ECE352 Final Project

Overview

The final project will involve converting a basic multicycle processor to a pipeline processor. On this package you will find:

- A verilog implementation of a multicycle processor
- An assembler for the processor
- Documentation describing the multicycle processor

Your *(tentative)* tasks over the next weeks will be to:

1. Add a performance counter that will count the number of clock cycles your processor took to execute a program.
2. Add a “nop” and a “stop” instruction. The stop instruction will signify the end of your program and will stop the performance counter.
3. Modify the processor to implement a 5-stage pipeline.
4. Enhance the processor to handle hazards automatically.

Guidelines

Make sure you are familiar with the multicycle processor and its implementation by reading the documentation in **multicycle_documentation**. Note this has some instructions for Quartus regarding modifying memory contents, which will not apply when using Modelsim.

The "initial memory" must have the program and data you want to use for testing; you can generate those using the assembler (in **multicycle_assembler**). Note that the assembler doesn't support the new instructions you're adding in the project, so you'll have to assemble them yourself and specify them as data bytes (or you can add support for them to the assembler). Also note that the assembler is picky with the syntax, all instructions should be indented otherwise they are interpreted as labels.

The **multicycle** directory contains the actual verilog source code files and Quartus project for the processor. You will be mostly using the Verilog for simulation under Modelsim.

Finally, in **ECE352_specific** you have the documentation about the tasks you need to perform in this project lab, as well as some test programs you'll have to demonstrate during the different weeks, and the dual-ported memory you'll have to use to enable pipelining.

Project Guide

This section will give you a rough guide for implementing your processor.

1. Add a "nop" instruction to your processor. The `nop` instruction takes no arguments and has the opcode `nop`. It has the following encoding:

   ```
   nop: 00001010
   ```

2. Add a 16-bit performance counter that will count the number of cycles your processor takes to execute a program. The counter will have the following behavior:

   a. It will be initialized to zero whenever you hit the processor reset button (or assert the reset signal in Modelsim).

   b. It will increment by one every clock cycle.

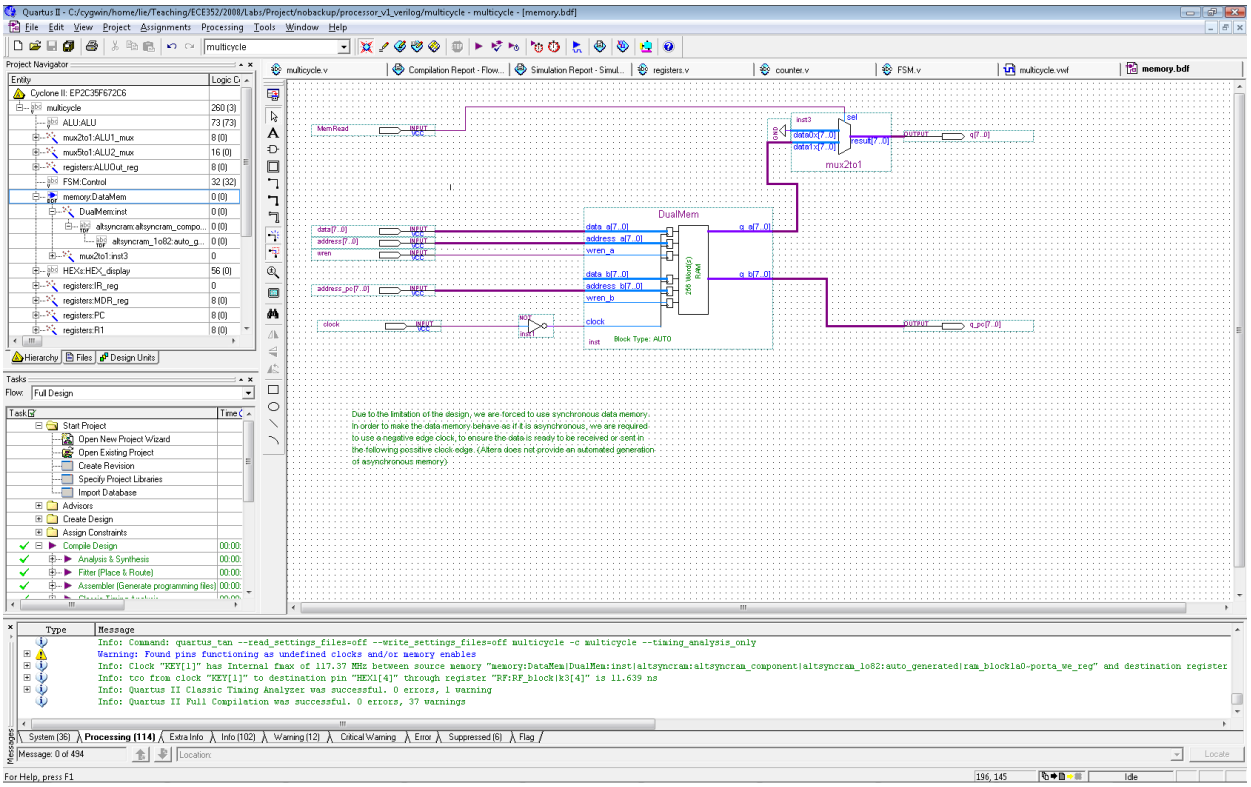
   c. Toggling `SW[2]`, will display the value of this counter on the HEX digit displays. Since you will be running under Modelsim, it's enough to have the counter as a signal that can be visualized as a waveform.

   d. This counter will stop incrementing when your processor detects a "stop" instruction in the instruction decode stage.

