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.

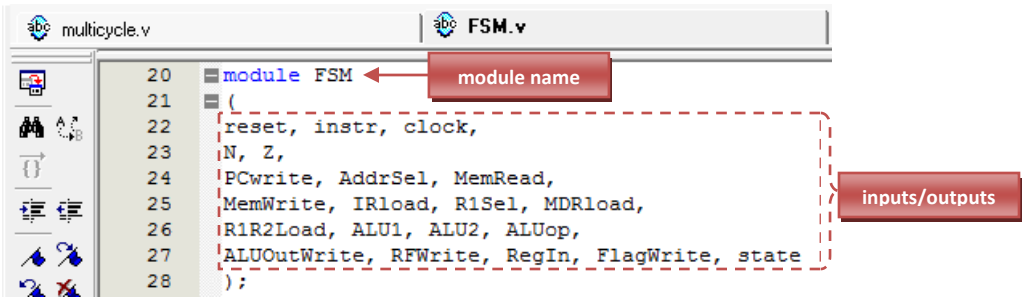


Figure 1.1 – module declaration with the module name and its inputs and outputs

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).

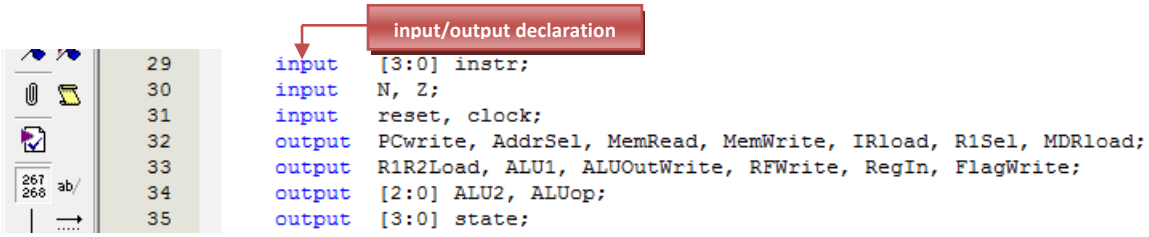


Figure 1.2 – declaration of inputs and outputs

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.

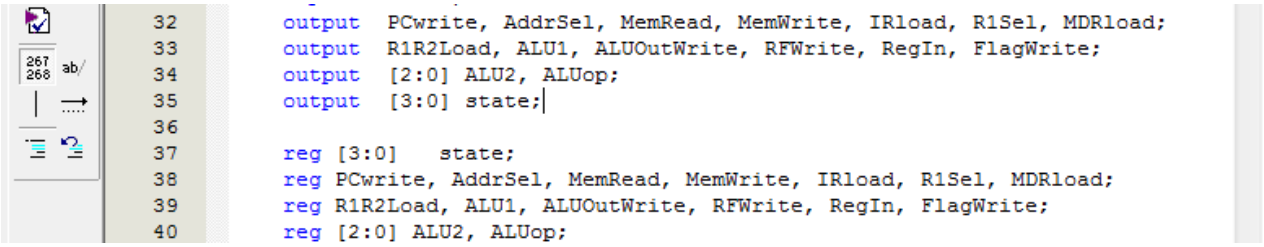
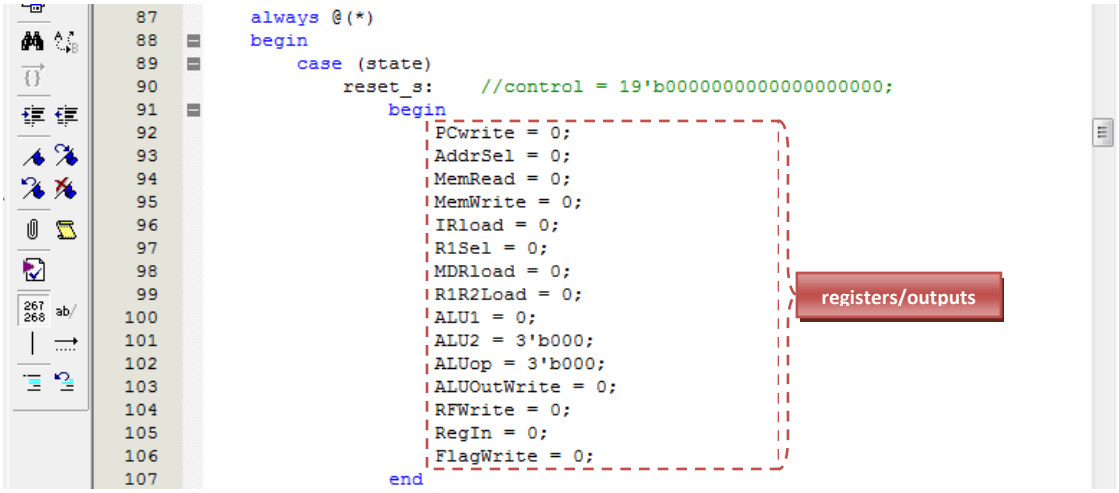


