

## EE 231L Lab 4

### Design and Implementation of State Machines Design of a Computer Control Unit

In this lab you will design a control system for a computer. You will design it as a state machine. Be sure to read the handout Using Verilog to Design State Machines. There are also a few other blocks you will need to implement your computer – two multiplexers, a decoder, and a tri-state buffer. In Part 1 of this lab, you will design these other blocks. In Part 2, you will design the computer control unit.

#### Part 1. Other Combinational Circuits.

##### 1. Multiplexer.

In the diagram of the final computer there is an element labeled MUX. This is a multiplexer. The MEM\_SEL lines are the selection lines of the multiplexer. Depending on the state of the MEM\_SEL lines, the MUX (multiplexer) will choose to output one of four possible signals: the value in either PROG\_ADDR, PC, MAR or X.

Write a Verilog program to implement the MUX. (Remember, PROG\_ADDR, PC, and MAR are each 8 bits wide).

##### 2. Decoder.

Directly to the right of the MUX is another computer element. This is the Decoder (DCD). The DCD determines if the memory address output by the MUX is equal to 0xFF.

- If the address equals 0xFF then the ADDR\_FF line should be brought low. This will allow either the external input or output to be enabled depending on the state of the M\_W (memory write) and M\_R (memory read) lines.
- If the output of the mux is not equal to 0xFF, then the ADDR\_NOTFF line should be brought low. When the ADDR\_NOTFF line is low, the memory is selected and can be read from or written to (depending on the state of M\_W and M\_R). Write a Verilog Program to implement the decoder.

##### 3. Tri-State Buffer.

You will need two tri-state buffers in the final computer. Verilog has an active-hi tri-state buffer (TRI). Here is a program which implements an 8-bit active-low tri-state buffer:

```
module trin (Y, E, F);
    parameter n = 8;
    input [n-1:0] Y;
    input E;
    output wire [n-1:0] F;

    assign F = E ? Y : 'bz;

endmodule
```

## Part 2. Design of Computer Control Unit.

The data-processing functions of the computer are divided into simple units called instructions. A computer program is just a collection of computer instructions. The instruction set of a computer are the basic operations that the computer can perform. The instruction set of our computer is shown in Figure 1.

In this lab you will design the computer control unit. The control unit is a finite state machine. Its inputs are the instruction register and the carry, as well as a clock pulse and RESET. The control unit's outputs are the control signals that direct the operation of the rest of the computer. The control unit can be in one of four states: RESET, C1, C2, C3:

RESET is the Reset state. The computer gets into this state when the Reset input is low, and stays in this state until the Reset input goes high.

C1 is the Fetch Cycle. The computer program is stored in memory. During the fetch cycle the next instruction is fetched from memory and loaded into the instruction register (INST).

C2 is the first of Execution Cycle. Once an instruction has been loaded into the INST, the control unit determines the required course of action to take based on the value of INST and the current state of the control unit.

C3 is the second Execution Cycle. Some instructions only require one execution cycle (C2) while others require two (C2 and C3).

The output of the control unit depends on both the present state and the input. (What type of state machine is this?)

<b>Mnemonic</b>	<b>Operation</b>
0 LDAA addr load ACCA from memory	Loads ACCA with the value in memory at address addr  C stays the same, Z changes
1 LDAA #num load ACCA with an immediate	Loads ACCA with num, the value in memory at the address immediately following the LDAA #num command C stays the same, Z changes
2 LDAA 0,X load ACCA indexed	Loads ACCA with the value in memory at the address in the X register. C stays the same, Z changes.
3 STAA addr store ACCA in memory	Stores the value in ACCA at the memory address addr C stays the same, Z changes
4 ADDA addr add ACCA and value in memory	Adds the value in memory location addr to the value in ACCA at saves the result in ACCA Z and C change
5 SUBA addr subtract value in memory from ACCA	Subtracts the value in memory location addr from the value in ACCA and saves the result in ACCA Z and C change
6 ANDA addr logical AND of ACCA and value in memory	Perform a logical AND of the value in memory location addr with the value in ACCA. Save result in ACCA C stays the same, Z changes.
7 ORAA addr logical OR of ACCA and value in memory	Perform a logical OR of the value in memroy location addr with the value in ACCA. Save result in ACCA C stays the same, Z changes.
8 CMPA addr compares ACCA to the value in addr	Compare ACCA to value in addr. This is done by subtracting the value in addr from ACCA. The C and Z bits are changed. ACCA does not change
9 LDX #num load X with an immediate	Loads X with num, the value in memory at the address immediately following the LDX #num command C stays the same, Z changes.
A INX increment X	Increment value in X C stays the same, Z changes.
B CPX #num compares X to the num	Compare X to num, the value in memory at the address immediately following the CPX #num command. The C and Z bits are changed. X does not change

<b>Mnemonic</b>	<b>Operation</b>
C COMA complement ACCA	Replace the value in ACCA with its one's complement C is set to 1, Z changes.
D INCA increment ACCA	Increment value in ACCA C stays the same, Z changes
E LSLA	Logical shift left of ACCA. C and Z change.
F LSRA	Logical shift right of ACCA. C and Z change.
10 ASRA	Arithmetic shift right of ACCA. C and Z change.
11 JMP addr jump	Jumps to the instruction stored in address addr (The value in PC is replaced with addr.) C and Z stay the same.
12 JCS addr jump if carry set	Jumps to the instruction stored in address addr if C = 1. If C is not set, continue with next instruction. C and Z stay the same.
13 JEQ addr jump if carry set	Jumps to the instruction stored in address addr if Z = 1. If Z is not set, continue with next instruction. C and Z stay the same.

