

## Lab 8: Computer Control Unit (CCU)

### Introduction

At the core of all computers is a control unit. It provides mechanisms the procedurally stepping through instructions. When executing an instruction the computer must step through multiple states. The task of stepping the computer through states and generating the necessary signals at each state is the purpose of the control unit. With some care, the system can be configured to work through a list of instructions, rather than just one. This forms the basis of software processors.

## 1 Prelab

Lab is nearing a first computer, but the control unit must first be built. A conceptual block diagram of a simple computer is shown in Figure 1. In previous labs the DATA MUX, the ALU, and required registers were already built. 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**, **FETCH**, **EX1** and **EX2**.

- **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.
- **FETCH** 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 (**IRX**).
- **EX1** is the first execution cycle. Once an instruction has been loaded into **IRX**, the control unit determines the required course of action to take based on the value of **IRX** and the current state of the control unit.
- **EX2** is the second execution cycle. Some instructions only require one execution cycle (**EX1**) while others require two (**EX1**, and **EX2**).

- 1.1. The output of the control unit depends on both the present state and the input. **What type of state machine is this?**
- 1.2. Draw the state diagram for the control unit.

## 2 Lab

- 2.1. Assign op codes to each instruction in the instruction set (Table 1.) **Justify your design choices**; thought now, can simplify problems later.
  - To improve readability, use **parameter(s)** to assign values that are frequently used in your program, e.g., op codes.
  - You should also provide default values for the control signals.

- 2.2. Write a Verilog program to implement the control unit.
- 2.3. Simulate the control unit. **What happens when Reset is low?** Test with different values for IRX 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 when the carry is set and when the carry is not set.**

### 3 Supplement: Control Signals

The outputs of the control unit are the control signals shown on the block diagram (Figure 1). Except for `Alu_Ctrl` and `Addr_Mux_Sel`, all of these signals are active low. In your Verilog code you will activate the appropriate signals at the correct times to implement the instruction the control unit is executing.

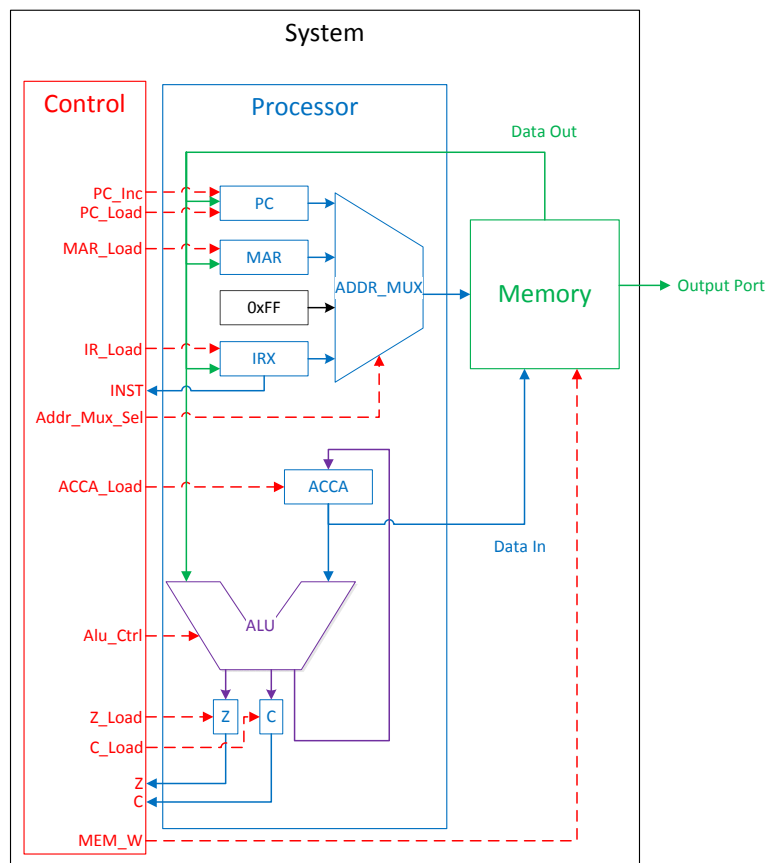


Figure 1: Processor Block Diagram

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 where it will read the instruction from memory and latch it into the IRX register. To do this, `IR_Load` and `PC_Inc` should be active (low), and `Addr_Mux_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 which is the address of the next instruction to execute, and the memory output enable line will be active (low). The memory will put the data at that address on its output lines, which are the input lines to the IRX register. On the next clock edge, the data from memory will be latched into the IRX 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 IRX register. Three examples are discussed in the Supplement (Section 4.)

## 4 Supplement: Code Execution

### 4.1 Example 1

Consider the instruction LDAA addr where addr = 0xF5. We will further assume that the instruction is in memory address 0x80 and 0x81, and that the code for LDAA addr is 0x01.

Table 2: Example Program 1: RST State

PC	Memory Address	Memory Data
→	0x80	0x01
	0x81	0xF5
	0x82	Next

IRX = ?

MAR = ?

**FETCH:** During the fetch cycle the instruction register must be loaded with the instruction operational (op.) code, 0x01. To do this the Addr\_Mux\_Sel must select the PC as the address source and the memory address 0x80 must be read which causes its value to be placed on the Data lines. The value on the Data lines must be latched into IRX, and the PC must be incremented. Thus during FETCH you should have PC\_Inc, IR\_Load and Addr\_Mux\_Sel.

