## Supplement for Assignment #7 (sections 4.8 - 4.10 of the textbook)

# Summary of the MARIE Assembly Language

| Type of Instructions | Mnemonic | Hex Opcode | Description |
|---|---|---|---|
| Arithmetic | ADD X | 3 | Add the contents of address X to AC |
| | SUBT X | 4 | Subtract the contents of address X from the AC |
| | ADDI X | B | Add Indirect: Use the value at X as the actual address of the data operand to add to AC |
| | CLEAR | A | Put all zeros in the AC |
| Data Transfer | LOAD X | 1 | Load the contents of address X into AC |
| | STORE X | 2 | Store the contents of AC at address X |
| I/O | INPUT | 5 | Input a value from the keyboard into AC |
| | OUTPUT | 6 | Output the value in AC to the display |
| Branch | JUMP X | 9 | Unconditional branch to X by loading the value of X into PC |
| | SKIPCOND C | 8 | Skip the next instruction based on the condition, C:<br>C = $000_{16}$: skip if AC is negative ($b_{11}b_{10} = 00_2$)<br>C = $400_{16}$: skip if the AC = 0 ($b_{11}b_{10} = 01_2$)<br>C = $800_{16}$: skip if the AC is positive ($b_{11}b_{10} = 10_2$) |
| Subroutine call and return | JNS X | 0 | Jump-and-Store: Store the PC at address X and jump to X+1 |
| | JUMPI X | C | Use the value **at** X as the address to jump to |
| | HALT | 7 | Terminate the program |

MARIE Machine-language Instruction Format

| 15 — 12 | 11 10 — 0 |
|---|---|
| Opcode | Address (or Condition) |

A simple MARIE program can be written to perform the high-level language statements:

RESULT = X + Y - Z
print RESULT

| Address | Label | Assembly Language | Machine Language |
|---|---|---|---|
| 0 | | LOAD X | $1006_{16}$ |
| 1 | | ADD Y | $3007_{16}$ |
| 2 | | SUBT Z | $4008_{16}$ |
| 3 | | STORE RESULT | $2009_{16}$ |
| 4 | | OUTPUT | $6000_{16}$ |
| 5 | | HALT | $7000_{16}$ |
| 6 | X, | DEC 10 | $000A_{16}$ |
| 7 | Y, | DEC 20 | $0014_{16}$ |
| 8 | Z, | DEC 5 | $0005_{16}$ |
| 9 | RESULT, | DEC 0 | $0000_{16}$ |

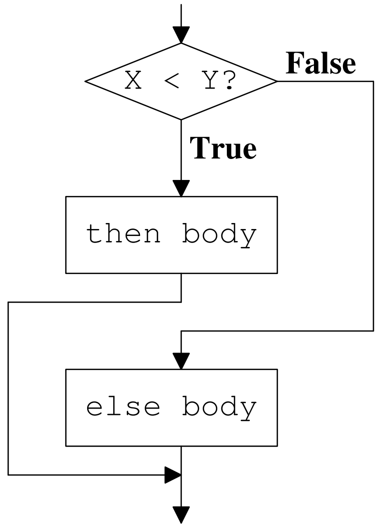
The lines at address 6 to 9 are *assembler directives* (directions to the assembler) to initialize the memory location associated with X (address 6) to DECimal 10, the memory location associated with Y (address 7) to 20, etc. Lines at address 0 to 5 are the actual machine-language MARIE program. If the PC = 0 (program counter), the program execution would start at address 0 which contains $1006_{16}$. This instruction would be fetched into the CPUs IR (instruction register), bits 15-12 contain the operations code of $1_{16}$ would be decoded to determine that it is a LOAD instruction. Execution of the LOAD causes the specified memory address's ($006_{16}$ in bits 11-0) content to be loaded into the accumulator (AC) register (i.e., the value $10_{10}$ would be loaded into the AC). During the fetch-decode-execute cycle, the PC would get incremented to the next instruction. The program instructions are executed sequentially until the HALT instruction which stops the program.