**Figure 1. Instruction Set**

The outputs of the control unit are the control signals shown on the block diagram of the computer. Except for ALU\_CTL and MEM\_SEL, all of these signals are active low, so your Verilog program should have a DEFAULTS section in which those signals will be high by default. In your Verilog code you will activate the appropriate signals at the correct times to implement the instruction the control unit is executing.

During the FETCH cycle the control unit will fetch the next instruction from memory to determine what instruction it should execute. Thus, the FETCH cycle will be the same for all instructions, it will read the instruction from memory, and latch it into the INST register. To do this, READ, INST\_L and PC\_I should be low, and MEM\_SEL should be set to select the address from the program counter PC. With the control lines set up like this, the address to the memory will be from the PC — i.e., the address of the next instruction to execute, and the memory output enable line will be low (active). The memory will put the data at that address on its output lines, which are the input lines to the INST register. On the next clock edge, the data from memory will be latched into the INST register, and the PC will be incremented to the next memory address. What the control unit does next will depend on the data loaded into the INST register. Here are a couple of examples:

**Example 1:**

Consider the instruction LDAA addr where addr = 0xF5. We will further assume that the instruction is in memory address 0x00 and 0x01, and that the code for LDAA addr is 0x01.

PC	Memory Address	Memory Data
→	00	01
	01	F5
	02	Next instruction

**INST = ??**

**MAR = ??**

C1: During the Fetch Cycle the instruction register must be loaded with the instruction op code, 0x01. To do this the MUX must select the PC as the address source, memory address 0x00 must be read which causes its value to be placed on the DATA lines. The value on the DATA lines must be latched into the INST register, and the PC must be incremented. Thus during C1 you should have *PC\_I*, *INST\_L* and *READ* active, and *MEM\_SEL* set to *PC*. Now the situation is as below:

PC	Memory Address	Memory Data
	00	01
→	01	F5
	02	Next instruction

**INST = 01 (LDAA addr op code)**

**MAR = ??**

C2: During C2, you must read the memory address that the PC is pointing at. By reading address 0x01 the value 0xF5 is placed on the DATA line. Then 0xF5 needs to be stored in the MAR register. Finally the program counter should be incremented. Thus during C2 you should have *PC\_I*, *MAR\_L* and *READ* active, and *MEM\_SEL* set to *PC*. After these steps the situation should be as shown below:

PC	Memory Address	Memory Data
	00	01
	01	F5
→	02	Next instruction

**INST = 01 (LDAA addr op code)**

**MAR = F5**

C3: Now that MAR contains the value 0xF5, the multiplexer should select MAR as the source of the address. This address should then be read which causes the memory contents of address 0xF5 to be placed onto the DATA line. Then the ALU can load this value into ACCA. During C3 you should have *ACCA\_L* and *READ* active, *MEM\_SEL* set to *MAR*, and *ALU\_CTL* set to *LOAD*. When the control lines are set up like this, the value of 0xF5 will be on the address lines of the memory unit, and the data lines out of the memory will contain the data in address 0xF5. This data will be passed through the ALU to the input of ACCA. On the next clock cycle, the value will be latched into ACCA. Note that you do not want *PC\_I* active because PC is already pointing to the next instruction to be executed.

**Example 2:**

The next instruction in the program is `LDAA #num` where `#num = 0xF5`. This instruction translates as "load accumulator A with the value F5". Assume the op code for `LDAA #num` is `0x02`. Before the program begins, the situation is as below:

PC	Memory Address	Memory Data
→	02	02
	03	F5
	04	Next instruction

**INST = ??**

**MAR = ??**

C1: The fetch cycle is the same for this command as it was in Example 1 (The fetch cycle is the same for all commands). After the fetch cycle the situation should be:

PC	Memory Address	Memory Data
	02	02
→	03	F5
	04	Next instruction

**INST = 02 (LDAA #num op code)**

**MAR = ??**

C2: During C2 the PC is pointing at memory address `0x03`. By reading this address, the value `0xF5` is placed on the DATA line. *READ*, *ACCA\_L* and *PC\_I*, should be active, *MEM\_SEL* should be set to select *PC*, and the *ALU\_CTL* lines should select the function which loads *ACCA*. When the control lines are set up like this, the value `0x03` will be on the address lines of the memory unit, and the data lines out of the memory unit will contain the data in address `0x03` (which, in this example, is `0xF5`). This data will be passed through the *ALU* to the input of *ACCA*. On the next clock cycle the data will be latched into *ACCA*. There is no C3 cycle.