   You must also add this stop instruction to your processor. The stop instruction takes no arguments and has the opcode `stop`. It has the following encoding:

   ```
   stop: 00000001
   ```

3. Convert the single memory into a dual-ported memory. This memory will have a read-only port for the Instruction Fetch (IF) stage and read-write port for executing loads and stores in the EX stage. To do this, unzip DualMem.zip into your project directory and recompile your project. This will install the new dual ported memory in your project. If you open your project in Quartus, by double clicking on the new memory module in your project hierarchy you will be able to view a schematic of the new element:



Notice that the new memory has some new signals:

- The `address_pc` port takes an address to be read.
- The `q_pc` output port will hold the value of memory at the address given by `address_pc`.
- The remaining signals have the same behavior as before. This memory is capable of performing operations on both ports simultaneously in the same cycle.

Now go back to the top level `multicycle.v` and edit the verilog to use these new signals. How should you hook up these new signals?

4. Replicate the instruction register so that there is one for the Register Fetch (RF), Execute (EX), and Write Back (WB) stages. The control signals for each stage should be determined only by the value of its local instruction register. You should also:

    a. Create separate adders for incrementing the program counter after every instruction. The PC increment cannot use the adder in the EX stage.

    b. Instructions that do not use all the stages must set control signals to do nothing when they reach an unused stage. *Hint: you can reuse the control signals that the nop instruction will set for the unused stages of other instructions.*

    c. Branches are tricky because you don't know whether to take the branch or not until the previous ALU operation has completed. Delay the computation of the new PC after a branch until the branch enters the EX stage and all proceeding instructions have gotten past the EX stage. At this point examine the N or Z bit and then write the result directly into the PC register at the end of the EX stage. Note this allows you to reuse the ALU in the EX stage. *Hint: Be careful not to clobber the N and Z registers when there is no ALU operation, they must store the results of the most recent ALU operation for branches to work!*

    Your processor should be able to properly execute any program that:

    - Does not have any data hazards
    - Follows every branch with a 3 `nop`'s (to avoid the control hazard). Note that these are known as "branch delay slots".

5. Enhance the processor to handle control hazards. Detect the presence of a branch in the ID stage and stop incrementing the PC until the branch resolves. Your processor should not need programs to contain branch delay slots to execute properly. *Hint: Create a state machine for the ID stage that remembers whether a branch has been fetched or not, and whether it has resolved or not.*

6. Enhance the processor to handle data hazards. To do this:

    a. Enhance the ID stage to check the later instruction registers in the RF, EX and WB stages for a conflict.

    b. When a conflict is detected, prevent the program counter from incrementing and stall the current instruction from proceeding to the RF stage. Insert `nop`'s into the pipeline until the source of the conflict has completed it's WB stage.

    At this point your processor should be able to execute programs that contain control and data hazards properly.

Building the assembler

The assembler can be built by running **make** on the **multicycle_assembler** directory inside the distribution, given a working C++ compiler (it is a single C++ file). For your convenience, a prebuilt executable for Windows is included (**asm.exe**)

Grading Scheme *(tentative)*

To attain full marks, you must complete all 6 components in the previous section. This project is fairly complicated and the processor design is considerably larger than any of the designs you have worked previously in the labs. To help keep you on track, we will have two check-points which you must meet. The project will be graded out of 16. The checkpoints will be worth 2 marks each and the final project will be worth 12 marks. Note that these checkpoints are *minimum* progress requirements, you are strongly advised to try and *stay ahead of them!* Grades are assigned as follows:

1. Checkpoint #1: Have steps 1-3 completed [2 marks + 2 marks if completed by end of 1st lab]
2. Checkpoint #2: Have step 4 completed. [4 marks + 2 marks if completed by end of 2nd lab]
3. Completed project: Have all step 5 [3 marks] and step 6 [3 marks] completed by end of project.

You are required to have at least one diagnostic program that demonstrates and tests the functionality required in each checkpoint. We suggest having several such programs that test all possible cases. A few diagnostic programs are available under **ECE352_specific/TestPrograms**, these will be used to test your modifications.

Simple Multi-Cycle Processor

Overview

This implementation of a simple multi-cycle processor consists of the datapath, control (FSM), and various inputs and outputs. The processor supports only fixed-length 8-bit instructions and data, and can only access memory one byte at a time. Overall, 10 instructions are implemented in hardware.

