Project #1: LC3 Microcontroller

1. Introduction

The main purpose of this project is to let you start dealing with more complex designs, and become familiar with some of the elements used within a CPU.

2. Learning Objectives

- Complete a design involving separate control and datapath with multiple modules
- Complete a design that includes most of the elements to be used in the CPU

3. Project Report

You are expected to turn in a report after the end of this project. Follow the project report format given on the *Laboratories* page on the course web-site. Be sure to include all items listed in that report format for full credit.

4. Wolfware Submission

You also need to submit your Verilog code electronically through Wolfware as **proj1.v**. This file should contain a module called SimpleLC3, as described below. It may use the '*include* directive to include other files, if you wish, but they must also be submitted with Wolfware. Your code must successfully execute with the test-bench provided on the course web-site (called **proj1test.v**) and give the correct expected output as listed in the test-bench. The memory file (**proj1.dat**) is also given, along with the expected output of the instruction set simulator (**proj1.out**). This program is taken from the example given in class. In addition, a second program will be used to test your code that will not be provided.

5. Lab Design: simplified LC3 microcontroller

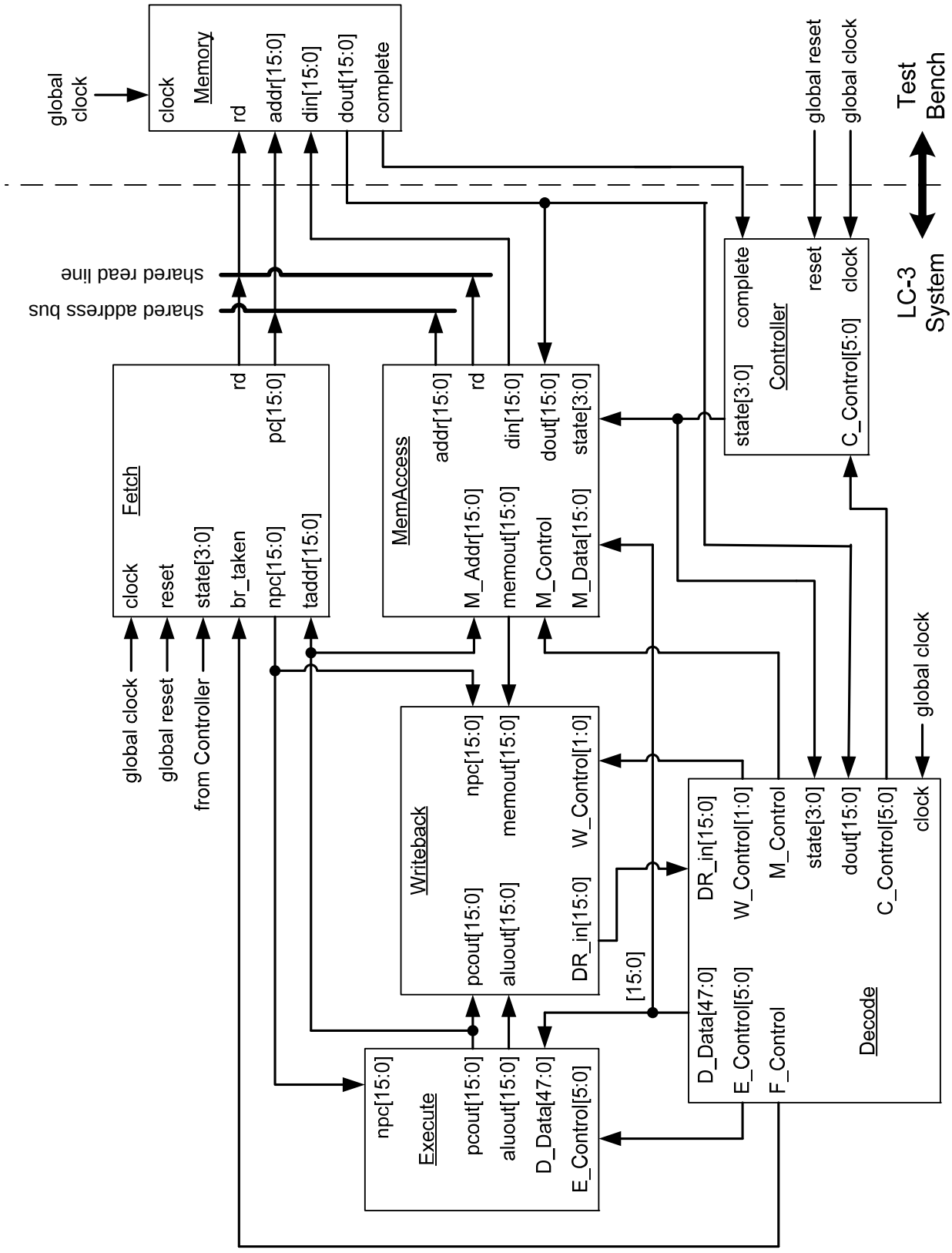
The microcontroller designed in this lab assignment is a simplified version of the original LC3 microcontroller. Specifically, four simplifications are considered as follows:

