Introduction to Computer Engineering

CS/ECE 252, Spring 2017
Rahul Nayar
Computer Sciences Department
University of Wisconsin – Madison
Chapter 6
Programming
Announcements

We are here

- Electronic circuits
  - ECE340
- Digital Design
  - CS/ECE352
- Computer Architecture
  - CS/ECE552
- Machine Language (ISA)
  - CS/ECE354
- Compiler
  - CS536
- Operating System
  - CS537
- Application Program
  - CS302
- Computer Architecture
  - CS/ECE552
- Digital Design
  - CS/ECE352
- Electronic circuits
  - ECE340
Solving Problems using a Computer

Methodologies for creating computer programs that perform a desired function.

Problem Solving
• How do we figure out what to tell the computer to do?
• Convert problem statement into algorithm, using *stepwise refinement*.
• Convert algorithm into LC-3 machine instructions.

Debugging
• How do we figure out why it didn’t work?
• Examining registers and memory, setting breakpoints, etc.

*Time spent on the first can reduce time spent on the second!*
### In-class Exercise: Fill in the Instructions and Comments in Table below

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3003</td>
<td>R3 ← M[R3 + 0x0010]</td>
<td></td>
</tr>
<tr>
<td>0x3004</td>
<td>If Z or P, goto 0x2FDF</td>
<td></td>
</tr>
<tr>
<td>0x3005</td>
<td>1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0x3006</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1</td>
<td>HALT (TRAP x25)</td>
</tr>
<tr>
<td>0x3007</td>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0x4000</td>
</tr>
</tbody>
</table>
### In-class Exercise Solutions

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3003</td>
<td>LDR R3, R3, 0x0010</td>
<td>$R3 \leftarrow M[R3 + 0x0010]$</td>
</tr>
<tr>
<td>0x3004</td>
<td>BRzp N Z P 0x2FDF</td>
<td>If Z or P, goto 0x2FDF</td>
</tr>
</tbody>
</table>
| 0x3005  | LDI R5, 0x0001 | $R5 \leftarrow \text{mem[mem[0x3007]]}$  
                        $R5 \leftarrow \text{mem[0x4000]}$ |
| 0x3006  | TRAP 0x25 | HALT (TRAP x25) |
| 0x3007  | 0x4000 |  |

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3003</td>
<td>LDR R3, R3, 0x0010</td>
<td>$R3 \leftarrow M[R3 + 0x0010]$</td>
</tr>
<tr>
<td>0x3004</td>
<td>BRzp N Z P 0x2FDF</td>
<td>If Z or P, goto 0x2FDF</td>
</tr>
</tbody>
</table>
| 0x3005  | LDI R5, 0x0001 | $R5 \leftarrow \text{mem[mem[0x3007]]}$  
                        $R5 \leftarrow \text{mem[0x4000]}$ |
| 0x3006  | TRAP 0x25 | HALT (TRAP x25) |
| 0x3007  | 0x4000 |  |
Stepwise Refinement
Also known as systematic decomposition.

Start with problem statement:

“We wish to count the number of occurrences of a character in a file. The character in question is to be input from the keyboard; the result is to be displayed on the monitor.”

Decompose task into a few simpler subtasks.

Decompose each subtask into smaller subtasks, and these into even smaller subtasks, etc.... until you get to the machine instruction level.
Problem Statement

Because problem statements are written in English, they are sometimes ambiguous and/or incomplete.

- Where is “file” located? How big is it, or how do I know when I’ve reached the end?
- How should final count be printed? A decimal number?
- If the character is a letter, should I count both upper-case and lower-case occurrences?

How do you resolve these issues?

- Ask the person who wants the problem solved, or
- Make a decision and document it.
Three Basic Constructs

There are three basic ways to decompose a task:

**Sequential**
- Task
- Subtask 1
- Subtask 2

**Conditional**
- Task
- Test condition
  - True
    - Subtask 1
  - False
    - Subtask 2

**Iterative**
- Task
- Test condition
  - True
    - Subtask
  - False
    - Subtask
Sequential

Do Subtask 1 to completion, then do Subtask 2 to completion, etc.