Functional Description

The inputs and outputs include:

- Inputs:
  - reset – clears the values of all registers and control signals, resets the condition flags, and resets the control FSM (KEY[0])
  - clock – all writes and cycle transitions happen on the positive edge of the clock (KEY[1])
  - HEXsel[1..0] – selects one of the four registers to be displayed on the hex displays; this exists for the users of the DE1 board, which does not have many hex displays. The chooseHEXs circuit component can still be used to display four values simultaneously (for example, of the four registers), but that requires eight hex displays, available only on the DE2. Users of the DE2 can make the necessary circuit connections if they so desire. (SW[17..16])

- Outputs:
  - PCWrite – LEDR[17]
  - AddrSel – LEDR[16]
  - MemRead – LEDR[15]
  - MemWrite – LEDR[14]
  - IRLoad – LEDR[13]
  - R1Sel – LEDR[12]
  - MDRLoad – LEDR[11]
  - R1R2Load – LEDR[10]
  - ALU1 – LEDR[9]
  - ALU2[2..0] – LEDR[8..6]
  - ALUOp[2..0] – LEDR[5..3]
  - ALUOutWrite – LEDR[2]
  - RFWrite – LEDR[1]
  - RegIn – LEDR[0]
  - FlagWrite – LEDG[7]
  - N (negative) – LEDG[1]
  - Z (zero) – LEDG[0]
  - Register Display
    - upper 4 bits HEX1[6..0]
    - lower 4 bits HEX0[6..0]
    - Note: as mentioned previously, the circuit can be modified

to support the remaining 6 hex displays, available on the DE2 board.

Instructions (Hardware)

- Addition (add)
- Subtraction (sub)
- NAND (nand)
- Shifting (shift)
- OR with Immediate (ori)
- Load (load)
- Store (store)
- Branch If Positive Zero (bpz)
- Branch If Zero (bz)
- Branch If Not Zero (bnz)

Assembler

An assembler is provided with the processor to aid students with the process of creating simple programs. It generates a Memory Initialization File (MIF) that can be used to initialize the memory of the processor design. The assembler can be invoked in the shell (after logging in using SSH) by typing “asm”. If this does not work, make sure that the assembler is located in the working directory or is configured to run properly. The assembler accepts one mandatory argument and one optional argument. The first argument is the name of the file containing the assembly program, and must be provided. It can have any file extension, but we suggest using “.s”. The second argument is optional, and is the name of the output MIF file. If it is not provided, the default output filename is always “data.mif”. This name was chosen because the Quartus implementation of the processor assumes “data.mif”, located in the same directory as the design, to be the initialization file. This can be changed, however, using a wizard in Quartus.

The assembler is based on the concept of columns, similar to many other assemblers. The first column begins at the first character of a line, the second column is separated from the first by any amount of whitespace, and the third column is separated from the second in the same way. If more columns are present, the assembler will issue an error, unless the text immediately after the third column is a comment.

Comments are indicated using a semicolon. The assembler will ignore anything following a semicolon, unless the semicolon interrupts an instruction or directive, in which case an error will be issued.

The first column must either be blank, or contain a unique label. Please note that this assembler only allows labels on the same line as the instruction/directive to which they refer. A label cannot appear on a line by itself.

The second column must contain a valid keyword identifying the instruction or directive. Finally, the third column must contain the parameters/operands. Please note that in this assembler no spaces are allowed in the third column. If two operands are required, they are separated using a comma without spaces in between.

Numerical constants can be specified in binary, octal, decimal, or hexadecimal format. Binary values are preceded with a '%' sign, octal values are preceded by an initial zero (0), and hexadecimal values are preceded with a '$' sign. If none of these characters precede the number, it is assumed to be in decimal format. Please note that negative signs are only supported when using the decimal format.

Overall, 14 keywords are supported:

- add r1,r2
- sub r1,r2
- nand r1,r2
- shift r1,imm3
    - shiftl r1,imm2
    - shiftr r1,imm2
- ori imm5
- load r1,(r2)
- store r1,(r2)
- bpz ADDR
- bz ADDR
- bnz ADDR
- org imm8
- db imm8

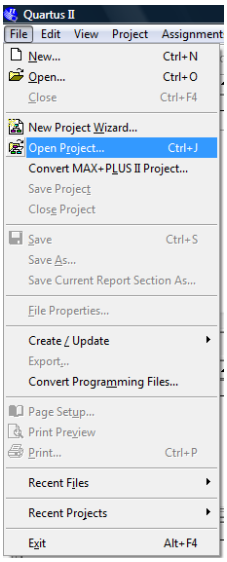
The shift instruction, as described in the course notes, accepts a 3-bit value. Bit #2 determines the direction (0 – right, 1 – left), and bits #1 and #0 determine the number of shift positions. For convenience, "shiftl" and "shiftr" are provided to shift left and right, respectively. They accept a number between 0 and 3, specifying the number of shift positions. They are not true instructions, because they both get converted to an equivalent "shift" instruction. Please note that imm2, imm3, and imm5 must be unsigned positive numbers (i.e. negative signs are not allowed). The brackets around "r2" for "load" and "store" are also required. ADDR can be either a previously defined label, or a 4-bit signed number. Please note that the effective branch address is calculated as follows: current_address + imm4 + 1. Do not forget about the 1 that is added, especially when using backward branching. "org" can be used to set the current address. It is specified here to use imm8, but the actual width of the number depends on the memory size. For this lab, a number between 0 and 255 can be used. "db" is used to place a single byte of at the current address. The byte can be either an

