# 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 AHDL to Design State Machines`. There are also a few other blocks you will need to implement your computer – a multiplexer, 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, and in Part 3, you will implement and test the 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 an Altera 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 an Altera Program to implement the decoder.

3. Tri-State Buffer.

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

```
SUBDESIGN 8trin
(
    enan            : INPUT;  %Active low enable input %
    data_in[7..0]   : INPUT;
    data_out[7..0]  : BIDIR;
)
```

```
VARIABLE
    buffer[7..0]    : TRI;

BEGIN
    buffer[].oe = !enan;            % Enable buffer when enan is low %
    buffer[].in = data_in[7..0];
    data_out[7..0] = buffer[].out;

END;
```

**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 bf 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 require 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?)

| **Mnemoinic** | **Operation** |
|---|---|
| **LDAA addr**<br>*load ACCA from memory* | Loads ACCA with the value in memory at address *addr*<br><br>C stays the same, Z changes |
| **LDAA #num**<br>*load ACCA with an immediate* | Loads ACCA with *num*, the value in memory at the address immediately following the **LDAA #num** command<br>C stays the same, Z changes |
| **LDAA 0,X**<br>*load ACCA indexed* | Loads ACCA with the value in memory at the address in the X register. C stays the same, Z changes. |
| **STAA addr**<br>*store ACCA in memory* | Stores the value in ACCA at the memory address *addr*<br>C stays the same, Z changes |
| **ADDA addr**<br>*add ACCA and value in memory* | Adds the value in memroy location *addr* to the value in ACCA at saves the result in ACCA<br>Z and C change |
| **SUBA addr**<br>*subtract value in memory from ACCA* | Subtracts the value in memroy location *addr* from the value in ACCA at saves the result in ACCA<br>Z and C change |
| **ANDA addr**<br>*logical AND of ACCA and value in memory* | Perform a logical AND of the value in memroy location *addr* with the value in ACCA. Save result in ACCA<br>C stays the same, Z changes. |
| **ORAA addr**<br>*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<br>C stays the same, Z changes. |
| **CMPA addr**<br>*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 |
| **LDX #num**<br>*load X with an immediate* | Loads X with *num*, the value in memory at the address immediately following the **LDX #num** command<br>C stays the same, Z changes. |
| **INX**<br>*increment X* | Increment value in X<br>C stays the same, Z changes. |
| **CPX #num**<br>*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 |
| **COMA**<br>*complement ACCA* | Replace the value in ACCA with its one's complement<br>C is set to 1, Z changes. |
| **INCA**<br>*increment ACCA* | Increment value in ACCA<br>C stays the same, Z changes |
| **LSLA** | Logical shift left of ACCA. C and Z change. |
| **LSRA** | Logical shift right of ACCA. C and Z change. |
| **ASRA** | Arithmetic shift right of ACCA. C and Z change. |
| **JMP addr**<br>*jump* | Jumps to the instruction stored in address *addr*<br>(The value in *PC* is replaced with *addr*.) C and Z stay the same. |
| **JCS addr**<br>*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. |
| **JEQ addr**<br>*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.**

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 AHDL program should have a `DEFAULTS` section in which those signals will be high be default. In your AHLD 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 the 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 ines 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 0f 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 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 `LDAA`, and the op code for **LDAA #num** is represented by the constant `LDAA_IMM`:

```
VARIABLE
Control:    MACHINE WITH STATES (RESET, C1, C2, C3);

BEGIN
    DEFAULTS
        %Enter default values here%
    END DEFAULTS;

Control.clk = CLOCK;
Control.reset = !reset;

CASE Control IS
    WHEN RESET =>
        MEM_SEL = PROG_ADDR;
        IF reset_in == GND THEN
            Control = RESET;
        ELSE
            Control = C1;
        ENDIF;
    WHEN C1 =>
        INST_L = GND;
        MEM_SEL[] = PC;
        READ = GND;
        PC_I = GND;
        Control = C2;

    WHEN C2 =>
        CASE INST[] IS
            WHEN LDAA =>
                MEM_SEL[] = PC;
                READ = GND;
                MAR_L = GND;
                PC_I = GND;
                Control = C3;
            WHEN LDAA_IMM =>
                MEM_SEL[] = PC;
                ALU_CTL[] = ALU_LOAD;
                ACCA_L = GND;
                PC_I = GND;
                Control = C1;
            % Add other instructions here %
```

```
            WHEN OTHERS =>
                Control = C1;
        END CASE;

    WHEN C3=>
        CASE INST[] IS
            WHEN LDAA =>
                MEM_SEL[] = MAR;
                READ = GND;
                ALU_CTL[] = ALU_LOAD;
                ACCA_L = GND;
                Control = C1;
            % Add other instructions here %
            WHEN OTHERS =>
                Control = C1;
        END CASE;
    END CASE;
END;
```

**Example 3:**
The next instruction in the program is **JMP addr** where addr = 0xF5. Assume the 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** (JMW 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 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 assign CONSTANTs to values that are frequently used in your program (such as the opcodes.) For more on CONSTANT select the Quartus II **Help** menu, **Search ...**, type *Constant keyword*, then choose **Constant Statement (AHDL)**.
    - 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.