- Get character input from keyboard
- Examine file and count the number of characters that match
- Count and print the occurrences of a character in a file
- Print number to the screen
Conditional

If condition is true, do Subtask 1; else, do Subtask 2.

Test character.
If match, increment counter.

file char = input?

True
Count = Count + 1
False
Iterative

Do Subtask over and over, as long as the test condition is true.

Check each element of the file and count the characters that match.

Check next char and count if matches.

more chars to check?

True

Check next char and count if matches.

False
Note:

- HW4 is due today (Remember to Staple)
- No class on Friday (March 3rd)
- Discussion Session for Mid-Term 2 on Monday (March 6th)
- Mid Term 2: Chapt 3 – Chapt 5
- Useful tip for Mid Term 2 Preparation:
  - Start Solving HW 5
- HW 5 is due on Monday (March 13th)
- Start Working on PenSim Simulator:
  - Instructions in the “computing” section of course website
  - Start Working on the scripts uploaded in the course website
Points Covered So Far…:

• Methodologies to write computer programs.
  • Problem Solving: Stepwise Refinement
  • Debugging

• Three ways to decompose tasks:
  • Sequential
  • Iterative
  • Conditional
Problem Solving Skills


• Like a puzzle, or a “word problem” from grammar school math.
  ➢ What is the starting state of the system?
  ➢ What is the desired ending state?
  ➢ How do we move from one state to another?

• Recognize English words that correlate to three basic constructs:
  ➢ “do A then do B” ⇒ sequential
  ➢ “if G, then do H” ⇒ conditional
  ➢ “for each X, do Y” ⇒ iterative
  ➢ “do Z until W” ⇒ iterative
LC-3 Control Instructions

How do we use LC-3 instructions to encode the three basic constructs?

**Sequential**
- Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.

**Conditional and Iterative**
- Create code that converts condition into N, Z, or P.
  
  **Example:**
  
  Condition: “Is R0 = R1?”
  
  Code: Subtract R1 from R0; if equal, Z bit will be set.

  - Then use BR instruction to transfer control to the proper subtask.
Code for Conditional

Assuming all addresses are close enough that PC-relative branch can be used.
Code for Iteration

Assuming all addresses are on the same page.
Example: Counting Characters

Input a character. Then scan a file, counting occurrences of that character. Finally, display on the monitor the number of occurrences of the character (up to 9).

START

A
- Initialize: Put initial values into all locations that will be needed to carry out this task.
  - Input a character.
  - Set up a pointer to the first location of the file that will be scanned.
  - Get the first character from the file.
  - Zero the register that holds the count.

B
- Scan the file, location by location, incrementing the counter if the character matches.

C
- Display the count on the monitor.

STOP

Initial refinement: Big task into three sequential subtasks.
Refining B

Scan the file, location by location, incrementing the counter if the character matches.

Test character. If a match, increment counter. Get next character.

Refining B into iterative construct.
Refining B1

Refining B1 into sequential subtasks.

Test character. If a match, increment counter. Get next character.

B1

Test character. If matches, increment counter.

B2

Get next character.

B3

Done?

Yes

No

Refining B1 into sequential subtasks.
Refining B2 and B3

Conditional (B2) and sequential (B3). Use of LC-2 registers and instructions.
The Last Step: LC-3 Instructions

Use comments to separate into modules and to document your code.

; Look at each char in file.
0001100001111100 ; is R1 = EOT?
0000010xxxxxxxxxx ; if so, exit loop
; Check for match with R0.
1001001001111111 ; R1 = -char
0001001001100001
0001001000000001 ; R1 = R0 - char
0000101xxxxxxxxxx ; no match, skip incr
0001010010100001 ; R2 = R2 + 1
; Incr file ptr and get next char
0001011011100001 ; R3 = R3 + 1
0110001011000000 ; R1 = M[R3]

Don't know PCoffset bits until all the code is done
Debugging
You’ve written your program and it doesn’t work. Now what?