Figure 1.3 – outputs and registers declarations




```
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.

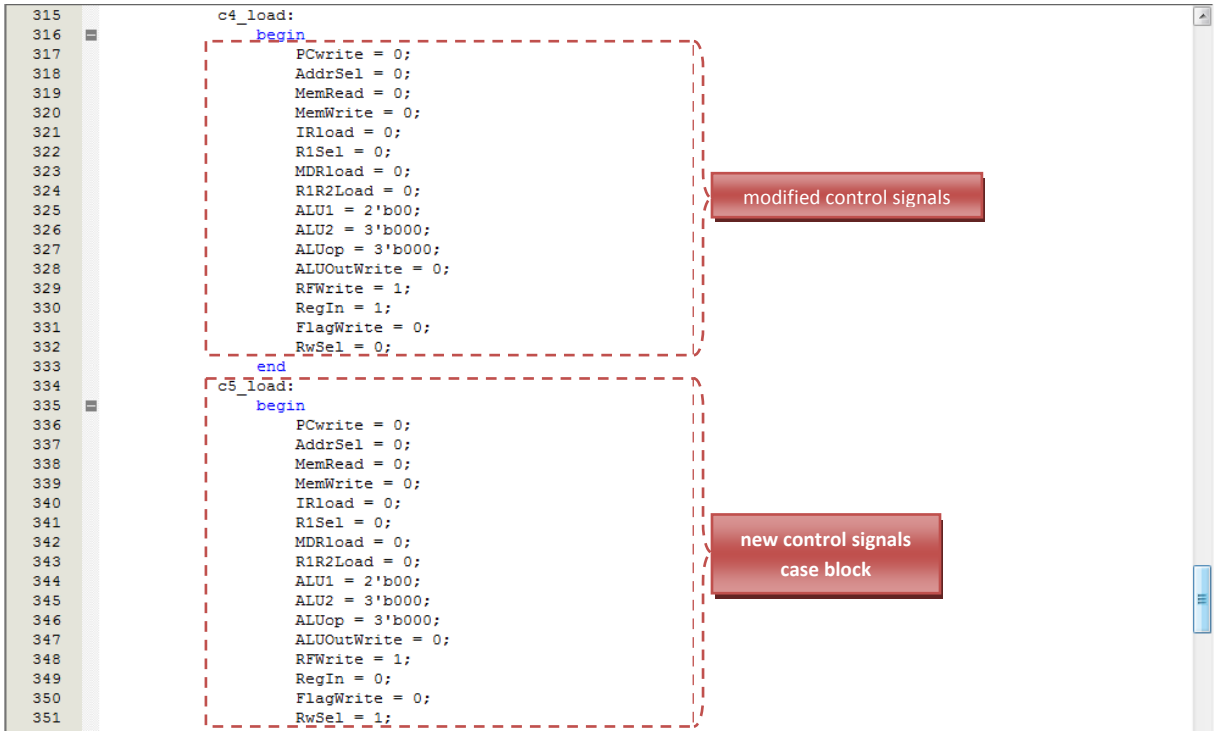


Figure 2.4 – modification of control signals

Modifying datapath

- **Verilog**

  If new simple components, such as a MUX or a register, which are not listed in the design implementation, are required, they can be generated using the *MegaWizard. MegaWizard* allows the creation of customized hardware components, such as MUXes, registers, or memory.

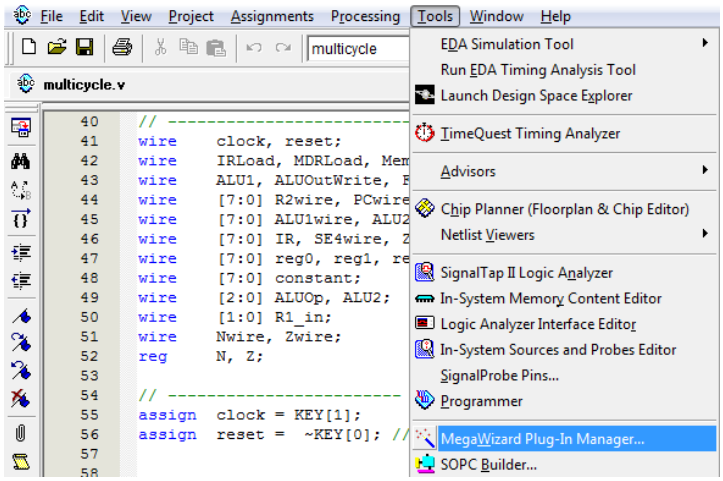
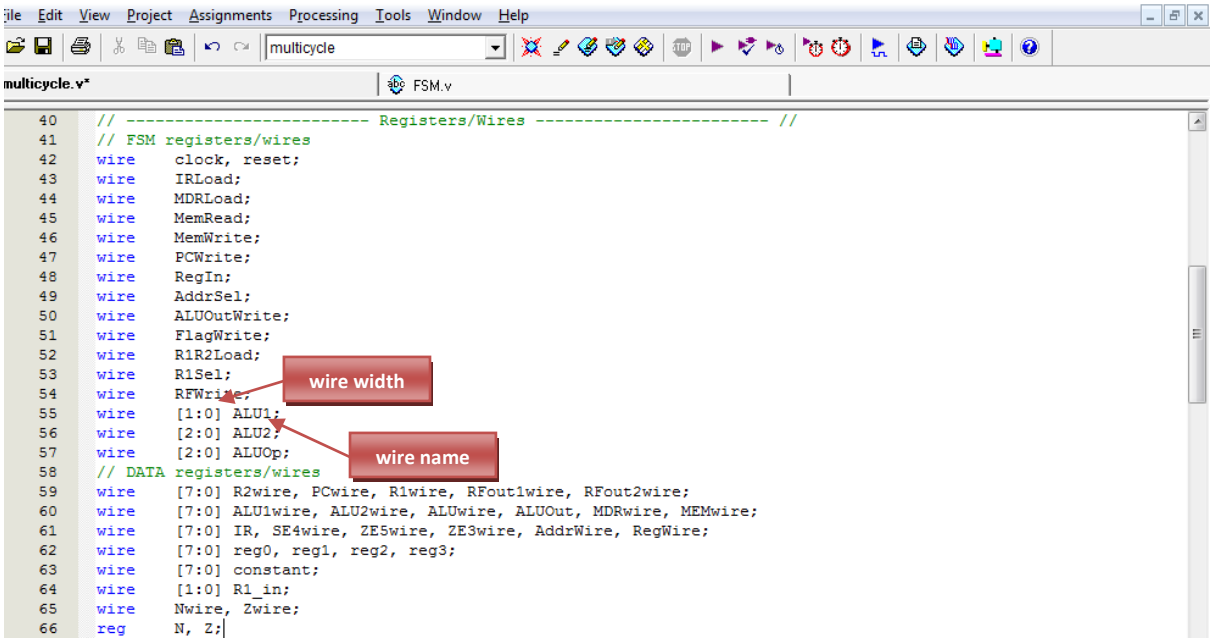


Figure 3.1 – "MegaWizard Plug-In Manager…"

To run “MegaWizard Plug-In Manager”, select *Tools > MegaWizard Plug-In Manager… (*See Figure 3.1).

Once all the components have been created, they must be instantiated and wired. If new wires are required, new wire declarations must be added. However, if only the width of the wire changes, one can modify or add the width size as required. See Figure 3.2.