1) A smaller instruction set: the ISA you need to implement does NOT contain the control instructions RTI and TRAP. All other instructions must be implemented.
2) No off-chip memory: The instructions of the program are assumed to be in the cache.
3) The programs consist of valid instructions ONLY, i.e., you do not have to perform error checking to detect bad instructions
4) No overflow detection is required.

5.1 Top-level module:

```
module SimpleLC3(clock, reset, addr, din, dout, rd, complete);
   input clock;               // Global system clock
   input reset;               // Global system reset
   output [15:0] addr, din;   // Address and Data-In lines for Memory
   input [15:0] dout;         // Data-Out lines from Memory
   output rd;                 // Memroy signal to indicate read or write
   input complete;            // Signal to indicate completion of read/write
```

SimpleLC3 Schematic



As shown in the schematic on the previous page, the top-level module is instantiated in the test-bench along with the memory.

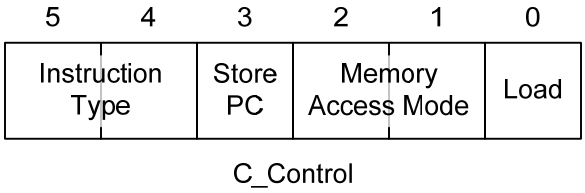
Special Signals:
The **SimpleLC3** module should be connected exactly as shown in the schematic. Note that there is a shared read-line and a shared address-bus for the memory, which means that these signals will be driven from two sources. In addition, note that **VSR2** field (which in this schematic is the least significant 16 bits of the the **D_Data** signal to the **Execute** block) goes into the **MemAccess** block as the **M_Data** signal. All other signals are simple inputs and outputs.

The specifications for the **Controller**, **Fetch** and **Execute** blocks were given in Homeworks #5, and #7. The specifications for the remaining 3 blocks are given here.

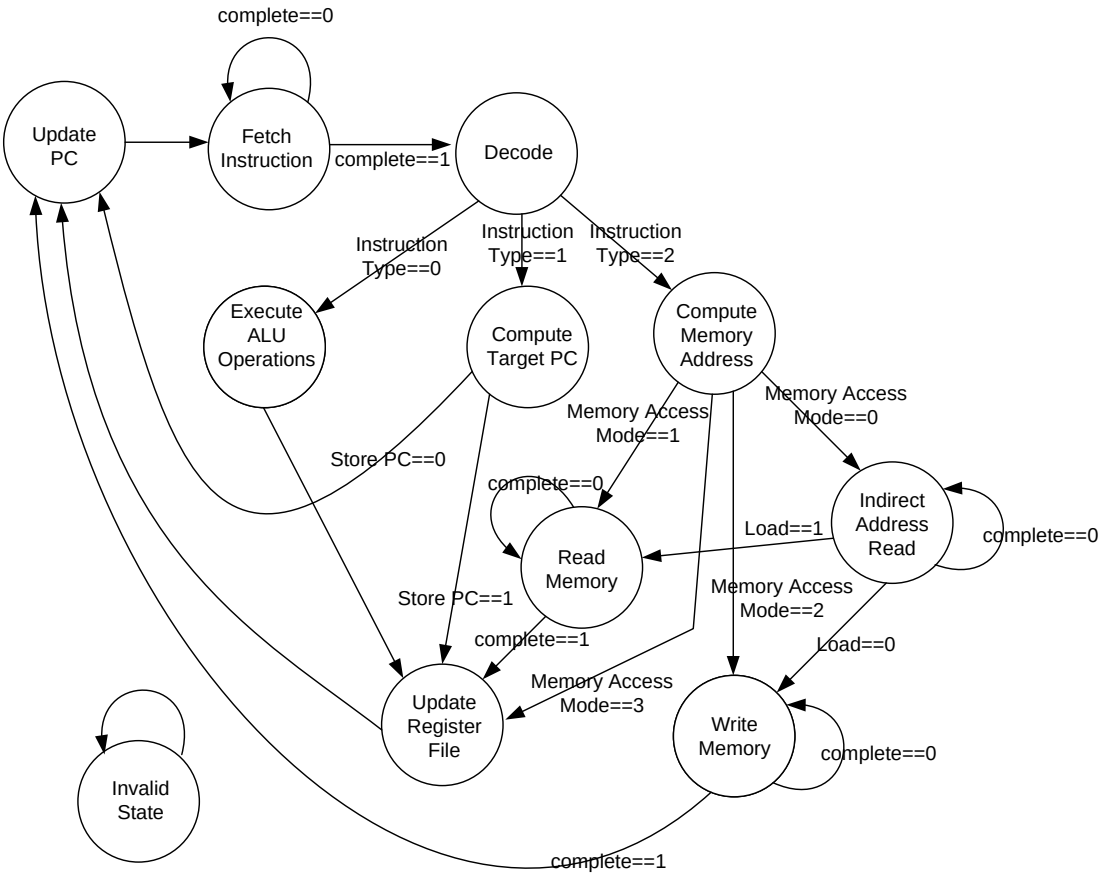
5.2 Controller

```
module Controller(clock, reset, state, C_Control, complete);
      input clock, reset;           // system clock and reset
      output [3:0] state;           // system state
      input [6:0] C_Control;        // control from Decode
      input complete;               // complete from Memory
```

