EE 231 Lab 4
Fall 2005

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. 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 three possible signals: the value in either `PROG_ADDR`, `PC`, or `MAR`.

   Write an Altera program to implement the `MUX`. (Remember, `PROG_ADDR`, `PC`, and `MAR` are each 8 bits wide).

2. 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 encoder.

3. You will need three 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;
```

Design and simulate an 8-bit tri-state buffer in AHDL.

Part 2. Design of the 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 three states: RESET, C1, C2, C3. Each state takes up one clock cycle.
•RESET is the **Reset** state. The computer gets into this state when the Reset button is pushed. This will reset the program counter PC to 0x00, which is the first instruction the computer will execute when Reset is released. In the Reset state, MEM_SEL should be set to PROG_ADDR. This will allow you to enter programs into memory. To enter a program, put the memory address you want to load on the PROG_ADDR lines, and put the data you want to load on the PROG_DATA lines. Then bring PROG_WRITE low, then high. This will write the value on PROG_DATA lines into the memory address on PROG_ADDR lines. Once the program is entered, release the Reset line. On the next clock cycle, the control unit should transition to state C1.
•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 **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. Some instructions require only require one execution cycle (C2) while others require two(C2 and C3).
•C3 is the second **Execution Cycle**. It is only needed by some of the instructions.

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

Mnemonic	*Operation*
LDAA addr *load ACCA from memory*	Loads register A with the value in memory at address 'addr'.
LDAA #num *load ACCA with an immediate value*	Loads register A with the 4 bit value immediately following the **LDAA #num** command in program memory.
STAA addr *store ACCA in memory*	Stores the value in register A at the memory address 'addr'.
ADDA addr *add ACCA and value in memory*	Adds the value in memory location 'addr' to the value in register A and places the result in register A.
SUBA addr *subtracts value in memory from ACCA*	Subtracts the value in memory location 'addr' from the value in register A. The result is stored in register A.
ANDA addr *logical AND of ACCA and value in memory*	Performs a logical AND of the value in memory location 'addr' with the value in register A. Puts the result in register A.
ORAA addr *logical OR of ACCA and value in memory*	Performs a logical OR of the value in memory location 'addr' with the value in register A. Puts the result in A.
CMPA addr *compares the value in ACCA to value in memory*	Compares the value in register A to the value in memory location 'addr'. This is done by subtracting the value in memory location 'addr' from the value in register A. The C bit reflects the result. The values in A and memory location 'addr' do not change.
COMA *complement ACCA*	Replaces the value in register A with its one's complement.
INCA *increment*	Adds one to the value in register A and stores the result in register A.
LSLA	Logical shift left of A
LSRA	Logical shift right of A
ASRA	Arithmetic shift right of A
JMP addr *jump*	Jumps to the instruction stored at address 'addr'. (The value in PC is replaced with 'addr'.)
JCS addr *jump if carry is set*	Checks to see if the carry is set. -If the carry is set, then the PC is loaded with 'addr'. -If the carry is not set, the jump command is ignored. The PC is incremented and the program continues with the next instruction.

Figure 1: Instruction Set of Computer

The output 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 AHDL code, you will bring the signals low 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:

The first instruction in the program is "LDAA addr". In this example, we will assume addr = 0xF5. We will further assume that the instruction is the first instruction of the program, and is located at address 0x00.

This instruction translates as "load accumulator A with the value located in memory address 0xF5". If this is the first line of the program, and the program starts at address 0x00 in memory, then before the fetch cycle the PC is pointing at address 0x00 as shown below.

PC	***Memory Address***	***Memory Data***
→	00	LDAA addr
	01	F5
	02	??

INST = ??
MAR = ??

<u>C1</u> During the Fetch Cycle the instruction register must be loaded with the instruction (LDAA addr). 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. Now the situation is as below:

PC	*Memory Address*	*Memory Data*
	00	LDAA addr
→	01	F5
	02	??

INST = LDAA addr
MAR = ??

(NOTE: The instruction register cannot actually contain "LDAA addr", it will contain the binary/hex op code you have chosen to represent the command LDAA addr).

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. After these steps the situation should be as shown below. Thus during C2, you should have PC_I low, MAR_L low, READ low, and MEM_SEL set to PC.

PC	*Memory Address*	*Memory Data*
	00	LDAA addr
	01	F5
→	02	??

INST = LDAA addr
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 loaded onto the DATA line. Then the ALU can load this value into ACCA. During C3, you should have PC_I low, READ low, ACCA_L low, and ALU_CTL set to the value which implements the LOAD function of the ALU. When the control lines are set up like this, the value 0xF5 will be on the address lines of the memory unit, the data lines out of the memory unit 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 low, because the program counter is already pointing to the next instruction to be executed, and should not be incremented.

Example 2:

The first instruction in the program is "LDAA #num" where #num = F5

This instruction translates as "load accumulator A with the value F5". Before the program begins, the situation is as below:

PC	***Memory Address***	***Memory Data***
→	00	LDAA #num
	01	F5
	02	?

INST = ?

<u>**C1**</u> 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***
	00	LDAA #num
→	01	F5
	02	?

INST = LDAA #num

<u>**C2**</u> During C2 the PC is pointing at memory address 0x01. By reading this address, the value 0xF5 is placed on the DATA line. READ, ACCA_L and PC_I should be low, 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 0x01 will be on the address lines of the memory unit, the data lines out of the memory unit will contain the data in address 0x01 (which is 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.

Shown below is some code to implement the commands **LDAA addr** and **LDAA #num**. This is just one possible implementation. (NOTE this is <u>not</u> a complete program, just a portion of code) Here **LDAA addr** is represented by the constant LDAA, and **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;
          Control = C1;
     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 first instruction in the program is “JMP addr” where addr = 0xF5

This instruction translates as “load program counter PC with the value F5”. Before the program begins, the situation is as below:

PC	***Memory Address***	***Memory Data***
→	00	JMP
	01	F5
	02	?

INST = ?

__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***
	00	JMP
→	01	F5
	02	?

INST = JMP addr

__C2__ During C2 the PC is pointing at memory address 0x01. By reading this address, the value 0xF5 is placed on the DATA line. READ, ACCA_L and PC_L should be low, and MEM_SEL should be set to select PC. The ALU is not being used. When the control lines are set up like this, the value 0x01 will be on the address lines of the memory unit, the data lines out of the memory unit will contain the data in address 0x01 (which is 0xF5). On the next clock edge, the value 0xF5 will be loaded into the program counter PC, so the next instruction the computer will execute will be that at address 0xF5. 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 see Altera's **Help** menu, **Search for Help on...**, type "Constant keyword", then choose *Constant Statement Description (AHDL)*.
    - You should also provide default values for the control signals.
      In Altera's **Help** menu, **Search for Help on...** type "Defaults Statement", then choose *Defaults Statement Description (AHDL)*.

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.