Computer Architecture
(CSC-3501)

Lecture 13
(28 Feb 2008)

Seung-Jong Park (Jay)
http://www.csc.lsu.edu/~sjpark

Announcement

4.10 A Simple Program

- Consider the simple MARIE program given below. We show a set of mnemonic instructions stored at addresses 100 - 106 (hex):

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

4.10 A Simple Program

- Let's look at what happens inside the computer when our program runs.
- This is the **`LOAD 104`** instruction:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ← PC	100	------	100	------	------
	IR ← M[MAR]	100	1104	100	------	------
	PC ← PC + 1	101	1104	100	------	------
Decode	MAR ← IR[11-0]	101	1104	104	------	------
	(Decode IR[15-12])	101	1104	104	------	------
Get operand	MBR ← M[MAR]	101	1104	104	0023	------
Execute	AC ← MBR	101	1104	104	0023	0023

4.10 A Simple Program

- Our second instruction is **`ADD 105`**:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11-0]	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

4.11 A Discussion on Assemblers

- Mnemonic instructions, such as **`LOAD 104`**, are easy for humans to write and understand.
- They are impossible for computers to understand.
- *Assemblers* translate instructions that are comprehensible to humans into the machine language that is comprehensible to computers
  - We note the distinction between an assembler and a compiler: In assembly language, there is a one-to-one correspondence between a mnemonic instruction and its machine code. With compilers, this is not usually the case.

4.11 A Discussion on Assemblers

- Assemblers create an *object program file* from mne monic *source code* in two passes.
- During the first pass, the assembler assembles as much of the program is it can, while it builds a *symb ol table* that contains memory references for all sym bols in the program.
- During the second pass, the instructions are comple ted using the values from the symbol table.

4.11 A Discussion on Assemblers

- Consider our example progr am (top).
  - Note that we have included tw o directives **HEX** and **DEC** that specify the radix of the consta nts.
- During the first pass, we hav e a symbol table and the par tial instructions shown at the bottom.

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

X	104
Y	105
Z	106

1	X
3	Y
2	Z
7 0 0 0	

4.11 A Discussion on Assemblers

- After the second pass, the assembly is complete.

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0

X	104
Y	105
Z	106

4.12 Extending Our Instruction Set

- So far, all of the MARIE instructions that we have di scussed use a *direct addressing mode*.
- This means that the address of the operand is expli citly stated in the instruction.
- It is often useful to employ a *indirect addressing*, wh ere the address of the address of the operand is giv en in the instruction.
  - If you have ever used pointers in a program, you are already familiar with indirect addressing.

4.12 Extending Our Instruction Set

- To help you see what happens at the machine level , we have included an indirect addressing mode inst ruction to the MARIE instruction set.
- The **ADDI** instruction specifies the address of the a ddress of the operand. The following RTL tells us w hat is happening at the register level:

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← AC + MBR
```


4.12 Extending Our Instruction Set

- Another helpful programming tool is the use of subr outines.
- The jump-and-store instruction, **JNS**, gives us limite d subroutine functionality. The details of the **JNS** ins truction are given by the following RTL:

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
AC ← PC
```

Does **JNS** permit recursive calls?

4.12 Extending Our Instruction Set

- Our last helpful instruction is the **`CLEAR`** instruction.
- All it does is set the contents of the accumulator to a ll zeroes.
- This is the RTL for **`CLEAR`**:

  AC ← 0

- We put our new instructions to work in the program on the following slide.

4.12 Extending Our Instruction Set

```
100 |       LOAD Addr          10E |       SKIPCOND 000
101 |       STORE Next         10F |       JUMP Loop
102 |       LOAD Num           110 |       HALT
103 |       SUBT One           111 |Addr   HEX 118
104 |       STORE Ctr          112 |Next   HEX 0
105 |Loop   LOAD Sum           113 |Num    DEC 5
106 |       ADDI Next          114 |Sum    DEC 0
107 |       STORE Sum          115 |Ctr    HEX 0
108 |       LOAD Next          116 |One    DEC 1
109 |       ADD One            117 |       DEC 10
10A |       STORE Next         118 |       DEC 15
10B |       LOAD Ctr           119 |       DEC 2
10C |       SUBT One           11A |       DEC 25
10D |       STORE Ctr          11B |       DEC 30
```