The Controller module is a finite state machine that controls the dataflow and therefore the execution of all the instructions in the microcontroller. The state transition diagram sketch is given below, in which the vertices represents states with the corresponding operations described inside. The transitions are denoted by the edges. The condition of each transition is determined by the current state and/or input signal **C_Control** generated by the decoder module. The C_Control can be broken down further into 4 fields as follows.



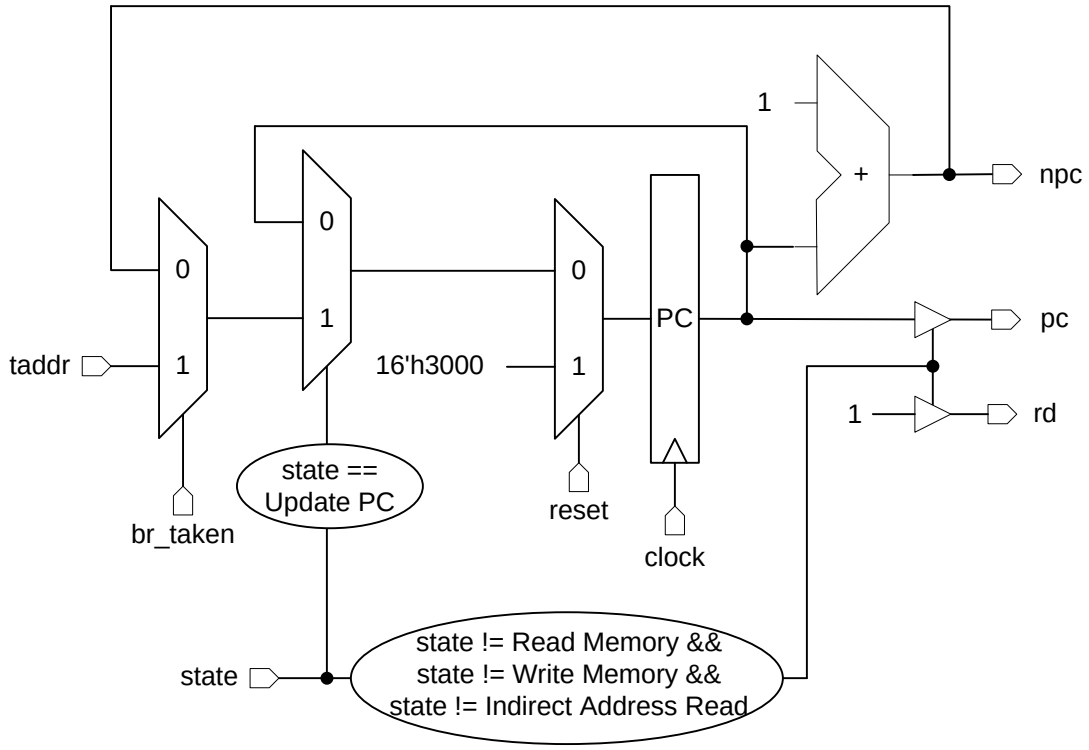
The self-pointing edges are used for Project 2 to cope with the memory latency. Such looping transitions only occur when the **complete** signal is zero, which never happens in Project 1. State transitions occur and only occur at positive edges of the clock signal. When the **reset** signal is high, the next-state should be the “Fetch Instruction” state.