What do you do when you’re lost in a city?
- Drive around randomly and hope you find it?
- Return to a known point and look at a map?

In debugging, the equivalent to looking at a map is *tracing* your program.
- Examine the sequence of instructions being executed.
- Keep track of results being produced.
- Compare result from each instruction to the *expected* result.
Debugging Operations

Any debugging environment should provide means to:

1. Display values in memory and registers.
2. Deposit values in memory and registers.
3. Execute instruction sequence in a program.
4. Stop execution when desired.

Different programming levels offer different tools.

- High-level languages (C, Java, ...) usually have source-code debugging tools.
- For debugging at the machine instruction level:
  - simulators
  - operating system “monitor” tools
LC-3 Simulator (PennSim)

- execute instruction sequences
- set/display registers
- set breakpoints
- set/display memory
Types of Errors

Syntax Errors

• You made a typing error that resulted in an illegal operation.
• Not usually an issue with machine language, because almost any bit pattern corresponds to some legal instruction.
• In high-level languages, these are often caught during the translation from language to machine code.

Logic Errors

• Your program is legal, but wrong, so the results don’t match the problem statement.
• Trace the program to see what’s really happening and determine how to get the proper behavior.

Data Errors

• Input data is different than what you expected.
• Test the program with a wide variety of inputs.
Tracing the Program

Execute the program one piece at a time, examining register and memory to see results at each step.

Single-Stepping
- Execute one instruction at a time.
- Tedious, but useful to help you verify each step of your program.

Breakpoints
- Tell the simulator to stop executing when it reaches a specific instruction.
- Check overall results at specific points in the program.
  - Lets you quickly execute sequences to get a high-level overview of the execution behavior.
  - Quickly execute sequences that your believe are correct.
In-class Exercise

LC-3 does not have a multiply instruction. Use step-wise refinement to multiple the contents of R4 and R5, and store the product/results in R2.

Quote of the Day:
“Computers are good at following instructions, but not at reading your mind.” – Donald Knuth
Example 1: Multiply

This program is supposed to multiply the two unsigned integers in R4 and R5.

Set R4 = 10, R5 =3.
Run program.
Result: R2 = 40, not 30.
Debugging the Multiply Program

<table>
<thead>
<tr>
<th>PC</th>
<th>R2</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3200</td>
<td>--</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>x3201</td>
<td>0</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>x3202</td>
<td>10</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>x3203</td>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>x3201</td>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>x3202</td>
<td>20</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>x3203</td>
<td>20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>x3203</td>
<td>20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>x3201</td>
<td>20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>x3202</td>
<td>30</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>x3203</td>
<td>30</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>x3201</td>
<td>30</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>x3202</td>
<td>40</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>x3203</td>
<td>40</td>
<td>10</td>
<td>-1</td>
</tr>
<tr>
<td>x3203</td>
<td>40</td>
<td>10</td>
<td>-1</td>
</tr>
</tbody>
</table>

Single-stepping
Breakpoint at branch (x3203)

Executing loop one time too many.
Branch at x3203 should be based on P bit only, not Z and P.

PC and registers at the beginning of each instruction
Example 2: Summing an Array of Numbers

This program is supposed to sum the numbers stored in 10 locations beginning with x3100, leaving the result in R1.

Address Instruction | Comment
--- | ---
00000000000110000000 | R1 ← R1 AND #0
010110010010000000 | R4 ← R4 AND #0
000110010010101010 | R4 ← R4 + #10
0010010011111110 | R2 ← M[3100]
0110011010000000 | R3 ← M[R2 + #0]
0001010010100001 | R2 ← R2 + #1
0001100100111111 | R4 ← R4 + # -1
BRp x3004 | HALT
Debugging the Summing Program

Running the the data below yields $R1 = \text{x0024}$, but the sum should be $\text{x8135}$. What happened?

