ECE 375
Computer Organization and Assembly Language Programming
Fall 2022
Solutions Set #1

[20 pts]

1- (a) Suppose a processor or CPU supports 64 different instructions and has a memory of 1K (K = 1024) words. Determine the size of each memory word for the following instruction formats:
(i) 3-address instruction format
(ii) 2-address instruction format
(iii) 1-address instruction format

Solution:

The opcode field is $\log_2 64 = 6$ bits, and each address field is $\log_2 1024 = 10$ bits. Thus, 3-address instruction format requires 36 bits, 2-address instruction format requires 26 bits, and 1-address instruction format requires 16 bits.

(b) Explain the advantages and disadvantages of the three instruction formats.

Solution:

The advantage of a 3-address instruction format is that it allows both the sources as well as the destination addresses to be defined, and all three locations can be different (e.g., $z \leftarrow x + y$). As such, it provides the most flexibility in defining a binary operation. The disadvantage of a 3-address instruction is that it requires three fields to define the addresses for the two source operands and the destination.

The advantage of a 2-address instruction format is that it provides similar capability as the 3-address instruction format with one less field, and thus, the instruction format can be shorter (e.g., $x \leftarrow x + y$). However, since one of the source addresses has to also serve as the destination address, an extra instruction will be required to move the result to another location other than the two sources addresses.

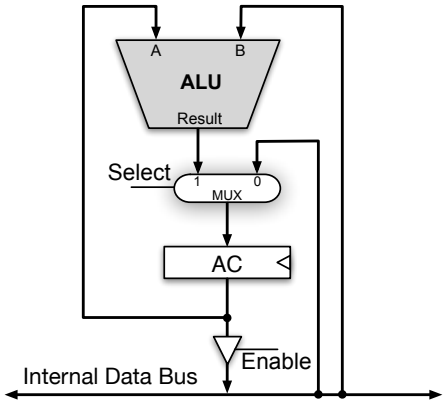
The advantage of a 1-address instruction is it size, since only one source address needs to be explicitly defined and the other source/destination is implied i.e., top of the stack (e.g., TOS $\leftarrow x +$ TOS)). However, since the stack is implied, extra moves/copies are needed to push/pop an operand onto/from the stack.

[20 pts]

2- Show a block diagram of the hardware necessary (similar to Figures 3.12 - 3.13 in the textbook) to connect AC to the Internal Data Bus and the ALU in the pseudo-CPU using a tri-state buffer and a multiplexer (MUX).

Solution

The figure below shows one possible solution. The AC register has two possible inputs: one from the output of the ALU, and the other from other registers on the Internal Data Bus. Therefore, a multiplexer (MUX) with the Select control signal is needed to choose from one of these two possibilities. The output of the AC register can go to either the left input (A) of the ALU or to the Internal Data Bus, which is gated with a tri-state buffer with the Enable control signal.



[20 pts]

3- For the pseudo-CPU shown in Figure 3.9 in the textbook, explain whether or not each of the following microoperations can be performed in a single clock cycle. Assume PC and MAR each contain 12 bits, AC and MDR each contain 16 bits, and IR is 4 bits.

(a) MAR ← AC, MDR ← MAR
(b) IR ← MDR, MAR ← MDR
(c) MAR ← MDR, MDR ← M(MAR)
(d) MDR ← AC + 1
(e) AC ← MDR, PC ← PC + 1
(f) PC ← PC + AC

Solution:

(a) These two data transfer operations cannot be performed at the same time. This is because both data transfer operations require the Internal Data Bus at the same time.

(b) These two data transfer operations can be performed at the same time. However, the bits that are transferred need to be more specifically defined. That is, MDR is 16 bits, but IR is only 4 bits and MAR is only 12 bits. Thus, the correct way of specifying these two data transfer operations is

IR ← MDR(15...12), MAR ← MDR(11...0) or
IR ← MDR(opcode), MAR ← MDR(address)

(c) These two data transfer operations can be performed at the same time. Suppose MDR contain *x* and MAR contain *y*. During the clock cycle, M(*y*) will be performed and the value read will be available at the input of MDR but not yet latched. At the same time, *x* from MDR will be available at the input of MAR but not yet latched. At the end of the clock, MDR will have M(*y*) and MAR will have *x*.

(d) This data transfer or microoperation cannot be performed. The reason is shown in Figure 3.10 in the textbook. As can be seen, the left input (A) to the ALU is AC and the right input (B) to the ALU can be any one of the registers connected to the Internal Data Bus. However, the result of the ALU operation can only be latched onto the AC.

(e) These two data transfer operations can be performed at the same time because the PC increment is done independent of the Internal Data Bus.

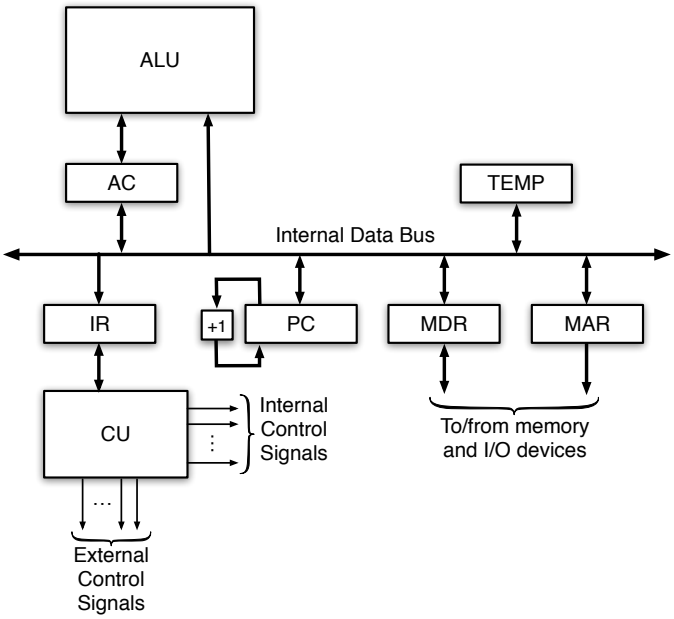
(f) This data transfer operation or microoperation cannot be performed. The reason is that the increment unit (+1) only accepts PC as input.