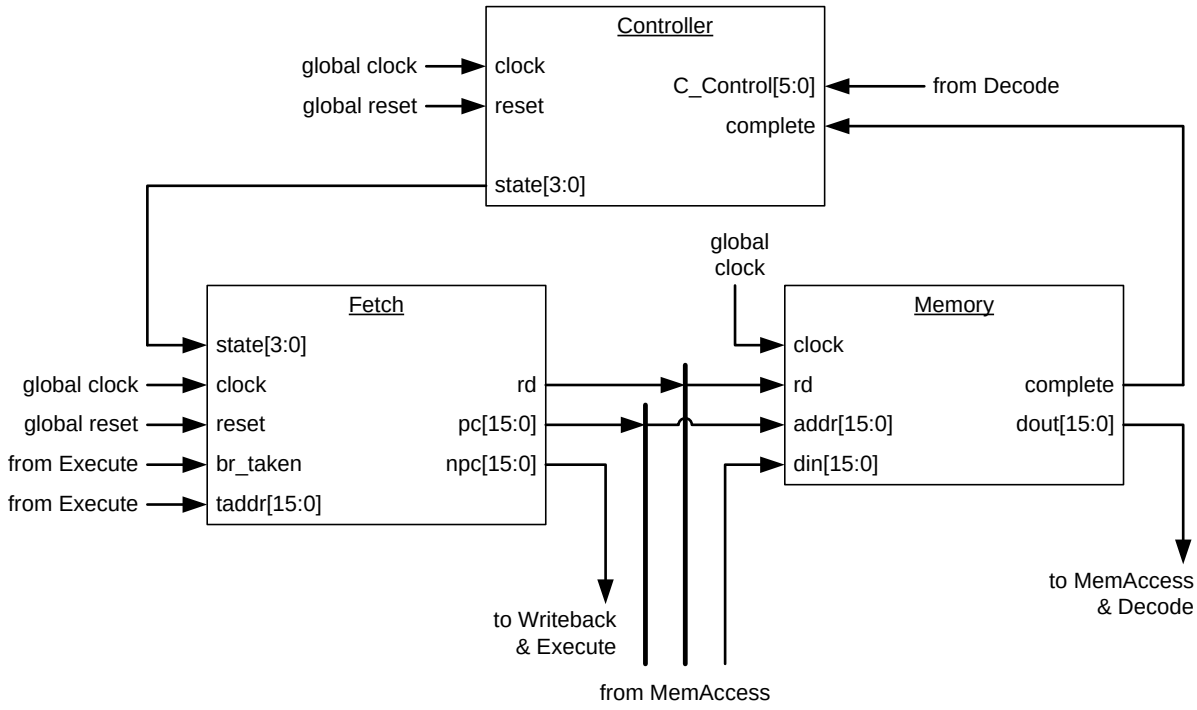
5.3 Fetch

```
module Fetch(clock, reset, state, pc, npc, rd, taddr, br_taken);
  input clock;                // system clock
  input reset;                // system reset
  input br_taken;             // signal from decoder, 1 means branch taken
  input [15:0] taddr;         // target address of control instructions
  input [3:0] state;          // system state from controller
  output [15:0] pc, npc;      // current PC and next PC, i.e., pc+1
  output rd;                  // memory read control signal
```

Fetch module is used to generate the program counter, which contains the address of the instruction to be fetched. The PC should be updated on the rising edge of the clock. Also, the PC should be updated only when the system is in the "Update PC" state, as determined by the Controller block. The signal **rd** should be high-impedance during the "Read Memory", "Write Memory", and "Indirect Address Read" states, because the MemAccess block will drive the shared memory bus during these cycles. In all other states, this signal should be high. **pc** is the memory address and should be high-impedance at the same times that **rd** is high-impedance. The first program instruction is located at the address 16'h3000. Therefore, **pc** should be set to 16'h3000 when **reset** is high. The block diagram of Fetch module is shown below.



The relation among Fetch, Controller, and off-chip memory module is shown below.