Shown below is some code to implement the **LDAA addr** and **LDAA #num** instructions. This is just one possible implementation. (NOTE this is not a complete program, just a portion of code) Here the op code for **LDAA addr** is represented by the constant `I0`, and the op code for **LDAA #num** is represented by the constant `I1`:

```

module CONTROL (IN, TIN, TOUT, CLK, RESN, ZREG, CREG, ALU_CTL, MEM_SEL, INST_L,
                PC_I, PC_L, ACCA_L, MAR_L, C_L, Z_L, X_I, X_L, READ, STORE, OUTPUT_ENA);
    input [7:0] IN;
    input [1:0] TIN;
    input CLK, RESN, ZREG, CREG;
    output reg [5:0] ALU_CTL;
    output reg [1:0] MEM_SEL, TOUT;
    output reg INST_L, PC_I, PC_L, ACCA_L, MAR_L, C_L, Z_L, X_I, X_L, READ, STORE, OUTPUT_ENA;

    parameter [1:0] T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;
    parameter [7:0] I0 = 8'h00, I1 = 8'h01, I2 = 8'h02, I3 = 8'h03,
                   I4 = 8'h04, I5 = 8'h05, I6 = 8'h06, I7 = 8'h07,
                   I8 = 8'h08, I9 = 8'h09, I10 = 8'h0a, I11 = 8'h0b,
                   I12 = 8'h0c, I13 = 8'h0d, I14 = 8'h0e, I15 = 8'h0f;
    parameter VCC = 1'b1, GND = 1'b0;

    always @(posedge CLK)
    begin
        ALU_CTL = 3'bxx; MEM_SEL = 2'bzz;
        INST_L = VCC; PC_I = VCC; PC_L = VCC; ACCA_L = VCC; MAR_L = VCC;
        C_L = VCC; Z_L = VCC; X_I = VCC; X_L = VCC; READ = VCC; STORE = VCC;
        if (RESN)
            begin
                MEM_SEL = 2'b00;
            end
        else
            begin
                case (TIN)
                    T0:
                        begin
                            INST_L = GND; MEM_SEL = 2'b10; READ = GND; PC_I = GND; TOUT = T1;
                        end
                    T1:
                        begin
                            case (IN)
                                I0:
                                    begin
                                        MEM_SEL = 2'b10; READ = GND; MAR_L = GND;
                                        PC_I = GND; TOUT = T2;
                                    end
                                I1:
                                    begin
                                        MEM_SEL = 2'b10; READ = GND; ALU_CTL = 4'b0001;
                                        ACCA_L = GND; PC_I = GND; TOUT = T0;
                                    end
                            end
                        end
                end

                // OTHER INSTRUCTIONS

            endcase
        end
        T2:
            begin
                case (IN)
                    I0:
                        begin
                            MEM_SEL = 2'b11; READ = GND; ALU_CTL = 4'b0000;
                            ACCA_L = GND; TOUT = T0;
                        end
                end

                // OTHER INSTRUCTIONS

            endcase
        end
    endcase
end
end
end
end

```

**Example 3:**

The next instruction in the program is JMP addr where addr = 0xF5. Assume the op code for JMP addr is 0x12. Before the program begins, the situation is as below:

PC	Memory Address	Memory Data
→	04	12
	05	F5
	06	Next instruction

**INST = ??**

**MAR = ??**

C1: The fetch cycle is the same for this command as it was in Example 1 (The fetch cycle is the same for all commands). After the fetch cycle the situation should be:

PC	Memory Address	Memory Data
	04	12
→	05	F5
	06	Next instruction

**INST = 12 (JMP addr)**

**MAR = ??**

C2: During C2 the PC is pointing at memory address 0x05. By reading this address, the value 0xF5 is placed on the DATA line. READ, ACCA\_L and PC\_L, should be active, and MEM\_SEL should be set to select PC. When the control lines are set up like this, the value 0x05 will be on the address lines of the memory unit, and the data lines out of the memory unit will contain the data in address 0x05 (which, in this example, is 0xF5). This data will be on the input lines to PC. On the next clock cycle the data will be latched into PC. There is no C3 cycle.

Implement the Control Unit:

1. Assign opcodes to each instruction in the instruction set.
2. Draw the state diagram for the control unit.
3. Write an Altera program to implement the control unit. If you are unsure about an instruction or how to implement an instruction, ask a TA or lab instructor. It is vital for the functioning of the final computer that each command be implemented properly by the control unit.
  - This is a complex program. To improve readability you should use PARAMETER to assign values that are frequently used in your program (such as the opcodes.) For more on PARAMETER select the Quartus II Help menu, Search ..., type Constant keyword, then choose Constant Statement (Verilog).
  - You should also provide default values for the control signals.
4. Simulate the control unit in Altera. What happens when RESET is low? Test with different values for INST and check that the control unit cycles through the appropriate states for that instruction and that the control signals are what you expect. Test the JCS command both when the carry is set and when the carry is not set.