the ALUop input to the ALU. Since these bits are "00" for ADD instructions ALUop will be "00" which corresponds to addition. So, the Ain and Bin inputs to the ALU will be added together by the ALU. The operand in register B is shifted as specified by bits 3 and 4 that are fed directly from the instruction register into the "shift" input of your datapath from Lab 5.

The AND instruction is very similar to ADD. However, both the CMP and MVN instructions, while using the ALU, are different. CMP is the only instruction that should update the three status bits. For CMP we use the ALUop for subtraction however, we are only interested in the value of the status outputs of the ALU. E.g., we can use CMP to check if the value in R1 and R2 are equal by subtracting R2 from R1 and checking if the result is zero using the Z status flag (we will add logic to read the status flags in Lab 8). As with ADD and AND we can shift the contents of the B register. In Lab 8 we will add branch instructions that read the status register after it is set by a CMP instruction. For MVN we perform a bitwise NOT on the contents of Rm. As with the other ALU operations we can shift the value in the B register.

3 Lab Procedure

The changes for this lab are broken into two stages.

3.1 Stage 1: Datapath Modifications

Extend the mux on the data input to your register file to have the four inputs illustrated on the right in Figure 5. The `sximm8` input (which stands for sign extended 8-bit immediate) will eventually be driven by the Instruction Decoder you add in Stage 2 below. You will use this input to the mux when implementing control logic for the "`MOV Rn,#<im8>`" instruction in Table 1. Here `mdata` is the 16-bit output of a memory block you will be adding in Lab 7. Next, `sximm8` is a 16-bit sign extended version of the 8-bit value in the lower 8-bits of the instruction register. Next, `PC` is an 8-bit "program counter" input that will be explained and used in Lab 8. However, to avoid introducing bugs later, it is recommended you add a 16-bit `mdata` and 8-bit `PC` inputs to your datapath module in Lab 6 and connect them to the 4-input 16-bit multiplexer as shown in Lab 6. For Lab 6 you can "assign" zero to `mdata` and `PC`.

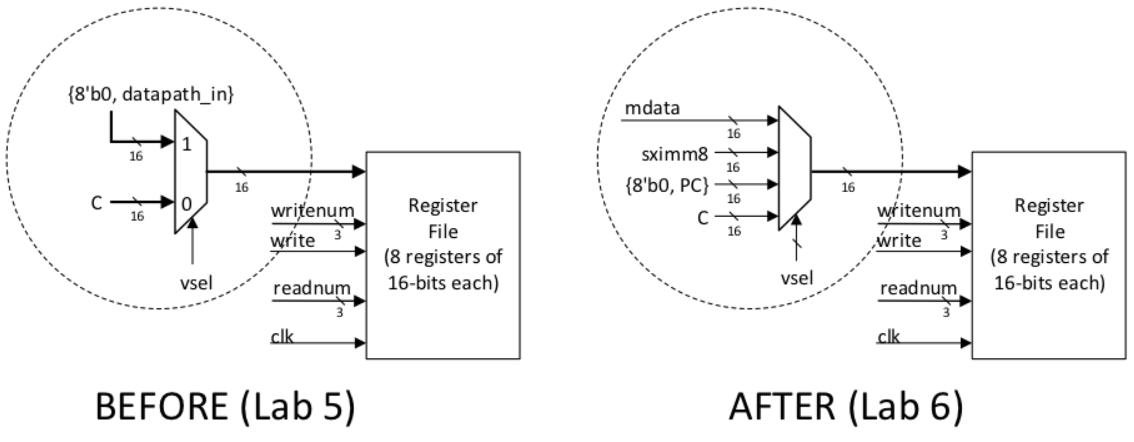

Figure 5: Modification to Lab 5 Datapath: Input to Register File

Next, modify the mux input to Bin as shown in Figure 6. Here `sximm5` is a 16-bit variable you should declare in datapath. Here sximm5 stands for "sign extended 5-bit immediate". We will connect sximm5 to another block in Stage 3.