The branch instructions, JUMP and SKIPCOND, potentially cause the PC to "jump" (i.e., alter the *flow of control* in the program). These instructions are useful for implementing high-level language selection (IF, IF-THEN-ELSE, SWITCH, etc.) and looping statements (FOR, WHILE, REPEAT, etc.). For example, consider the following IF-THEN-ELSE statement and corresponding flow-chart:

**HLL statement**

```
if X < Y then
    ...
else
    ...
end if
```

**Flow chart**




**Assembly Language**

```
          LOAD X
          SUBT Y
          SKIPCOND 000
          JUMP ELSE
THEN,

          JUMP END_IF
ELSE,

END_IF,
```

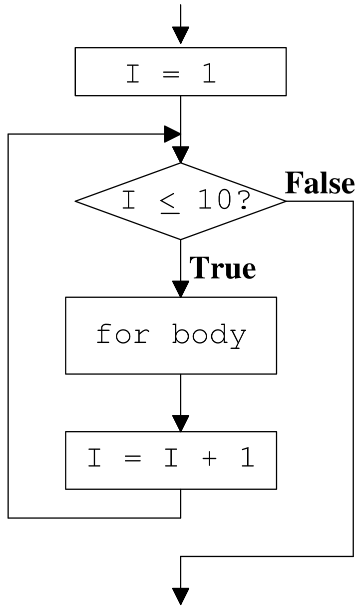
If X < Y is True, then the value of (X-Y) in the AC is negative. The "SKIPCOND 000" cause the JUMP ELSE instruction to be jumped over if the AC is negative. Since the then-part code follows the JUMP ELSE instruction, it is only executed if X < Y. After the then-part code is executed, the JUMP END_IF causes the else-body to be skipped. If X < Y is False, then the value of (X - Y) in the AC will not be negative the SKIPCOND 000 instruction will not jump over the JUMP ELSE instruction.

For a loop example, consider the following FOR-loop and corresponding flow-chart:

**HLL statement**

```
for I = 1 to 10 do
    ...
end for
```

**Flow chart**




**Assembly Language**

```
FOR_INIT, LOAD ONE
          STORE I
FOR_COND, LOAD I
          SUBT TEN
          SKIPCOND 800
          JUMP FOR_BODY
          JUMP END_FOR
 FOR_BODY,

          LOAD I
          ADD ONE
          STORE I
          JUMP FOR_COND
 END_FOR,
```

If I ≤ 10 is False, then (I - 10) is positive, so the SKIPCOND 800 skips to JUMP END_FOR. Thus, dropping out of the FOR loop. Otherwise, the JUMP FOR_BODY is not skipped. After the for-body executes and the loop-control variable I is incremented, the JUMP FOR_COND loops back to recheck the loop control variable.

The simplicity of the MARIE instruction set make writing assembly-language programs difficult. So, we'll only write small toy programs in MARIE, and later learn to write realistic assembly-language programs in the slightly more complex MIPS instruction set. However, the simplicity of the MARIE architecture is a huge benefit as we turn our attention to the hardware of implementing the CPU datapath and control unit.