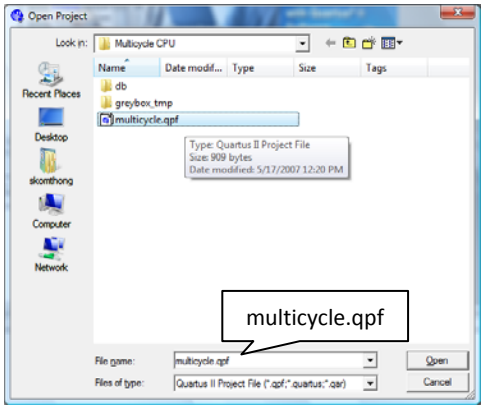
8-bit number, or a character in single quotes. Only one byte can be specified per "db" directive. While "org" and "db" are not instructions, they can be associated with a label, identifying the new address that they define (in the case of "org"), or the address at which they place their data (in the case of "db"). Please note that a label associated with an "org" statement and with an instruction or directive immediately following it identify the same address, unless another "org" statement is the following directive.

Changing Initial Memory Contents

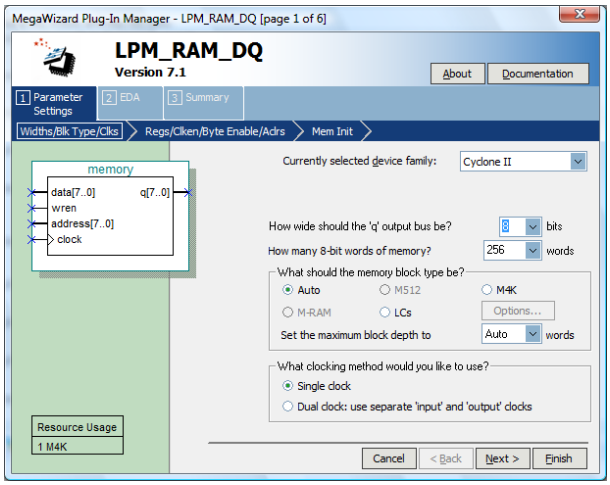
By default, the memory block used in the design assumes that its initial contents come from a MIF file called "data.mif". However, it may sometimes be useful to be able to change this to a different file name.

1. To change the initial contents of the memory, first open the Quartus project called "multicycle.qpf".

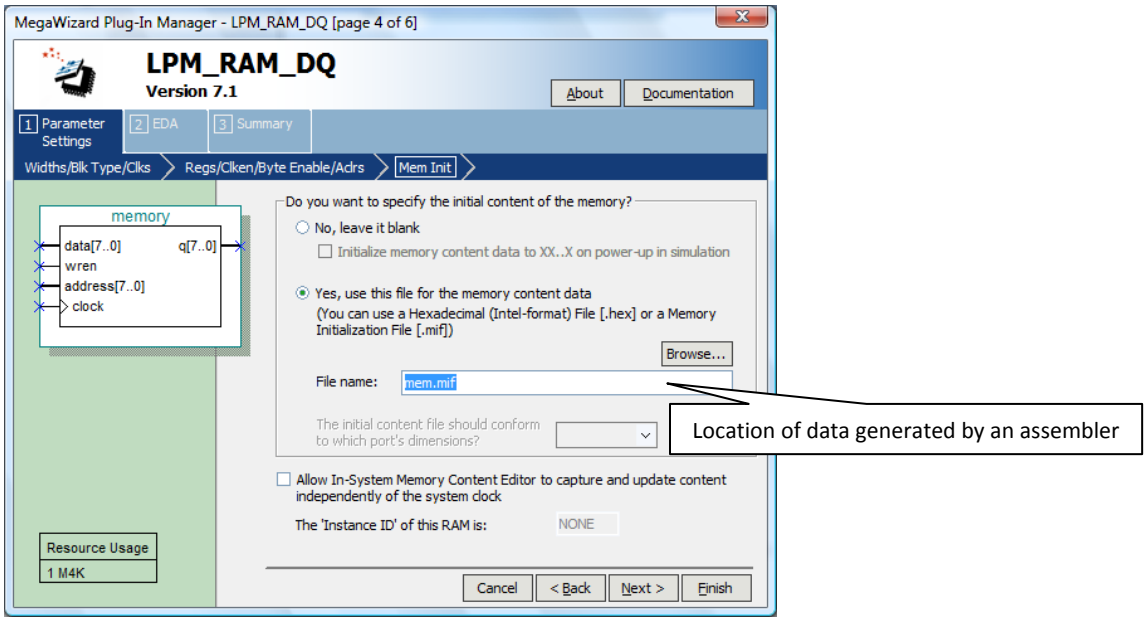




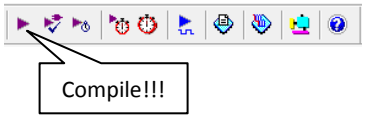
2. Double click on the component labeled "Data Memory".