```
40  // ------------------------ Registers/Wires ------------------------ //
41  // FSM registers/wires
42  wire    clock, reset;
43  wire    IRLoad;
44  wire    MDRLoad;
45  wire    MemRead;
46  wire    MemWrite;
47  wire    PCWrite;
48  wire    RegIn;
49  wire    AddrSel;
50  wire    ALUOutWrite;
51  wire    FlagWrite;
52  wire    R1R2Load;
53  wire    R1Sel;
54  wire    RFWrite;
55  wire    [1:0] ALU1;
56  wire    [2:0] ALU2;
57  wire    [2:0] ALUOp;
58  // DATA registers/wires
59  wire    [7:0] R2wire, PCwire, R1wire, RFout1wire, RFout2wire;
60  wire    [7:0] ALU1wire, ALU2wire, ALUwire, ALUOut, MDRwire, MEMwire;
61  wire    [7:0] IR, SE4wire, ZE5wire, ZE3wire, AddrWire, RegWire;
62  wire    [7:0] reg0, reg1, reg2, reg3;
63  wire    [7:0] constant;
64  wire    [1:0] R1_in;
65  wire    Nwire, Zwire;
66  reg     N, Z;
```

Figure 3.2 – wire declarations with different wire sizes (widths)

Once all the components are created and wires are declared, they can be wired directly as shown in Figure 3.3.