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3100</td>
<td>x3107</td>
</tr>
<tr>
<td>x3101</td>
<td>x2819</td>
</tr>
<tr>
<td>x3102</td>
<td>x0110</td>
</tr>
<tr>
<td>x3103</td>
<td>x0310</td>
</tr>
<tr>
<td>x3104</td>
<td>x0110</td>
</tr>
<tr>
<td>x3105</td>
<td>x1110</td>
</tr>
<tr>
<td>x3106</td>
<td>x11B1</td>
</tr>
<tr>
<td>x3107</td>
<td>x0019</td>
</tr>
<tr>
<td>x3108</td>
<td>x0007</td>
</tr>
<tr>
<td>x3109</td>
<td>x0004</td>
</tr>
</tbody>
</table>

Start single-stepping program...

<table>
<thead>
<tr>
<th>PC</th>
<th>R1</th>
<th>R2</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>x3001</td>
<td>0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>x3002</td>
<td>0</td>
<td>--</td>
<td>0</td>
</tr>
<tr>
<td>x3003</td>
<td>0</td>
<td>--</td>
<td>10</td>
</tr>
<tr>
<td>x3004</td>
<td>0</td>
<td>x3107</td>
<td>10</td>
</tr>
</tbody>
</table>

Loading **contents** of $M[x3100]$, not address. Change opcode of x3003 from 0010 (LD) to 1110 (LEA).

Should be x3100!
Example 3: Looking for a 5

This program is supposed to set R0=1 if there’s a 5 in one ten memory locations, starting at x3100. Else, it should set R0 to 0.

R0 = 1, R1 = -5, R3 = 10
R4 = x3100, R2 = M[R4]

R2 = 5?
Yes
R0 = 1

No
R3 = 0?
Yes
R3 = R3 - 1
R2 = M[R4]

No
R0 = 0

HALT

R0 = 1, R1 = -5, R3 = 10
R4 = x3100, R2 = M[R4]

x3000 0101000000100000
x3001 0001000000100001
x3002 0101001001100000
x3003 0001001001111011
x3004 0101011011100000
x3005 0001011011110101
x3006 001010000001001
x3007 0110010100000000
x3008 0010100100000001
x3009 0000100000000101
x300A 0011001001000001
x300B 0001011011111111
x300C 0110010100000000
x300D 0000001111111101
x300E 0101000000100000
x300F 1111000000100101
x3010 0011000100000000
Debugging the Fives Program

Running the program with a 5 in location x3108 results in \( R0 = 0 \), not \( R0 = 1 \). What happened?

Perhaps we didn’t look at all the data? Put a breakpoint at x300D to see how many times we branch back.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3100</td>
<td>9</td>
</tr>
<tr>
<td>x3101</td>
<td>7</td>
</tr>
<tr>
<td>x3102</td>
<td>32</td>
</tr>
<tr>
<td>x3103</td>
<td>0</td>
</tr>
<tr>
<td>x3104</td>
<td>-8</td>
</tr>
<tr>
<td>x3105</td>
<td>19</td>
</tr>
<tr>
<td>x3106</td>
<td>6</td>
</tr>
<tr>
<td>x3107</td>
<td>13</td>
</tr>
<tr>
<td>x3108</td>
<td>5</td>
</tr>
<tr>
<td>x3109</td>
<td>61</td>
</tr>
</tbody>
</table>

PC | R0 | R2 | R3 | R4  
---|----|----|----|-----
\text{x300D} | 1   | 7  | 9  | x3101 |
\text{x300D} | 1   | 32 | 8  | x3102 |
\text{x300D} | 1   | 0  | 7  | x3103 |
\text{0}    | 0   | 7  | x3103 |

Branch uses condition code set by loading R2 with \( M[R4] \), not by decrementing R3. Swap x300B and x300C, or remove x300C and branch back to x3007.
In-class Exercise:
Design an algorithm for the following problem:
Finding First 1 in a Word

This program is supposed to return (in R1) the bit position of the first 1 in a word. The address of the word is in location x3009 (just past the end of the program). If there are no ones, R1 should be set to –1.