3. Select “Mem Init”, and locate the MIF file generated by the assembler.



4. Compile the project.



5. Run the program!

Simulating the multicycle processor with ModelSim

1. Open your processor project in Quartus II and go to the *Files* tab. Double-click on *memory.bdf* to open the instruction memory schematic. The simulator does not understand schematic files, so we have to automatically generate a Verilog file from the schematic.

2. Choose the menu entry *File->Create/Update->Create HDL Design File from Current File…* Make sure that the export file name is *memory.v* and that Verilog HDL is selected. Click OK. Leave Quartus.
   (Note for ECE352: You will later in the project exchange the memory schematic with a new one. Make sure to re-export a new *memory.v* after you did that.)

3. Start ModelSim Altera Starter Edition from the Start Menu

4. *File->Change Directory* to your project directory, probably named *processor_v3_verilog*

5. On the ModelSim command line, you can enter file systems commands like **cd** and **dir** (Unix commands like **ls** and **pwd** also work). Check with **dir** if the expected project files are in the folder.

6. There are three scripts in the project folder, **compile.do**, **simulate.do** and **wave.do**.
   Type **cat compile.do** to see what the first script does:
   a. **vlib work** creates a local folder called *work* in which all the compiled simulation files are written. The folder functions as a working library of compiled components.
   b. The **vlog** command compiles each Verilog file in the project into the working library.

7. Type **do compile.do** to compiles all the components in the project. If you open the tree called "work" in the *library* window in the top left, you can see all the compiled Verilog modules.

8. Type **cat simulate.do** to see what's in the simulation script:
    a. **vsim** starts the simulation. It's **–L** parameters are the built-in Altera component libraries that the simulation uses (e.g. for the instruction RAM block that we are using). The last parameter is the top-level module that we are simulating. Note that this is not the top-level module of the Quartus project (*multicycle*), but instead the testbench module *multicycle_tb*, which instatiates our whole processor as a submodule and generates simulated inputs like the clock and the reset. Run **cat multicycle_tb.v** to see what the testbench looks like.
    b. **mem load** loads data from a *.mem file into the simulated instruction RAM. The file used is provided as the only command-line argument. When you run a test program through our provided assembler, it produces a file *data.mif* which can be used to initialize the RAM in the Quartus project, and a *data.mif.mem* file which the simulator can read. They both hold the same data, just in different formats.

9. Type **do simulate.do data.mif.mem** to run the simulation. You can see that the window makeup of the simulator changes. To the left is now a window called *sim* which shows the module hierarchy for the simulated system. If you click on a module, the window *Objects* to the right of the *sim* windows shows all the wires and regs that are in the selected module.

10. Type **do wave.do** to open a new window that shows you simulated signals. You can see signals like *clock* and *reset*. Vectors have a little plus sign beside your name with which you can see every bit separately. However, right now all the signals are indicated as 'x' or 'z' and now waveforms can be seen. This is because we are still at simulation time 0.

11. Type **run 1000 ns**. Now you can see the first microsecond of system simulation. Note how the testbench briefly asserts *reset* and then releases it, as the testbench code described. You can also see the clock oscillate.

12. The best thing about ModelSim is that you can look at internal signals that interest you. In the *sim* window, click on DUT (device-under-test, the name of the processor instance). In the *Objects* window, left-click *MEMwire*, drag it into the bottom of the wave window and release it. You cannot yet see anything because Modelsim did not log the signal data when it simulated. If you type **run 500 ns**, you can see how the signal develops between 1000 and 1500 ns. If you want to see the signal from the start, type **restart** and **run 1500 ns**. On the top of the window, click the magnifier icon with the plus sign a few times to a look at a shorter instant in time, so you can the single signal changes. The *MEMwire* value is still hard to read because it's displayed in binary. Right-click on *MEMwire* and pick *Radix->Hexadecimal*.

13. Type **cat wave.do** to finally see what the waveform window script looks like. We won't go through it in detail, but can see that it is a script that opens the window, adds all the waves we see and sets a few more display parameters. You can save your own window setting by going to the Menu *File->Save Format*. Save the settings as *wave1.do*. If you execute **do simulate.do** and **do wave1.do** again, you see that *MEMwire* is now included and you see the window in time that you had last selected.

