ECE 375 Homework 1

Computer Organization and Assembly Language Programming

Winter Term 2018

Jeremy Fischer

1 Homework Questions

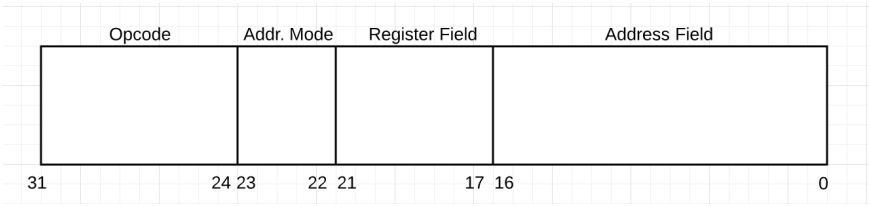
1. Consider a 1-address CPU that has a memory unit with 128K words of 32 bits each. An instruction is stored in one memory word. The instruction format is divided into four fields: an opcode field, a 2-bit addressing mode field that specify direct, indirect, indirect with pre-decrement, or indirect with post-increment addressing mode, a register field that specifies one of 32 registers, and an address field. For your information, given an *address*

   - Direct addressing is where the operand is located in M[*address*].
   - Indirect addressing is where the operand is located in M[M[*address*]].
   - Indirect addressing with pre-decrement is where the operand is located in M[-M[*address*]].
   - Indirect addressing with post-increment is where the operand is located in M[M[address]+].

   (a) What is the maximum number of opcodes that can be incorporated into the CPU? How many bits are in the opcode field, the register field, and the address field? Draw the instruction format and indicate the number of bits in each field.

   Answer:

   The number of opcodes is 2^{K} where K is the number of bits occupied by a single opcode. Therefore, the number of opcodes is $2^8 = 256$ opcodes.



   (b) If this 1-address format is to be implemented on the pseudo-CPU discussed in class, how many bits are required for the registers PC, MAR, MDR, IR, and AC?

   Answer:

   If we were using the CPU discussed in class then:

   PC: 13-bits because it holds an address, the address field of the CPU in class is 13-bits

   MAR: 13-bits because it holds an address, the address field of the CPU in class is 13-bits

   MDR: 16-bits because it holds words, a word in the CPU in class is 16-bits

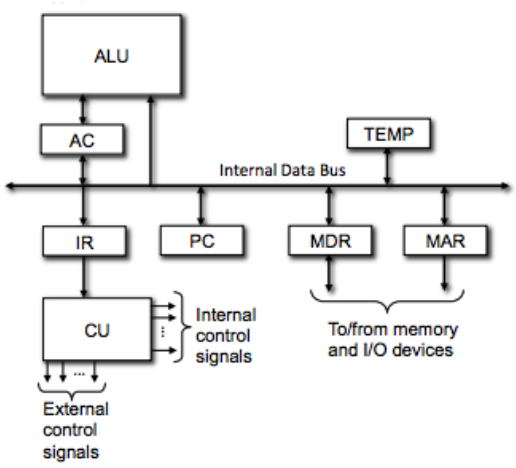
   IR: 3-bits because it holds opcodes, an opcode in the CPU in class is 3-bits

   AC: 16-bits

2. Consider the following hypothetical 1-address assembly instruction called Store Accumulator with Post-increment of the form

```
STA (x)+   ; M(M(x)) <-- AC,     M(x) <-- M(x) + 1
```

Suppose we want to implement this instruction on the pseudo-CPU discussed in class augmented with a temporary register TEMP. 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. Give the sequence of *microoperations* required to implement the Execute cycle (Fetch cycle is given below) for the above STA (x)+ instruction. Your solution should result in exactly 8 microoperations. Assume PC is currently pointing to the STA (x)+ instruction and only PC and AC have the capability to increment/decrement itself.



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

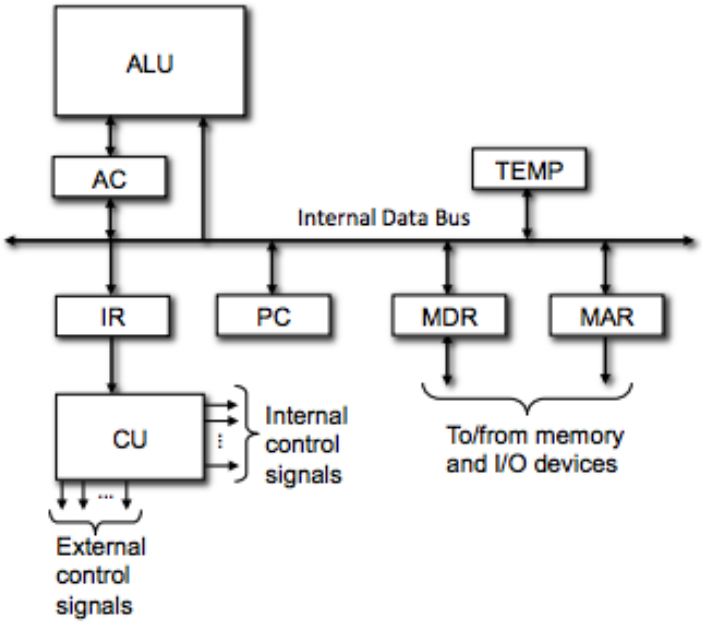
Answer:

Step 1: MDR ← AC ;move value to MDR
Step 2: TEMP ← MAR ;store x so you can access later
Step 3: M[M[MAR]] ← MDR ;store value at M(M(x)) where x is held in MAR
Step 4: AC ← M[MAR] ;move M(x) to AC so M(x) can be incremented
Step 5: AC ← AC + 1 ;increment AC, M(x) + 1
Step 6: MDR ← AC ;move M(x) + 1 to MDR
Step 7: M[TEMP] ← MDR ;store M(x) + 1 in M(x)
Step 8: PC ← PC + 1 ;get the next instruction

3. The PDP-8 was the first successful commercial minicomputer, produced by Digital Equipment Corporation (DEC) in the 1960s (see http://en.wikipedia.org/wiki/PDP-8). One of the assembly instructions it supported was Increment and Skip if Zero (ISZ) of the form

```
ISZ Y ; M(Y) <-- M(Y) + 1,  If (M(Y) + 1 = 0) Then PC <-- PC + 1
```

Suppose the pseudo-CPU discussed in class with a temporary register (TEMP) shown below can be used to implement the ISZ instruction. Give the sequence of *microoperations* required to fetch and execute ISZ. Your solution should result in minimum number of *microoperations*. You must not destroy the original content of the AC. Assume PC is currently pointing to the instruction and only PC and AC have the capability to increment itself.



Answer:

Step 1: MAR ← PC ;move the instruction address to MAR

Step 2: MDR ← M[MAR] ;move the opcode to MDR

Step 3: IR ← $\text{MDR}_{\text{opcode}}$;load the instruction to IR to be decoded

Step 4: MAR ← $\text{MDR}_{\text{address}}$;move Y to MAR

Step 5: PC ← PC + 1 ;increment PC to point to next instruction

Finished Fetching

Step 6: TEMP ← AC ;store old value of AC

Step 7: MDR ← M[MAR] ;MDR gets M(Y)

Step 8: AC ← MDR ;AC gets M(Y)

Step 9: AC ← AC + 1 ;AC gets M(Y) + 1

Step 10: MDR ← AC ;MDR gets M(y) + 1

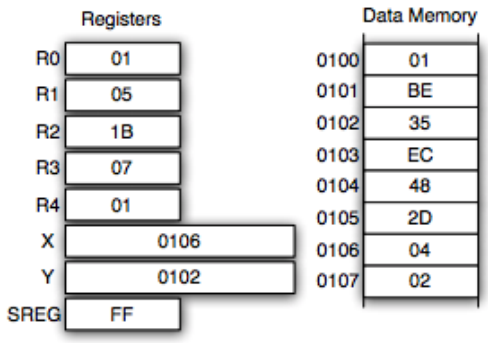
Step 11: M[MAR] ← MDR ;M(y) gets M(y) + 1

Finished Incrementing

Step 12: If (Z = 1) then PC ← PC + 1 ;if Z = 1 last ALU operation was 0, skip

Step 13: AC ← TEMP ;move original AC value back to AC

4. 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 `CPI R28, 2`

CPI compares the register and the immediate values

The operation (0x02 - 0x02) results in zero. Therefore, the Zero flag of SREG (bit 1) is set signifying that the two are equal.

ii `STD Y+0x05, R2`

STD stores contents of R_r in indirect with displacement

After the operation data memory at 0x0107 has the value 0x1B

iii `LD  R3, -X`

LD loads indirect (& with pre-decrement/post-increment)

After the operation R3 has the value 0x2D. X holds address 0106, but due to the pre-decrement, the real address we want is 0105, which holds 0x2D.

iv `EOR  R1, R3`

EOR is exclusive-OR

After the operation R1 has the value 0x02

v `SBIW  R29:R28, 8`

SBIW subtracts an 8- bit value from a 16-bit register, such as X, Y or Z

After the operation R29:R28 is 0x00FA

vi `SBR  R26, 7`

SBR sets bits in a register

After the operation register R26 contains 0x07