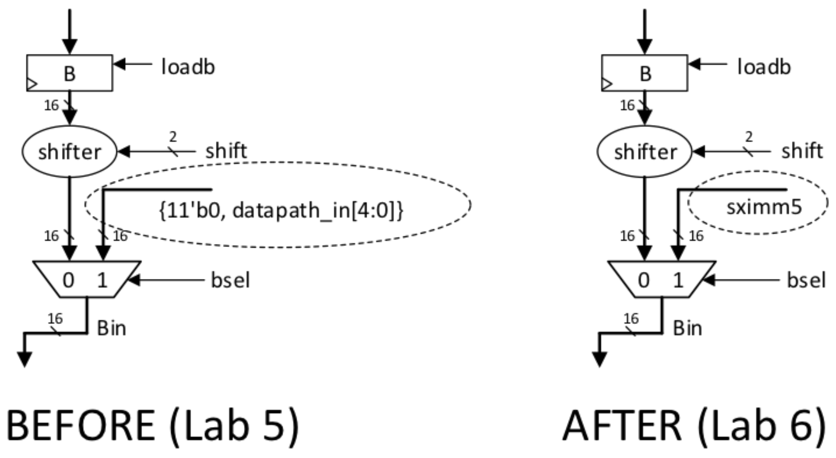

Figure 6: Modification to Lab 5 Datapath: "Bin" Multiplexer

Next, extend the status register to three bits. One bit should represent a "zero flag", which was what "status" represented in Lab 5. Another bit should represent a "negative flag" and be set to `1'b1` if the most significant bit of the main 16-bit ALU result is 1. The final bit represents an overflow flag. You should compute signed overflow as described in Slide Set 6 and/or Section 10.3 of Dally. In Lab 8 you will use the status flags to support "if" statements and "loops" in C. You will save time debugging in Lab 8 if you get your CMP instruction working now. Thus, for Lab 6 ensure these flags are correctly set by the CMP instruction. Note these status bits should not be changed by any other instruction besides CMP.

3.2 Stage 2: Datapath Controller

The next step is to add an instruction register, an instruction decoder block, and finally design a state machine to control your datapath. Inside a file `cpu.v` create a module `cpu` to instantiate and connect together these three components along with your datapath. A significant portion of your Lab 6 mark will be determined using an auto grader (see Section 4). To avoid losing marks your top level module must be called `cpu` and follow the specification below:

```
module cpu(clk,reset,s,load,in,out,N,V,Z,w);
  input clk, reset, s, load;
  input [15:0] in;
  output [15:0] out;
  output N, V, Z, w;
```

The value on `in` should be copied into your instruction register on the rising edge of `clk` if `load` is set to 1. If load is 0, the contents of your instruction register should NOT change. On the rising edge of `clk` if `reset` is 1 your state machine should go to a reset state. After being reset, your state machine should not perform any computations until `s` is set to 1 and there is a rising edge of `clk`, much like in the example state machine in Figure 4. The value on `out` should be the contents of register `C` inside your datapath. The outputs `N`, `V` and `Z` should provide the value of the negative, overflow and zero status register bits. The output `w` should be set to 1 if your state machine is in the reset state and is *waiting* for `s` to be set to 1. After executing an instruction, your state machine should return to this state. See the example state machine in Figure 4. Your `cpu` module must also be in a file named `cpu.v`.

Figure 7 illustrates how your state machine is connected to the rest of the circuit. The instruction currently being executed is stored in the 16-bit Instruction Register. The instruction register and *nsel* output of the state machine are inputs to the Instruction Decoder block, which is described below.

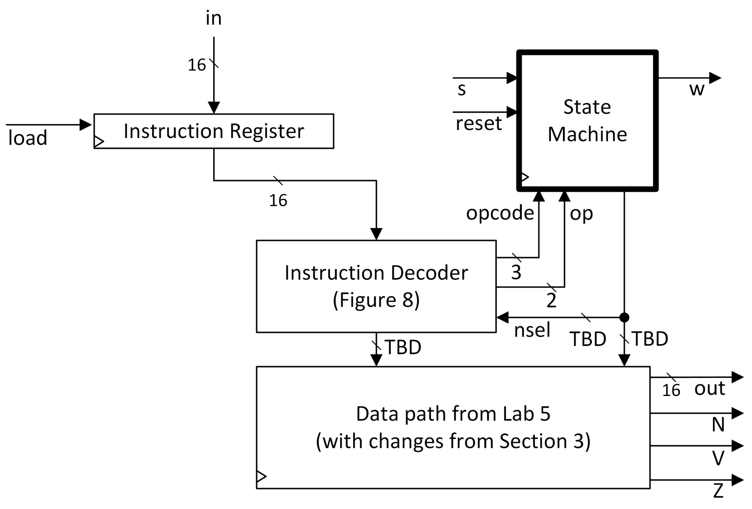

Figure 7: Your `lab6_top` module should contain your finite state machine controller (Controller), an instruction register, an instruction decoder and a datapath. In this figure, TBD means "to be determined" (by you!).