14. You can now edit your Verilog files and simulate the modified system. A few <u>important</u> reminders:
    - After each Verilog change, you have to-recompile and re-start the simulation.
    - If you add an extra Verilog file, don't forget to include it in the compile script.
    - When you change the memory type in the first project steps, don't forget to re-generate a *memory.v* file
    - If you generate new memory files with the assembler, do not forget to change the **.mem* file name in the *simulate.do* file
    - If you get different results on the board than from the simulator, make sure that when you simulate with the *XY.mif.mem*, you use *XY.mif* to initialize the memory in Quartus.

Multi-cycle Processor Modification

Introduction

Modification of the multi-cycle processor deals with changing the datapath and control. In this design, the datapath is implemented in '*multicycle.v*' (*Verilog implementation),* and in '*multicycle.bdf*' (*schematic implementation)*, and the control is implemented in '*FSM.v*'. When implementing the design, modifying the control first is suggested.

Review of Verilog

Verilog code consists of three major parts: the header, the module declaration, and the body. Module declaration consists of the keyword '*module*' and its module name, and all inputs and outputs appear in brackets. See Figure 1.1.

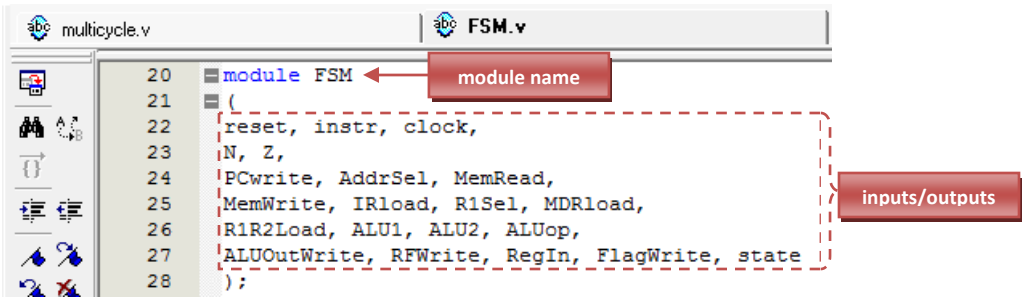


Figure 1.1 – module declaration with the module name and its inputs and outputs

To add an input or output in Verilog code, simply add the name inside the brackets of the module declaration. Then, add a line indicating whether it is an input or output, together with its size (if the signal is more than one bit in size).

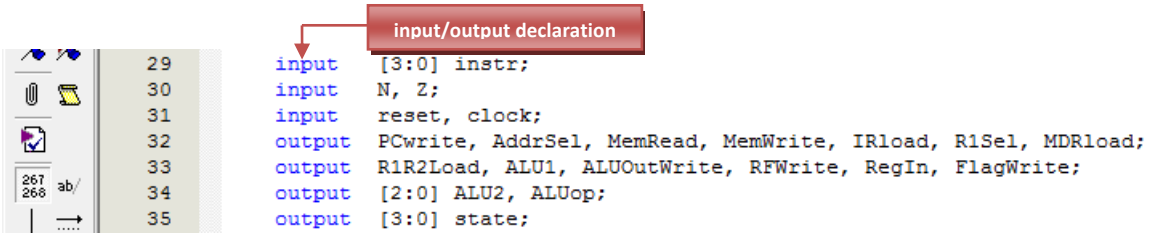


Figure 1.2 – declaration of inputs and outputs

If any output requires its value to be stored, or requires to be used in an *always* block, then a '*reg'* declaration is required. See Figure 1.3 and 1.4.

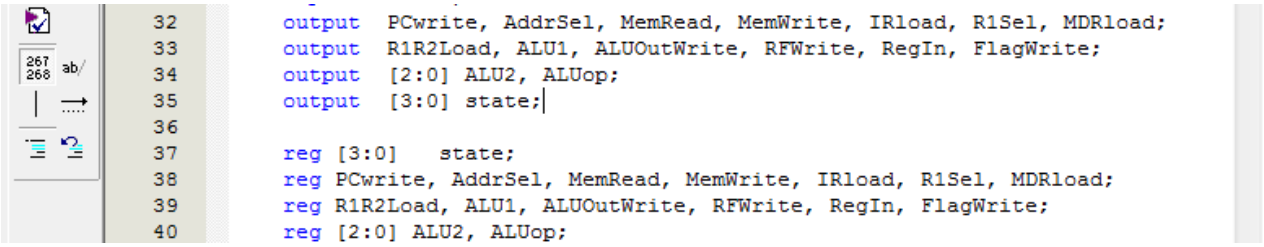
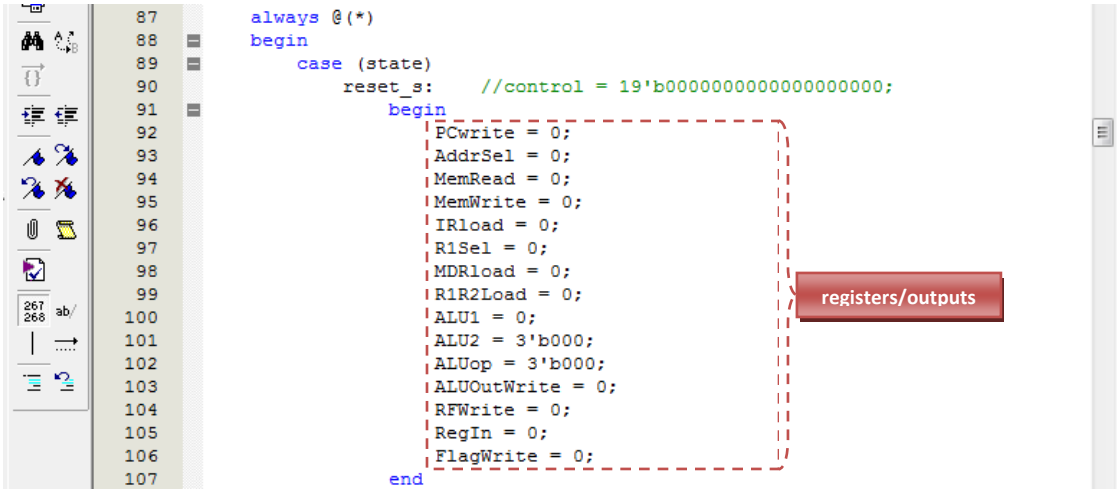