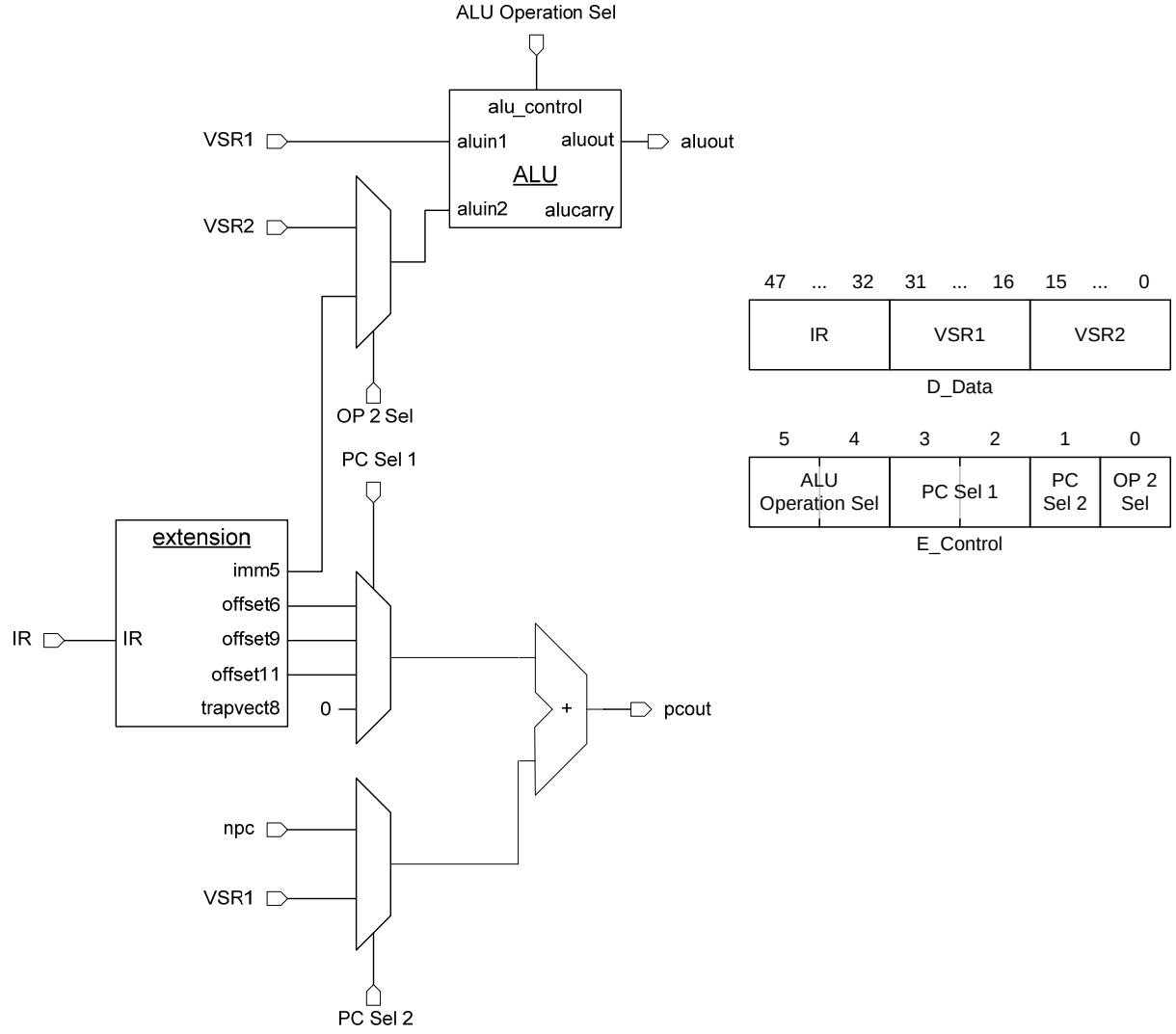


5.4 Execute

```
module Execute(E_control, D_data, aluout, pcout, npc);

  input [5:0] E_control;          // control signals from Decode
  input [47:0] D_data;            // data from Decode
  output [15:0] aluout;           // output of ALU
  output [15:0] pcout;            // output of the address computation adder
  input [15:0] npc;               // next PC from Fetch
```

Execute module performs the arithmetic and logical instructions, target PC computation, and memory address computation. The **E_Control** input is an aggregate of the **ALU Operation Sel**, **OP 2 Sel**, **PC Sel 1**, and **PC Sel 2**. The **D_Data** input is an aggregate of the **IR**, **VSR1** and **VSR2** signals. The block diagram is given below with the **ALU** and **extension** modules in homework 3. Note that overflow checking is not being done, so the **alucarry** output of the ALU is ignored.