Table 3: Example Program 1: FETCH State

PC	Memory Address	Memory Data
	0x80	0x01
→	0x81	0xF5
	0x82	Next

IRX = 0x01 (LDAA addr op code)

MAR = ?

**EX1:** During EX1, you must read the memory address that the PC is pointing at. By reading address 0x81 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 EX1 you should have PC\_Inc and MAR\_Load active, and Addr\_Mux\_Sel set to PC. After these steps the situation should be as shown in Table 4.

Table 4: Example Program 1: EX1 State

PC	Memory Address	Memory Data
	0x80	0x01
	0x81	0xF5
→	0x82	Next

IRX = 0x01 (LDAA addr op code)  
MAR = 0xF5

**EX2:** 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 **EX2** you should have **ACCA\_Load** active, **Addr\_Mux\_Sel** set to **MAR**, and **ALU\_Ctrl** 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\_Inc** active because **PC** is already pointing to the next instruction to be executed.

## 4.2 Example 2

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

Table 5: Example Program 2: RST State

PC	Memory Address	Memory Data
→	0x82	0x02
	0x83	0xF5
	0x84	Next

IRX = ?  
MAR = ?

**FETCH:** The fetch cycle is the same for this command as it was in Example 1 (Section 4.1.) After the fetch cycle the situation should be:

Table 6: Example Program 2: FETCH State

PC	Memory Address	Memory Data
	0x82	0x02
→	0x83	0xF5
	0x84	Next

IRX = 0x02 (LDAA #num op code)  
MAR = ?

**EX1:** During the EX1 cycle the PC is pointing at memory address 0x83. By reading this address, the value 0xF5 is placed on the Data line. ACCA\_Load and PC\_Inc, should be active, Addr\_Mux\_Sel should be set to select PC, and the ALU\_Ctrl lines should select the function which loads ACCA. When the control lines are set up like this, the value 0x83 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 0x83 (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 EX2 cycle.

### 4.3 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:

Table 7: Example Program 3: RST State

PC	Memory Address	Memory Data
→	0x84	0x12
	0x85	0xF5
	0x86	Next

IRX = ?

MAR = ?

**FETCH:** The fetch cycle is the same for this command as it was in Example 1 (Section 4.1.) After the fetch cycle the situation should be:

Table 8: Example Program 3: FETCH State

PC	Memory Address	Memory Data
	0x84	0x12
→	0x85	0xF5
	0x86	Next

IRX = 0x12 (JMP addr op code)

MAR = ?

**EX1:** During the EX1 cycle the PC is pointing at memory address 0x85. By reading this address, the value 0xF5 is placed on the Data line. PC\_Load should be active, and Addr\_Mux\_Sel should be set to select PC. When the control lines are set up like this, the value 0x85 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 0x85 (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 EX2 cycle.

**Note:** When making conditional jumps, be sure to increment the Program Counter (PC) if the jump is not executed. Otherwise, the address will be interpreted (undesiredly) as the next instruction to be executed.

Table 1: Computer Instructions

Op.Code	Instruction	Operation (Mnemonic)
	nop	Do nothing. (No Operation)
	LDDA addr	Loads ACCA with the value in memory at address addr. C stays the same, Z changes. (Load ACCA from memory)
	LDDA_IMM #num	Loads ACCA with num, the value in memory at the address immediately following the LDAA #num command. C stays the same, Z changes. (Load ACCA with an immediate)
	STAA addr	Stores the value in ACCA at memory address addr. C stays the same, Z changes. (Store ACCA in memory)
	ADDA addr	Adds the value in memory location addr to the value in ACCA and saves the result in ACCA. C and Z change. (Add ACCA and value in memory)
	SUBA addr	Subtracts the value in memory location addr from the value in ACCA and saves the result in ACCA. C and Z change. (Subtract value in memory from ACCA)
	ANDA addr	Perform a logical AND of the value in memory location addr with the value in ACCA. Save the result in ACCA. C stays the same, Z changes. (Logical AND of ACCA and value in memory)
	ORAA addr	Perform a logical OR of the value in memory location addr with the value in ACCA. Save the result in ACCA. C stays the same, Z changes. (Logical OR of ACCA and value in memory)
	CMPA addr	Compare ACCA to value in addr. This is done by subtracting the value in addr from ACCA. ACCA does not change. C and Z change. (Compares ACCA to the value in addr)
	COMA	Replace the value in ACCA with its one's complement. C is set to 1 and Z changes. (Compliment ACCA)
	INCA	Increment value in ACCA. C stays the same and Z changes. (INCA ACCA)
	LSLA	Logical shift left of ACCA. C and Z change. (Logical shift left ACCA)
	LSRA	Logical shift right of ACCA. C and Z change. (Logical shift right ACCA)
	ASRA	Arithmetic shift right of ACCA. C and Z change. (Arithmetic shift right ACCA)
	JMP addr	Jumps to the instruction stored in address addr. The PC is replaced with addr. C and Z stay the same. (Jump)
	JCS addr	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. (Jump if carry set)
	JCC addr	Jumps to the instruction stored in address addr if C = 0. If C is set, continue with next instruction. C and Z stay the same. (Jump if carry not set)
	JEQ addr	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. (Jump if Z set)