Figure 1.3 – outputs and registers declarations




```
87   always @(*)
88   begin
89       case (state)
90           reset_s:    //control = 19'b0000000000000000000;
91               begin
92                   PCwrite = 0;
93                   AddrSel = 0;
94                   MemRead = 0;
95                   MemWrite = 0;
96                   IRload = 0;
97                   R1Sel = 0;
98                   MDRload = 0;
99                   R1R2Load = 0;
100                  ALU1 = 0;
101                  ALU2 = 3'b000;
102                  ALUop = 3'b000;
103                  ALUOutWrite = 0;
104                  RFWrite = 0;
105                  RegIn = 0;
106                  FlagWrite = 0;
107              end
```

Figure 1.4 – outputs which require their values to be stored

Modifying the Control

The Verilog implementation of the control is called '*FSM.v*'. If new signals are to be introduced, refer to "Review of Verilog". If new states are required, add the new states to the parameter declarations. Each state parameter contains the state name and a numerical value. Any numerical value can be assigned to a state, as long as it is unique. The width of the parameter values may have to be increased from the default size of 4 bits.

Before Declaration

```
44   parameter [3:0] reset_s = 0, c1 = 1, c2 = 2,
45                   c3_asn = 3, c4_asnsh = 4, c3_shift = 5,
46                   c3_ori = 6, c4_ori = 7, c5_ori = 8,
47                   c3_load = 9, c4_load = 10, c3_store = 11,
48                   c3_bpz = 12, c3_bz = 13, c3_bnz = 14;
```

After Declaration

```
44   parameter [3:0] reset_s = 0, c1 = 1, c2 = 2,
45                   c3_asn = 3, c4_asnsh = 4, c3_shift = 5,
46                   c3_ori = 6, c4_ori = 7, c5_ori = 8,
47                   c3_load = 9, c4_load = 10, c3_store = 11,
48                   c3_bpz = 12, c3_bz = 13, c3_bnz = 14;
49                   c5_load = 15;
```

new state

Figure 2.1 – declaration of new load state

Once a new state has been declared, the state transition *case* block must be modified. The name of each new state needs to be included as a new case of the state transition *case* block. See Figure 2.2.

```
53  always @(posedge clock or posedge reset)
54  begin
55      if (reset) state = reset_s;
56      else
57      begin
58          case(state)
59              reset_s:    state = c1;         // reset state
60              c1:         state = c2;         // cycle 1
61              c2:         begin               // cycle 2
62                              if(instr == 4'b0100 | instr == 4'b0110 | instr == 4'b1000) state = c3_asn;
63                              else if( instr[2:0] == 3'b011 ) state = c3_shift;
64                              else if( instr[2:0] == 3'b111 ) state = c3_ori;
65                              else if( instr == 4'b0000 ) state = c3_load;
66                              else if( instr == 4'b0010 ) state = c3_store;
67                              else if( instr == 4'b1101 ) state = c3_bpz;
68                              else if( instr == 4'b0101 ) state = c3_bz;
69                              else if( instr == 4'b1001 ) state = c3_bnz;
70                              else state = 0;
71                          end
72              c3_asn:     state = c4_asnsh;   // cycle 3: ADD SUB NAND
73              c4_asnsh:   state = c1;         // cycle 4: ADD SUB NAND/SHIFT
74              c3_shift:   state = c4_asnsh;   // cycle 3: SHIFT
75              c3_ori:     state = c4_ori;     // cycle 3: ORI
76              c4_ori:     state = c5_ori;     // cycle 4: ORI
77              c5_ori:     state = c1;         // cycle 5: ORI
78              c3_load:    state = c4_load;    // cycle 3: LOAD
79              c4_load:    state = c5_load;    // cycle 4: LOAD (modified)
80              c5_load:    state = c1;         // cycle 5: LOAD (modified)
81              c3_store:   state = c1;         // cycle 3: STORE
82              c3_bpz:     state = c1;         // cycle 3: BPZ
83              c3_bz:      state = c1;         // cycle 3: BZ
84              c3_bnz:     state = c1;         // cycle 3: BNZ
85          endcase
86      end
```