5.5 MemAccess

```
module MemAccess(state, M_Control, M_Data, M_Addr, memout, addr, din, dout, rd);
  input [3:0] state;       // System State from Controller
  input M_Control;         // Control Signal to indicate address from dout
  input [15:0] M_Data;     // Data for store operations
  input [15:0] M_Addr;     // Address for load/store operations
  output [15:0] addr;      // Address lines to memory
  output [15:0] din;       // Data-in lines to memory
  output rd;               // Memory signal to indicate read or write
  input [15:0] dout;       // Data-Out lines from Memory
  output [15:0] memout;    // Data read from Memory to write in register file
```

The **MemAccess** block is the master of the shared memory bus during the *Read Memory*, *Write Memory*, and *Read Indirect Address* states. It should setup the memory bus lines as follows:

- Read Memory – **rd** should be 1 and **din** doesn't matter. **addr** should be set to either **M_Addr** or **dout**, depending on **M_Control**. **addr** should be set to **dout** in this state only if the opcode shows an LDI operation.
- Write Memory – **rd** should be 0 and **din** should be **M_Data**. **addr** should be set to either **M_Addr** or **dout**, depending on **M_Control**. **addr** should be set to **dout** in this state only if the opcode shows an STI operation.
- Read Indirect Address **- rd** should be 1 and **din** doesn't matter. **addr** should be set to **M_Addr**.

The **memout** signal should always pass the value of **dout** through to the **Writeback** block.

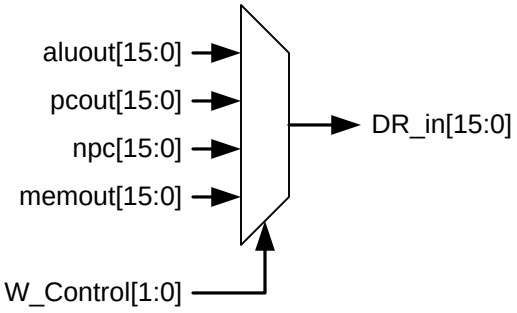
5.6 Writeback

```
module Writeback(W_Control, aluout, memout, pcout, npc, DR_in);
  input [15:0] aluout, memout, pcout, npc;  // Possible data to store
  input [1:0] W_Control;   // Control signal to choose what will be written
  output [15:0] DR_in;     // Data that will be stored in the register-file
```

The **Writeback** block should set the **DR_in** lines to the value to be written into the register-file. This value is selected from the following four choices:

- **aluout** – The output of the ALU in the **Execute** block
- **pcout** – The computed memory address output of the **Execute** block
- **npc** – The next value of the program counter from the **Fetch** block
- **memout** – The value read from memory, from the **MemAccess** block

The **W_Control** signal will be used to select between these possibilities. The schematic is shown below.