## MARIE Registers and Buses:

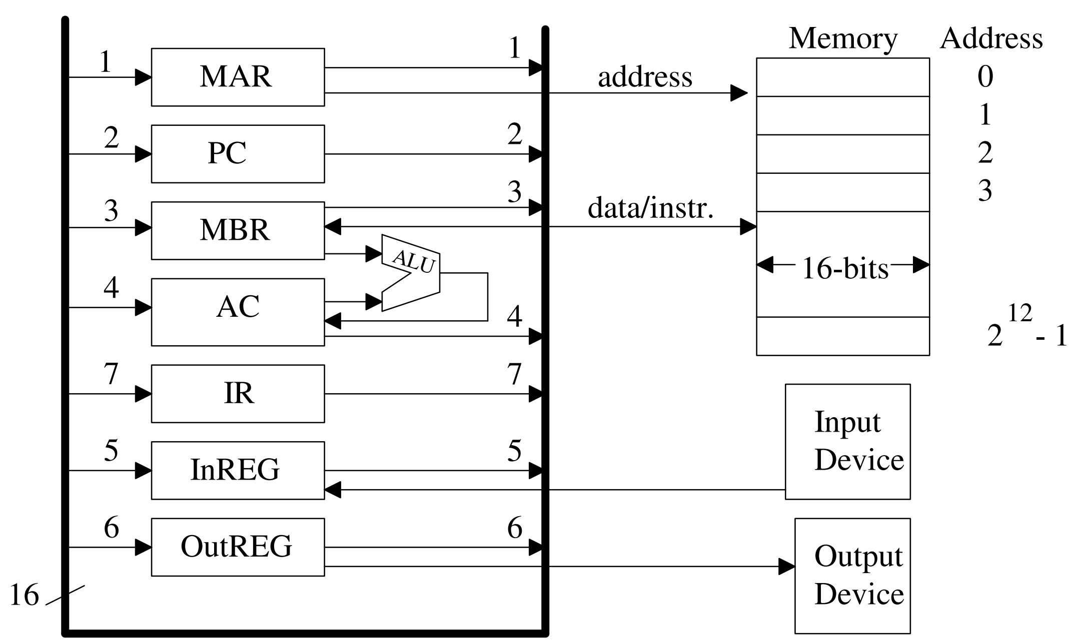
The revised Figure 4.9 (below) has moved the Memory from the CPU chip and hence the internal CPU Datapath. Thus, memory can only be accessed via the MAR (Memory-Address Register) and the MBR (Memory-Buffer Register) which is much more realistic. This has some impact on the microoperations that access memory. For example, fetching the instruction pointed at by the PC into the IR would require the following microoperations:

MAR ← PC
MBR ← M[MAR] (read from memory into the MBR instead of directly into the IR as descibed on page 199)
IR ← MBR

However, the authors seem to understand this since their microoperations to execute the Load X (on page 196) use the MBR correctly:

MAR ← X (X is the address part of the IR, so this should technically be MAR ← $IR_{11-0}$ )
MBR ← M[MAR] (read from memory into the MBR instead of directly into the AC)
AC ← MBR

Revised Figure 4.9 Datapath in MARIE



## Supplement for Assignment #7 (sections 4.8 - 4.10 of the textbook)

The text discusses the microoperations of the fetch-decode-execute machine cycle in the execution of the "Simple Program" below that calculates RESULT = X + Y.

| Address | Label | Assembly Language | Machine Language |
|---|---|---|---|
| 100 | | LOAD X | $1104_{16}$ |
| 101 | | ADD Y | $3105_{16}$ |
| 102 | | STORE RESULT | $2106_{16}$ |
| 103 | | HALT | $7000_{16}$ |
| 104 | X, | DEC 35 | $0023_{16}$ |
| 105 | Y, | DEC -23 | $FFE9_{16}$ |
| 106 | RESULT, | DEC 0 | $0000_{16}$ |

### Revised Figure 4.14 (a) LOAD X ($1104_{16}$ in ML)

| Step | Step # | RTN | PC | IR | MAR | MBR | AC |
|---|---|---|---|---|---|---|---|
| (initial values) | | | 100 | | | | |
| Fetch | $T_0$ | MAR ← PC | 100 | | 100 | | |
| | $T_1$ | MBR ← M[MAR] | 100 | | 100 | 1104 | |
| | $T_2$ | IR ← MBR | 100 | 1104 | 100 | 1104 | |
| | $T_3$ | PC ← PC + 1 | 101 | 1104 | 100 | 1104 | |
| Decode IR[15-12] | $T_4$ | MAR ← IR[11-0] | 101 | 1104 | 104 | 1104 | |
| Get operand | $T_5$ | MBR ← M[MAR] | 101 | 1104 | 104 | 0023 | |
| Execute | $T_6$ | AC ← MBR | 101 | 1104 | 104 | 0023 | 0023 |