```
83  FSM     Control(
84      .reset(reset),.clock(clock),.N(N),.Z(Z),.instr(IR[3:0]),
85      .PCwrite(PCWrite),.AddrSel(AddrSel),.MemRead(MemRead),.MemWrite(MemWrite),
86      .IRload(IRLoad),.R1Sel(R1Sel),.MDRload(MDRLoad),.R1R2Load(R1R2Load),
87      .ALU1(ALU1),.ALUOutWrite(ALUOutWrite),.RFWrite(RFWrite),.RegIn(RegIn),
88      .FlagWrite(FlagWrite),.ALU2(ALU2),.ALUop(ALUOp)
89  );
```

Figure 3.3 – Wiring components

- **Schematic**

In schematic, to create a new component, double click at the background of the design. Once double click, a ‘Symbol’ screen would pop-up. If the component does not exist under ‘Project’, new components can be created through the library or *MegaWizard* under another sub-folder. See Figure 3.4.

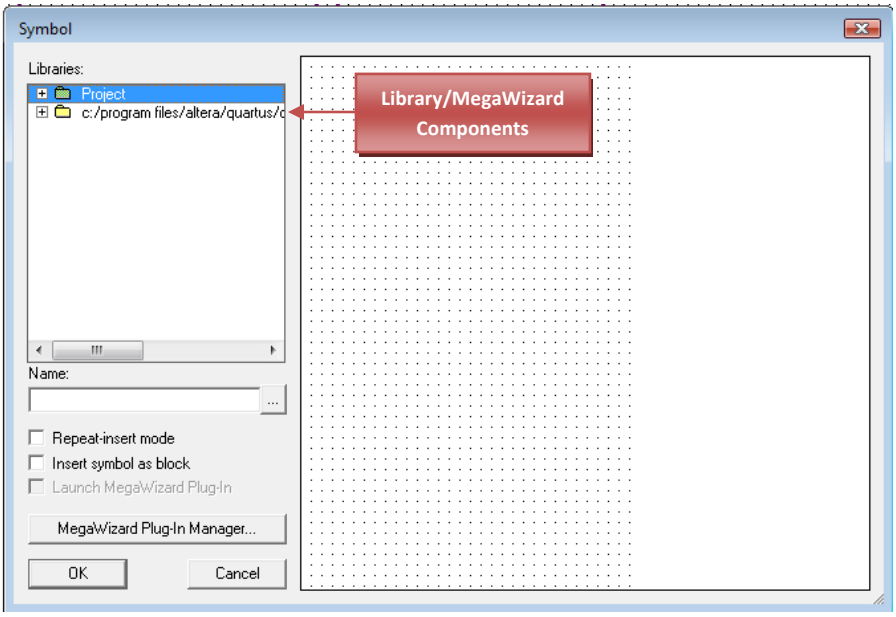


Figure 3.4 – 'Symbol' dialog box allows creation of symbols/components

After all components are created and placed on the background, they are to be wired. Wires are located on the left of the Quartus window screen. See Figure 3.5.

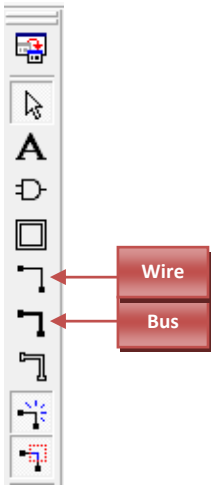


Figure 3.5 – Wiring tools

Wire each component together by dragging the wires from one component input to another component's output or vice-versa.