The purpose of the Instruction Decoder block is to extract information from the instruction register that can be used to help control the datapath. The Instruction Decoder block in Figure 7 should implement the logic shown in Figure 8. Your state machine should drive any datapath inputs not set by the decoder block (e.g., labeled "TBD" in Figure 7).

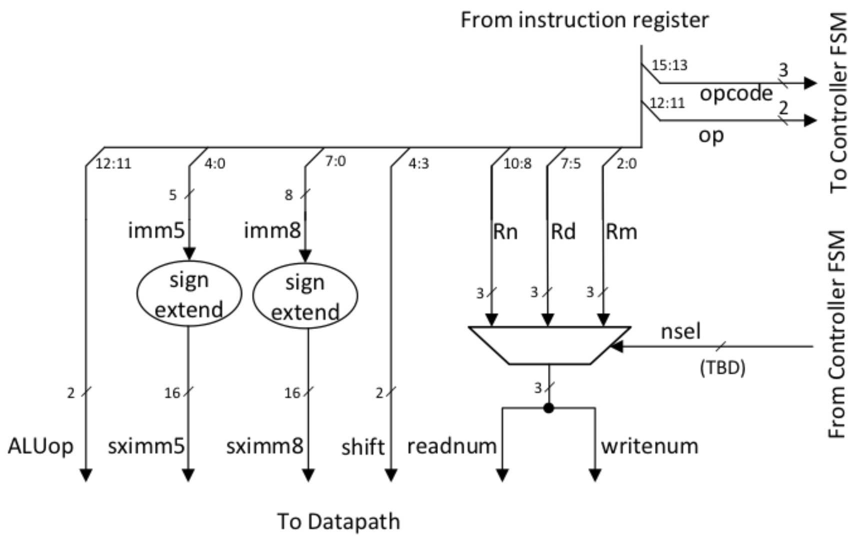

Figure 8: Instruction Decoder

The output of the state machine should be the settings of all the inputs to the datapath, the signal `nsel` used to select which register to connect to `readnum` and `writenum`, and the `w` output used to indicate to the autograder that your state machine is (or is not) in the wait state. The inputs to the state machine are `clk`, `reset`, the start signal `s` the `opcode` and `op` fields of the current instruction in the instruction register.

How should you design your state machine? The examples shown in Figure 3 and 4 used a Moore type finite state machine where the output depends only on the current state. You can do the same or, if you want, use a Mealy state machine. If you are starting early, you may want to read ahead in Slide Set 7 to see how a state machine can be coded up using a **`casex`** statement. The coding style shown there can result in cleaner less buggy finite state machine code.

Regardless of which type of state machine or the coding style you use, the best way to *design* your state machine is in stages. In the first step, get your state machine to work for a single instruction from Table 1. Pick an instruction from Table 1 and think through what steps you need to perform with your datapath from Lab 5 to perform the steps listed in the "Operation" column. One way to do this is by referring to Figure 1 in the Lab 5 handout. Work out the number of clock cycles it takes to perform the data operations required for the instruction. Each cycle will require an additional state beyond the "Wait" state shown in the example in Figure 3. After the last state required to execute the instruction on your datapath, your state machine should return to the "Wait" state as shown in the example in Figure 3. In the "Wait" state make sure your `w` output is 1 so the autograder (or your own test bench) knows the computer has finished executing the instruction and is ready to execute a new instruction.

Once you have very carefully tested your first instruction is working, both in ModelSim and on your DE1-SoC, you should check that version into your revision control system (e.g., git) in case you make a change that breaks that first instruction while modifying your state machine to support a second instruction. When adding the second instruction, you should add a "Decode" state like that shown in Figure 4. You should reuse this Decode state when adding any subsequent instructions. After adding the Decode state, figure out the states corresponding to the steps required to execute the new instruction on your datapath. You will add additional states (e.g., like WriteImm in Figure 4) for this new instruction. Now the "Decode" state has two potential next states. To decide which next state your state machine should go to from "Decode" use the `opcode` and `op` values used for encoding the instruction you are adding (find these in Table 1). See Figure 4 for a simplified example showing how to determine which state to go to after "Decode".