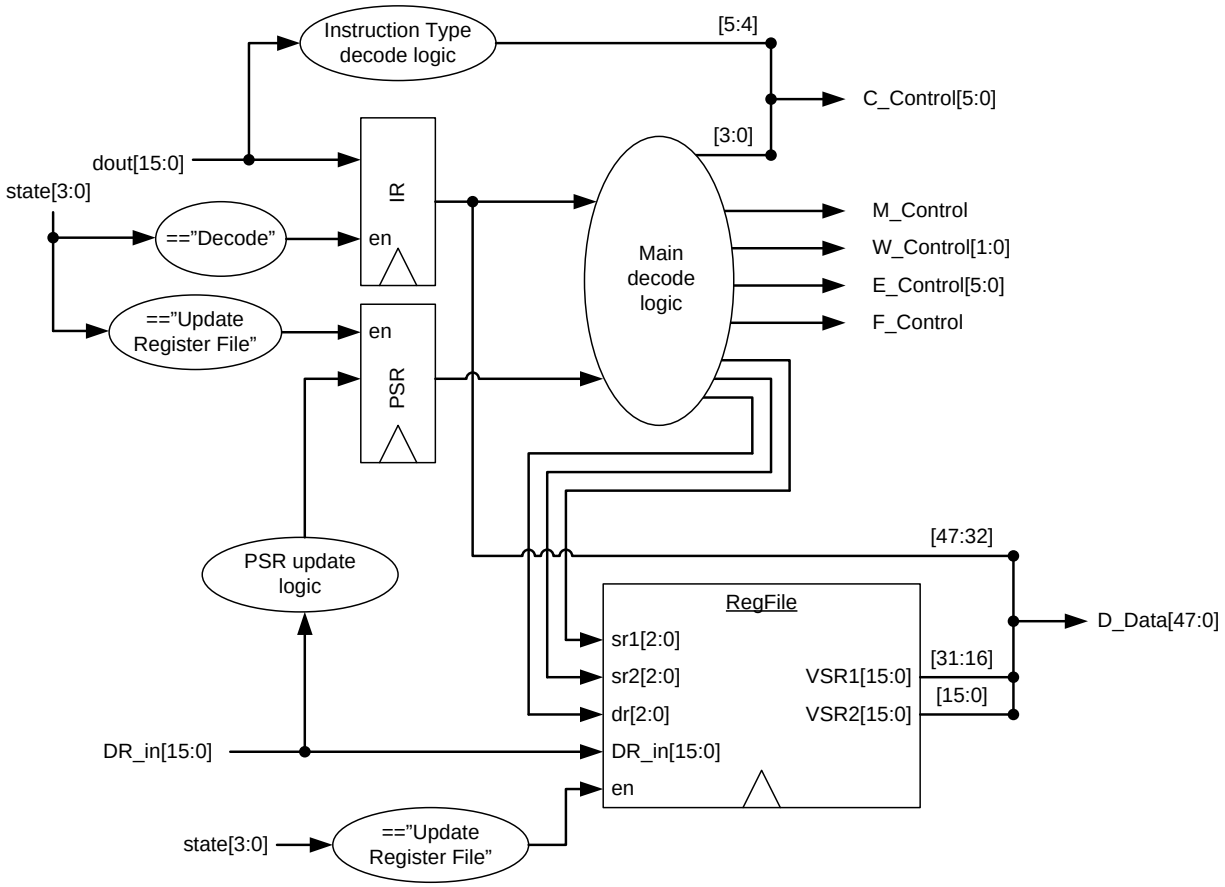


5.7 Decode

```
module Decode(clock, state, dout, C_Control, E_Control,
            M_Control, W_Control, F_Control, D_Data, DR_in);
  input clock;              // Global system clock
  input [3:0] state;        // System state from Controller
  input [15:0] dout;        // Data-out lines from Memory
  input [15:0] DR_in;       // Data to be written to Register-File
  output M_Control;         // MemAccess control line
  output [1:0] W_Control;   // Writeback control lines
  output [5:0] C_Control;   // Controller control lines
  output [5:0] E_Control;   // Execute control lines
  output [47:0] D_Data;     // Data for Execute and MemAccess blocks
  output F_Control;         // Fetch control line
```

The **Decode** block contains the logic illustrated in the schematic below. It contains an instruction register (IR) that stores the current instruction during the *Decode* state. It contains a program status register (PSR) that stores the status of the last value written to the register file (positive, negative, or zero) and is update only on the *Update Register File* state. Lastly, it contains a register file that can read two locations on one cycle and write to one location in the same cycle. However, the register file writes only during the *Update Register File* state.

Based on the contents of IR and PSR, the decode block generates all of the control signals for the other blocks (C_Control, M_Control, W_Control, E_Control, and F_Control) as well as the source and destination addresses in the register file (sr1, sr2, and dr). Note, however, that the "instruction type" field of the C_Control signal must be valid during the *Decode* state and will therefore not be valid if this field is computed from the contents of IR. Therefore, the "instruction type" field is computed from the memory output, which makes it valid during the *Decode* state (but not necessarily the states after *Decode*).