### Revised Figure 4.14 (b) ADD Y ($3105_{16}$ in ML)

| Step | Step # | RTN | PC | IR | MAR | MBR | AC |
|---|---|---|---|---|---|---|---|
| (initial values AFTER LOAD X) | | | 101 | 1104 | 104 | 0023 | 0023 |
| Fetch | $T_0$ | MAR ← PC | 101 | 1104 | 101 | 0023 | 0023 |
| | $T_1$ | MBR ← M[MAR] | 101 | 1104 | 101 | 3105 | 0023 |
| | $T_2$ | IR ← MBR | 101 | 3105 | 101 | 3105 | 0023 |
| | $T_3$ | PC ← PC + 1 | 102 | 3105 | 101 | 3105 | 0023 |
| Decode IR[15-12] | $T_4$ | MAR ← IR[11-0] | 102 | 3105 | 105 | 3105 | 0023 |
| Get operand | $T_5$ | MBR ← M[MAR] | 102 | 3105 | 105 | FFE9 | 0023 |
| Execute | $T_6$ | AC ← AC + MBR | 102 | 3105 | 105 | FFE9 | 000C |

### Revised Figure 4.14 (c) STORE RESULT ($2106_{16}$ in ML)
### (YOU COMPLETE THIS AS PART OF LECTURE)

| Step | Step # | RTN | PC | IR | MAR | MBR | AC |
|---|---|---|---|---|---|---|---|
| (initial values AFTER ADD Y) | | | 102 | 3105 | 105 | FFE9 | 000C |
| Fetch | $T_0$ | | | | | | |
| | $T_1$ | | | | | | |
| | $T_2$ | | | | | | |
| | $T_3$ | | | | | | |
| Decode IR[15-12] | $T_4$ | | | | | | |
| Execute* | $T_5$ | | | | | | |

* "Get Operand" step is not necessary for STORE instructions

# Supplement for Assignment #7 (sections 4.8 - 4.10 of the textbook)

**Advanced MARIE Assembly Language Example:** Print null terminated string to output

**HLL:**
```
index = 0
while str[index] != 0 do
    output str[index]
    index = index + 1
end while
```

| Address | Label | Assembly Language | | Machine Language |
|---|---|---|---|---|
| 0 | | CLEAR | | $A000_{16}$ |
| 1 | | STORE INDEX | | $2011_{16}$ |
| 2 | WHILE, | LOAD STR_BASE | | $1013_{16}$ |
| 3 | | ADD INDEX | | $3011_{16}$ |
| 4 | | STORE ADDR | | $2012_{16}$ |
| 5 | | CLEAR | | $A000_{16}$ |
| 6 | | ADDI ADDR | | $B012_{16}$ |
| 7 | | SKIPCOND 400 | | $8400_{16}$ |
| 8 | | JUMP DO | | $900A_{16}$ |
| 9 | | JUMP END_WHILE | | $900A_{16}$ |
| A | DO, | OUTPUT | | $6000_{16}$ |
| B | | LOAD INDEX | | $100D_{16}$ |
| C | | ADD ONE | | $300B_{16}$ |
| D | | STORE INDEX | | $2011_{16}$ |
| E | | JUMP WHILE | | $9002_{16}$ |
| F | END_WHILE, | HALT | | $7000_{16}$ |
| 10 | ONE, | DEC 1 | | $0001_{16}$ |
| 11 | INDEX, | DEC 0 | | $0000_{16}$ |
| 12 | ADDR, | HEX 0 | | $0000_{16}$ |
| 13 | STR_BASE, | HEX 14 | | $0014_{16}$ |
| 14 | STR, | DEC 72 | / H | $0048_{16}$ |
| 15 | | DEC 69 | / E | $0045_{16}$ |
| 16 | | DEC 76 | / L | $004C_{16}$ |
| 17 | | DEC 76 | / L | $004C_{16}$ |
| 18 | | DEC 79 | / O | $004F_{16}$ |
| 19 | | DEC 13 | /carriage return | $000D_{16}$ |
| 1A | | DEC 87 | / W | $0057_{16}$ |
| 1B | | DEC 79 | / O | $004F_{16}$ |
| 1C | | DEC 82 | / R | $0052_{16}$ |
| 1D | | DEC 76 | / L | $004C_{16}$ |
| 1 | | DEC 68 | / D | $0044_{16}$ |
| 1F | NULL, | DEC 0 | / NULL CHAR | $0000_{16}$ |