After adding each instruction test that *both* the new instruction and the prior instructions work (both in ModelSim and on your DE1-SoC). Unless you are VERY confident in your Verilog coding abilities, you should NOT attempt to code up a state machine for all instructions before doing any testing. If you do, you will spend much more time trying to figure out the source of even a single bug than you would have by testing each additional instruction as you add it. Since you need to show a testbench as part of the marking scheme, why not create it as you go and use it to help you save time by catching bugs early?

To reduce the complexity of your state machine you may want to see if you can find ways to reuse states added for earlier instructions when adding a new instruction. However, this is not required.

The input to your top level module is the encoded instruction. Thus, to test your overall design you will first want to create some simple programs that you can input to your instruction register one at a time. To do that, first write a textual assembly code representation and only then encode each instruction into into 1's and 0's using Table 1. As an example, the following test case.

```
MOV R0, #7 // this means, take the absolute number 7 and store it in R0
MOV R1, #2 // this means, take the absolute number 2 and store it in R1
ADD R2, R1, R0, LSL#1 // this means R2 = R1 + (R0 shifted left by 1) = 2+14=16
```

can be encoded as:

```
1101000000000111
1101000100000010
1010000101001000
```

For full marks on the lab you need to encode additional instructions. You can use the `lab6_top.v` file we provide to test your design on the DE1-SoC. This is a modified version of `lab5_top.v` and works in a similar way.

4 Marking Scheme

Both partners must be in attendance during the demo. You **must** include at least one line of comments per always block, assign statement or module instantiation and in test benches you must include one line of comments per test case saying what the test is for and what the expected outcome is. For your state machine include one comment per state summarizing the datapath operations and one comment per state transition explaining when the transition occurs.

You will lose marks if your handin submission does not contain a quartus project file, modelsim project file, or programming (.sof) file. You will also lose marks if your handin submission is missing any source code (whether synthesizable or testbench code) or waveform format files.

Your mark will be computed as the sum of the following rubric:

Stage 1 changes [1 Mark] For your Stage 1 code in datapath.v and being able to explain the associated Verilog to the TA. You may lose marks here for not following the style guidelines and/or for lack of commenting in your code or if you have no testbench in datapath_tb.v for your Stage 1 changes.

Stage 2 changes [3 Marks] Your state machine must include sufficient comments. It is recommended but not required to use the second style for coding a state machine shown in Slide Set 5 (using a separate vDFF module for the state, an assign statement for the reset mux, and a separate always block for your combinational logic for state transition logic and output logic). You may want to look into using **`casex`**, introduced in Slide Set 7 when specifying your state machine. Again, this is not required. During your marking session you must demonstrate that your state machine works using your submitted testbench with ModelSim and your submitted `lab6_wave.do` file. Your mark for this part will be:

3/3 If your state machine in cpu.v implements all instructions in Table 1 and you demonstrate a set of very convincing test cases in cpu_tb.v of your own devising including at least three tests for each instruction in Table 1. Each test should be designed to test a different part of your design that might have an error and you should be able to explain to your TA what potential error or mistake in coding that design is meant to catch. It is your responsibility to think of what might go wrong when coding your state machine and combining it with your datapath and how the tests might catch those errors.

2/3 If your state machine implements all the instructions from Table 1, but you have less than three tests for each instruction in your cpu_tb.v or you cannot provide examples of the types of bugs that the tests might catch.

1/3 If your state machine does not implement all of the instructions in Table 1 or your state machine includes very few comments or comments that are not very meaningful, or you do not include a testbench.

DE1-SoC Demo [2 Marks] For demonstrating your CPU works on your DE1-SoC using a test case of your own devising involving some of the LEDs on the DE1-SoC. To get 2/2 marks here this test case MUST work AND use ALL of the instructions in Table 1. You will get 1/2 here if this test case works and it uses at least three different types of instructions from Table 1 (but not all of them). If you cannot get a test case involving at least three different types of instructions from Table 1 your mark will be 0/2.

Autograder [4 Marks] Finally, four marks will be assigned objectively by an auto-grader that will test your `lab6_top` using a variety of inputs. **Your `lab6_top` must follow the specification in Section 3.2 and moreover, the output of the registers inside your register file must be accessible via the hierarchical names `lab6_top.DP.REGFILE.R0` through names `lab6_top.DP.REGFILE.R7`.** Thus, your datapath must be instantiated with the instance name `DP`, and inside of your datapath module your register file must have the instance name `REGFILE`, inside of your register file, the 16-bit registers R0 through R7 must be accessible on signals (wire or reg) called R0 through R7. To ensure this may need to make minor changes to your datapath from Lab 5. Your mark for this part will depend upon how many instructions pass our test cases and will be:

4/4 If every single *type* of instruction in Table 1 passes *all* of the auto-graders test cases.