[20 pts]

4- Consider the following hypothetical 1-address assembly instruction called "Add Then Store Indirect with Pre-decrement" of the form

`ADDTHENSTORE -(x)  ; M[x] ← M[x]-1, M[M[x]] ← AC+M[M[x]]`

Suppose we want to implement this instruction on the pseudo-CPU discussed in class augmented with TEMP register as shown below. Give the sequence of *microoperations* required to implement the Execute cycle (Fetch cycle is given below) for the above `ADDTHENSTORE -(x)` instruction. *Your solution should result in exactly 9 microoperations*. Assume an instruction consists of 16 bits: A 4-bit opcode and a 12-bit address. All operands are 16 bits. PC and MAR each contain 12 bits. AC, MDR, and TEMP each contain 16 bits, and IR is 4 bits. Note that the original content of AC should be preserved. Assume PC is currently pointing to the `ADDTHENSTORE` instruction and only PC and AC have the capability to increment/decrement itself.



Fetch Cycle
Cycle 1: MAR ← PC;
Cycle 2: MDR ← M[MAR], PC ← PC+1 ; Read inst. & increment PC
Cycle 3: IR ← MDR_{opcode}, MAR← $MDR_{address}$

Solution:

This instruction is similar to the LDA -(x), but instead of just loading the operand in to AC, you have to perform AC ← AC + operand and then the result needs to be stored back in memory location pointed by EA (i.e., M[M[x]]. Thus, the instruction can be described by the following three steps:

M[x] ← M[x]–1 ; Predecrement and store back the address pointing to the operand
AC ← AC + M[M[x]] ; Load the operand in to AC and perform add
M[M[x]] ← AC + M[M[x]] ; Store the result back into memory

The solution requires 9 microoperations because several parallel operations are possible. In Cycle 1, reading from memory and moving AC to TEMP can be done at the same time. In Cycle 5, reading memory and moving the content of MDR into MAR does not conflict because MAR latches the content at the end of the cycle. Thus, the previous content of MAR is valid until the end of the cycle, which is need to properly read from memory. In addition, the content of TEMP can also be moved to AC at the same time. Finally, in Cycle 9, writing to memory and moving the content of TEMP into AC can be done at the same time.

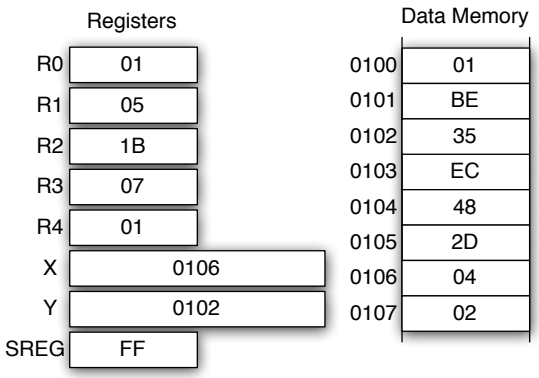
Execute Cycle
Cycle 1: MDR ← M[MAR], TEMP ← AC ; Read EA+1 and save AC in TEMP
Cycle 2: AC ← MDR ; Move EA+1 into AC
Cycle 3: AC ← AC - 1 ; Decrement EA+1

Cycle 4: MDR ← AC	; Move EA into MDR ;
Cycle 5: M[MAR] ← MDR, MAR ← MDR	; Store EA back in location pointed by x and move ; EA into MAR
Cycle 6: MDR ← M[MAR], AC ← TEMP	; Read operand and restore AC
Cycle 7: AC ← AC + MDR	; Add and transfer result to AC
Cycle 8: MDR ← AC	; Store result back in memory
Cycle 9: M[MAR] ← MDR, AC ← TEMP	; and restore AC

[20 pts]

5- Based on the initial register and data memory contents shown below (represented in hexadecimal), show how these contents are modified (in *hexadecimal*) after executing each of the following AVR assembly instructions. Do not be concerned about what happens to the Status Register (SREG) *after* the operation. *Instructions are unrelated*.

(i) `ldi r27, 85`
(ii) `ror r2`
(iii) `adc r2, r1`
(iv) `sts $0007, r28`
(v) `sbiw XH:XL, 2`



Solution:

(i) R27, which is the upper byte of the X-register, changes to \$55 (i.e., 85_{10}). Thus, X changes to \$5506.
(ii) R2 is 0b00011011, thus rotating it right through the carry, which is 1 because SREG is \$FF, results in 0b10001101 = \$8D. Thus, R2 changes to \$8D.
(iii) Since SREG indicates \$FF, C-bit (LSB) is set.
Thus, \$1B + \$05 + \$01(carry) = \$21
R2 changes to \$21
(iv) Since R28 is the lower byte of the Y-register, i.e., R28 = \$02, and thus M[\$0007]=\$02.
(v) Since X is \$0106, subtracting 2 from it results in \$0104. Thus, X changes to \$0104.