Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

MARIE Simulator

Schedule

- **Today** – More assembly programming
- **Next Tuesday**
  - Meet in KH 203 (normal classroom)
  - MARIE instruction decoding hardware
  - Plus **Quiz 3!**
    - Quiz 3 topic: Assembly programming!
    - *I will give you Table 4.7 from the book…*
- **Next Thursday** – Begin Chapter 5
  - Closer look at instruction sets

MARIE Programming

- Writing code in assembly can be very intimidating at first (or always)
- High-level languages are easier to code in because
  - You're more comfortable with them
  - They take fewer steps
- Treat the problem like a complier would
  - Think about how to code in another language, like C++
  - Convert each construct into a series of MARIE instructions

MARIE Programming

- What was easy about programming MARIE?
- What was hard about programming MARIE?
- What tips would you share to other beginning assembly programmers?

Assembly Pitfall – Program Organization

- The assembler and CPU do exactly what we tell it.
  - However, that doesn’t always mean it does what we intend!
- You must be very explicit when organizing your program

Assembly Pitfall – Program Organization

- **What did the programmer intend for this code to do?**
  - Z=X+Y
- **What does the program actually do?**
  - Our variable X (0006) gets interpreted as a JNS 6 instruction!

```
        LOAD  X
        ADD   Y
X,      DEC   6
Y,      DEC   -3
Z       DEC   0
        STORE   Z
        HALT
```

Assembly Pitfall – Program Organization

- **Code should always jump around any data**
- Compilers did this for you in COMP 51
  - Automatically separate code and data

```
        LOAD  X
        ADD   Y
        JMP   SKIP
X,      DEC   6
Y,      DEC   -3
Z       DEC   0
SKIP,   STORE  Z
        HALT
```

Clever Tricks

Clever Tricks – Memory Addresses

```
      ORG 100            / Example 4.1
      Load   Addr        /Load address of first number to be added
      Store  Next        /Store this address is our Next pointer
      Load   Num         /Load the number of items to be added
      Subt   One         /Decrement
      Store  Ctr         /Store this value in Ctr to control looping
Loop, Load   Sum         /Load the Sum into AC
      AddI   Next        /Add the value pointed to by location Next
      Store  Sum         /Store this sum
      Load   Next        /Load Next
      Add    One         /Increment by one to point to next address
      Store  Next        /Store in our pointer Next
      Load   Ctr         /Load the loop control variable
      Subt   One         /Subtract one from the loop control variable
      Store  Ctr         /Store this new value in loop control variable
      Skipcond 000       /If control variable < 0, skip next instruction
      Jump   Loop        /Otherwise, go to Loop
      Halt               /Terminate program
Addr, Hex    117         /Numbers to be summed start at location 117
Next, Hex    0           /A pointer to the next number to add
Num,  Dec    5           /The number of values to add
Sum,  Dec    0           /The sum
Ctr,  Hex    0           /The loop control variable
One,  Dec    1           /Used to increment and decrement by 1
      Dec    10          /The values to be added together
      Dec    15
      Dec    20
      Dec    25
      Dec    30
```

This is location 117

- Think back to first MARIE program (Example 4.1)
- ADDR variable holds the address of the element to be added
  - It's a pointer
  - Value = 0x117

Clever Tricks – Memory Addresses

- **What happens if the program changes?**
  - Say, we add a few instructions

- We would have to find the *new* starting address of the data, save its value in our assembly code, and re-run the assembler
  - **Annoying!**

```
ADDR, HEX 125
```

Clever Tricks – Memory Addresses

- Perfect world: The assembler lets us use a label and fills in the address from the symbol table on pass 2
  - Unfortunately we don't live in this perfect world!
  - Assembler doesn't do this directly, but we can fool it into doing something similarly useful
- Solution takes advantage of the fact that the instruction format always uses the lower 12 bits of each instruction for the address
  - i.e. this solution works for MARIE, but not necessarily other assembly languages

Clever Tricks – Memory Addresses

- **What happens when these changes are made to the example program?**

```
100              Load    Addr
101              Store   Next
…
106   Addi       Next
…
111   Addr,      JnS   List
112   Next,      Hex   0
…
116   One,       Dec   1
117   List,      Dec   10
```

Clever Tricks – Memory Addresses

- Recall the RTL for the ADDI instruction:

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

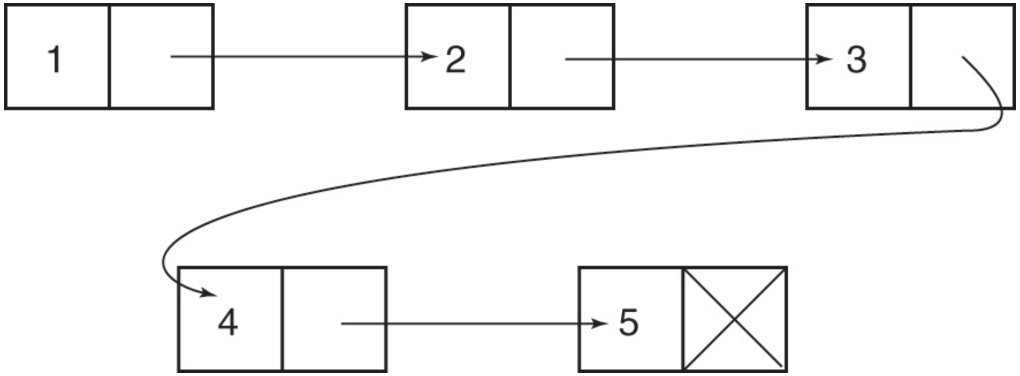
- The third operation truncates the opcode
  - MBR = 16 bits, but MAR = 12 bits
- Result: MAR ends up with only the address of LIST

Clever Tricks – Memory Addresses

- This trick works with all instructions
  - `JnS` is safest since its opcode is 0
  - The resulting value placed in memory is **only** the 12-bit address

Homework 4.33 Tips

- Either use the JNS trick just shown, *or* write your program in two passes
  - Pass 1: Write the code the traverses the linked list
  - Run the assembler and look at memory addresses where it placed your program
  - Pass 2: Update the linked list memory addresses based on the assembler listing file produced



Clever Tricks – Faking LOADI

- MARIE has LOAD-Indirect (LOADI) and STORE-Indirect (STOREI) instructions
  - But clever programmers don't need them!
- **How could I "emulate" the `LOADI X` instruction using several non-indirect MARIE instructions?**

```
CLEAR      / Put 0 in AC
ADDI X     / Add indirect value from Mem[Mem[X]]
```

Clever Tricks – Faking STOREI

- **How could I "emulate" the `STOREI X` instruction using several non-indirect MARIE instructions?**
  - This is harder!
- Idea: Take advantage of the stored program concept
  - Instructions are just data
- We need a sequence of instructions that construct a STORE instruction with the desired address
- This would be a good application of a subroutine
  - Pass the value to store in AC, place the address in a parameter variable

Clever Tricks – Faking STOREI

- Equivalent code to `STOREI X`:

```
            LOAD     STROPCODE   / Get opcode
            ADD      X           / Combine addr
            STORE    STOREI      / Save
STOREI,     HEX      0     / Data: build instruction
                           / here, then execute it
            / Program continues here...
...
...
STROPCODE,  STORE    0     / Data: Just opcode
                           / for store
```

Subroutines

Subroutines

- `Result = addOne(input1);`
- **What do we need for a subroutine? (i.e. function)**
  - Arguments to the function (i.e. input data)
  - Return value from the function
  - A way to jump to the function
  - A way to return from the function when finished

Subroutine Example

```
      Load  Data         / get  value
      Store Arg1         / store value as argument
      Jns AddOne         / call subroutine
      Load Return        / load subroutine return data
      Output             / print it!
      Halt               / terminate
Data, Dec 20             / original value
/ ** Subroutine **
AddOne,  Dec 0           / return address placed here
      Load  Arg1         / get argument
      Add   One          / increment it
      Store Return       / save return value
      JumpI AddOne       / return with value in a
Arg1, Dec 0              / Empty: subroutine argument
Return, Dec 0            / Empty: subroutine return value
```

Subroutine Example

- Write a subroutine which calculates the length of a null-terminated ASCII string
  - **What is an ASCII string?**
  - **What is null terminated?**

Subroutine Example

- Write a subroutine which calculates the length of a null-terminated ASCII string

```
unsigned int strlen(char *str)
{
     unsigned int len = 0;
     while (*str != 0)
     {
         str++;
         len++;
     }
     return len;
}
```

Subroutine Example

- This subroutine needs:
  - A parameter which is the starting address of the string
  - A variable to keep track of string length
  - A loop which terminates when a 0 is found
  - Inside the loop, increment the pointer and the length count
  - To return the length variable

Lab Exercises

- Pick <u>**one**</u> of these, and **demonstrate it** by the end of lab
  - Need to provide a sample "main program" that includes calling the subroutine

1. Write a **subroutine** which searches an array for a particular value
   - Inputs: number, starting address of array, size of array
   - Output: index of first match, -1 if not found
2. Write a **subroutine** which makes a copy of a *null-terminated string*
   - Inputs: addresses of source and destination strings