3/4 If all but one *type* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **one** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

2/4 If all but two *types* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **two** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

1/4 If all but three *types* of instruction in Table 1 passes *all* of the auto-graders test cases. This means for example, if you did not have time to get **three** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

0/4 If four or more *types* of instruction in Table 1 each fail at least *one* of the auto-graders test cases. This means for example, if you did not have time to get **four** of the instructions in Table 1 working, but you got all the other instructions working (as judged by our autograder), then you would get this mark.

IMPORTANT: Note that for the autograder to be able to evaluate your design ALL code instantiated in your `cpu` module **MUST** be synthesizable by Quartus 15.0 AND you MUST include a ModelSim (.mpf) project file that MUST be called `lab6.mpf` that MUST include an unmodified version of `cpu_tb_autograder_example.v` provided for use with Lab 6 on the Piazza website. To ensure your code is compatible with our autograder you should both ensure you can download your working design to your DE1-SoC **AND** be sure you can simulate the `cpu_autograder_example_tb` testbench module we provide in `cpu_tb_autograder_example.v` provided on Piazza. You should be sure you get the message "`INTERFACE OK`" in the ModelSim transcript window when you do this. Please note that the message "`INTERFACE OK`" does **NOT** ensure your Lab 6 submission will pass any of our autograder tests, but if you do not get this "`INTERFACE OK`" message you will get 0/4 for this part.

5 Lab Submission

Submit all files by the deadline on Page 1 using handin. If you are working with a partner, but each partner happened to create their own independent solution code, then you must submit both solutions through Partner #1's account following the instructions in the Peer Help Policy to use directories named "not-for-demo" and "for-demo".

IMPORTANT: All students including those working alone MUST include a CONTRIBUTIONS file for Lab 6 through 8. Your CONTRIBUTIONS file MUST accurately and truthfully report the contributions of each student. While one partner may draft the CONTRIBUTIONS file, by submitting a CONTRIBUTIONS file Partner 1 is officially stating for the record that BOTH partners have affirmed the contents of the submitted CONTRIBUTIONS file are accurate and truthful. If you are Partner 1 you should obtain an email from your partner confirming their approval of the version of the CONTRIBUTIONS file you are submitting before you submit it and you should retain that email for your records. We may ask to see this email at a later date. Follow the guidelines in the Lab 3 handout for the types of things to comment upon in this file. **If**

no CONTRIBUTIONS file is submitted via handin your grade for the lab will be at most 5 out of 10 (even if you are working alone).

Use the same procedure outlined at the end of the Lab 3 handout except that now because you are submitting Lab 6, you would use:

```
handin cpen211 Lab6-<section>
```

where `<section>` should be replaced by your lab section. Remember you can overwrite previous or trial submissions by adding `-o`.

For your demo make sure to submit a `lab6_wave.do` file that shows important internal signals such as the controller state, the instruction register, datapath control inputs, registers A, B and C and any other signals you think may help you locate bugs more quickly. Also be sure to include your .sof, .mpf, .qpf files as if any of these are missing you may lose significant marks.

6 Lab Demonstration Procedure

As with Lab 3 to 5, your TA will have your submitted code with them and have setup a "TA marking station" where you will go when it is your turn to be marked. Be sure to bring your DE1-SoC in case the TA does not have theirs and/or they need to mark multiple groups in parallel.

7 Hints and Tips

You may find it helpful to use the Verilog include directive to define constants used in multiple files.

After getting your Verilog to compile in ModelSim, but before running any simulations in ModelSim, it is worth try to compile your synthesizable modules in Quartus just to look at the warnings. Quartus provides useful warnings for many "silly mistakes" that ModelSim happily ignores. If you see no suspicious warnings in Quartus, then move on to simulating your testbench in ModelSim.

When using "`$stop;`" in a testbench and running with "`run -all`" in ModelSim a source window will pop-up when "`$stop`" is reached. If you use a text editor other than ModelSim to edit your files (e.g., vi or emacs), make sure to close this window before restarting simulation. If you for any reason modify your testbench outside of ModelSim (perhaps to add a test case) and you then restart simulation you will get a long set of about 50 pop-ups saying the file was modified outside of ModelSim. If you forget and this happens, note you can likely close these roughly 50 dialogs faster then restarting ModelSim by clicking on "Skip Messages" then selecting "Reload" repeatedly.