Quote of the Day:
“Intelligence is not what we know, but what we do when we don't know.”
-- Jean Piaget (1896-1980)
Example 4: Finding First 1 in a Word

This program is supposed to return (in R1) the bit position of the first 1 in a word. The address of the word is in location x3009 (just past the end of the program). If there are no ones, R1 should be set to –1.

R1 = 15
R2 = data

R2[15] = 1?

Yes

x3000 0101001001100000
x3001 0001001001101111
x3002 1010010000000110
x3003 0000100000000100
x3004 0001001001111111
x3005 0001010010000010
x3006 0000100000000001
x3007 0000111111111100
x3008 1111000000100101
x3009 0011000100000000

No

decrement R1

shift R2 left one bit

R2[15] = 1?

Yes

x3000 0101001001100000
x3001 0001001001101111
x3002 1010010000000110
x3003 0000100000000100
x3004 0001001001111111
x3005 0001010010000010
x3006 0000100000000001
x3007 0000111111111100
x3008 1111000000100101
x3009 0011000100000000

No
Debugging the First-One Program

Program works most of the time, but if data is zero, it never seems to HALT.

Breakpoint at backwards branch (x3007)

<table>
<thead>
<tr>
<th>PC</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3007</td>
<td>14</td>
</tr>
<tr>
<td>x3007</td>
<td>13</td>
</tr>
<tr>
<td>x3007</td>
<td>12</td>
</tr>
<tr>
<td>x3007</td>
<td>11</td>
</tr>
<tr>
<td>x3007</td>
<td>10</td>
</tr>
<tr>
<td>x3007</td>
<td>9</td>
</tr>
<tr>
<td>x3007</td>
<td>8</td>
</tr>
<tr>
<td>x3007</td>
<td>7</td>
</tr>
<tr>
<td>x3007</td>
<td>6</td>
</tr>
<tr>
<td>x3007</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3007</td>
<td>4</td>
</tr>
<tr>
<td>x3007</td>
<td>3</td>
</tr>
<tr>
<td>x3007</td>
<td>2</td>
</tr>
<tr>
<td>x3007</td>
<td>1</td>
</tr>
<tr>
<td>x3007</td>
<td>0</td>
</tr>
<tr>
<td>x3007</td>
<td>-1</td>
</tr>
<tr>
<td>x3007</td>
<td>-2</td>
</tr>
<tr>
<td>x3007</td>
<td>-3</td>
</tr>
<tr>
<td>x3007</td>
<td>-4</td>
</tr>
<tr>
<td>x3007</td>
<td>-5</td>
</tr>
</tbody>
</table>

If no ones, then branch to HALT never occurs!
This is called an “infinite loop.”
Must change algorithm to either
(a) check for special case (R2=0), or
(b) exit loop if R1 < 0.
Debugging: Lessons Learned

Trace program to see what’s going on.
- Breakpoints, single-stepping

When tracing, make sure to notice what’s really happening, not what you think should happen.
- In summing program, it would be easy to not notice that address x3107 was loaded instead of x3100.

Test your program using a variety of input data.
- In Examples 3 and 4, the program works for many data sets.
- Be sure to test extreme cases (all ones, no ones, ...).
Backup Slides
LC-3 Simulator

- Execute instruction sequences
- Set/display registers and memory
- Stop execution, set breakpoints

The image shows a screenshot of the LC3 Simulator with various registers and memory values displayed. The simulator interface includes options to set breakpoints, which are indicated by red circles, and execute instruction sequences.