state transition case block

modified state transition

Figure 2.2 – modified state transition case block with new load state introduced

After the new state has been added and the state transition case block has been modified, the control signals *case* block (in the level-sensitive *always* block) can now be modified. See Figure 2.3.

```
92      case (state)
93          reset_s:    //control = 19'b0000000000000000000;
94              begin
95                  PCwrite = 0;
96                  AddrSel = 0;
97                  MemRead = 0;
98                  MemWrite = 0;
99                  IRload = 0;
100                 R1Sel = 0;
101                 MDRload = 0;
102                 R1R2Load = 0;
103                 ALU1 = 0;
104                 ALU2 = 3'b000;
105                 ALUop = 3'b000;
106                 ALUOutWrite = 0;
107                 RFWrite = 0;
108                 RegIn = 0;
109                 FlagWrite = 0;
110             end

                                        ...

404     default:    //control = 19'b0000000000000000000;
405         begin
406             PCwrite = 0;
407             AddrSel = 0;
408             MemRead = 0;
409             MemWrite = 0;
410             IRload = 0;
411             R1Sel = 0;
412             MDRload = 0;
413             R1R2Load = 0;
414             ALU1 = 0;
415             ALU2 = 3'b000;
416             ALUop = 3'b000;
417             ALUOutWrite = 0;
418             RFWrite = 0;
419             RegIn = 0;
420             FlagWrite = 0;
421         end
422 endcase
```

control signals case block

Figure 2.3 – control signals case block with output control signals

When a new state is added, the control signals for that state need to be added in the control signals case block. When making modifications to existing states, changes can be made directly to each control signal. See Figure 2.4.

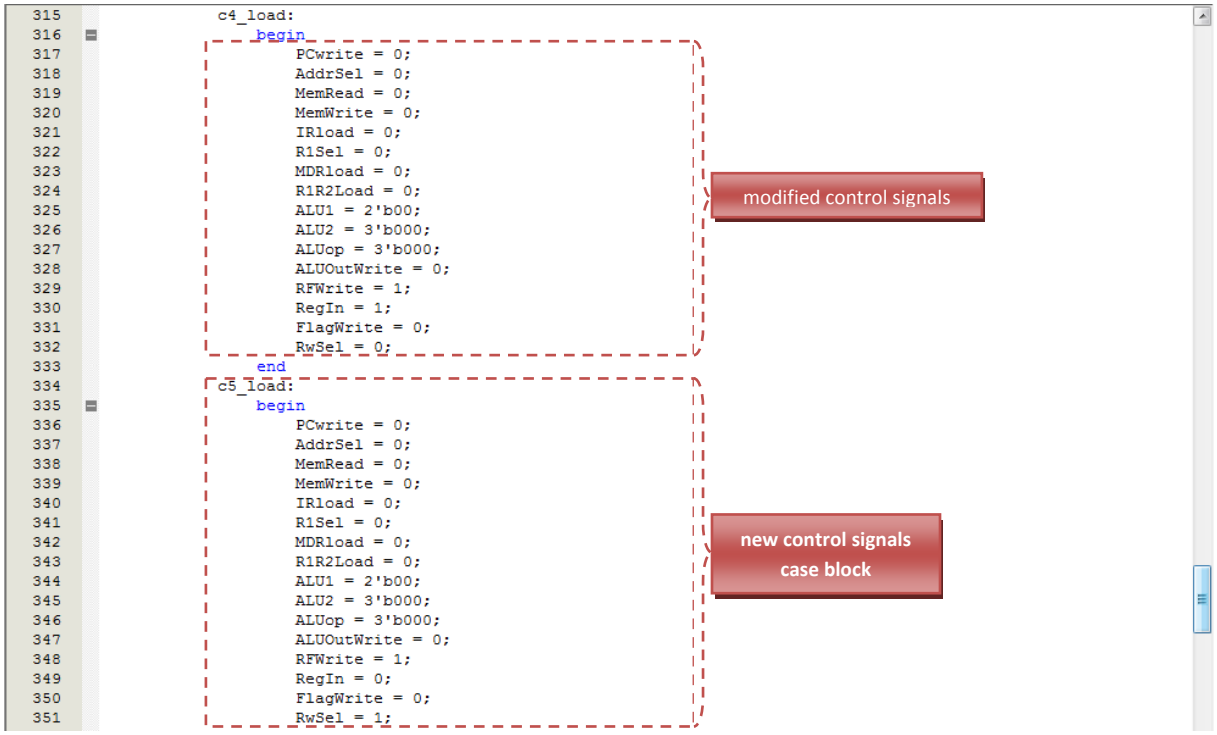


Figure 2.4 – modification of control signals