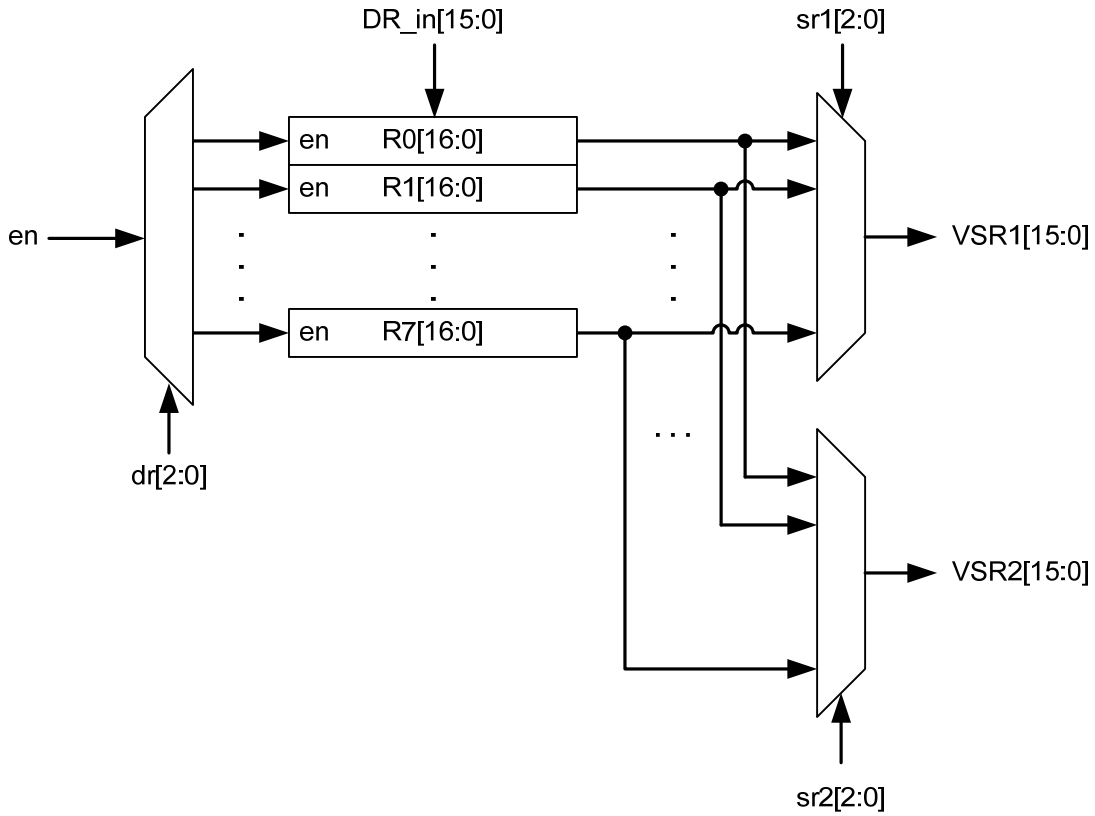
When writing the Verilog code for the main decode combinational logic, it is helpful to first create a table that has the values needed for each control signal in order to set up the micro-controller to implement each operation. Empty tables are provided below for your convenience.

		C_Control				
Operation	Mode	Instr. Type	Store PC	Mem. Access Mode	load	M_Control
ADD	0					
	1					
AND	0					
	1					
NOT						
BR						
JMP/RET						
JSR						
JSRR						
LD						
LDR						
LDI						
LEA						
ST						
STR						
STI						

Operation	Mode	E_Control: ALU Op Sel	PC Sel 1	PC Sel 2	Op 2 Sel	W_Control	F_Control
ADD	0						
	1						
AND	0						
	1						
NOT							
BR							
JMP/RET							
JSR							
JSRR							
LD							
LDR							
LDI							
LEA							
ST							
STR							
STI							

Operation	Mode	DR	SR1	SR2
ADD	0			
	1			
AND	0			
	1			
NOT				
BR				
JMP/RET				
JSR				
JSRR				
LD				
LDR				
LDI				
LEA				
ST				
STR				
STI				

The **RegFile** block inside the **Decode** block can be conceptually thought of as the schematic below. In the schematic, the outputs of eight 16-bit registers fan-out to two 8-to-1 MUXes, which are used to determine the **VSR1** and **VSR2** signals, depending on the **sr1** and **sr2** select lines. The **DR_in** input fans-out to the data-inputs of all eight registers. Each register has an enable input that determines if it will load the input value, and these enable inputs are connected to a 1-to-8 decoder that passes the master enable signal to one register, depending on the **dr** input.



You may want to implement the **RegFile** with a memory, rather than eight individual registers. The code will be much simpler and easy to understand. However, the waveform capture formats typically don't store the values in memories. If you want to view the contents of your register file in a waveform viewer, you may want to have a set of eight assign statements, such as the following:

```
assign R0=mem[0];
assign R1=mem[1];
. . .
```

This approach will not affect your synthesis results. You should still get 16×8=128 flip-flops when synthesizing.