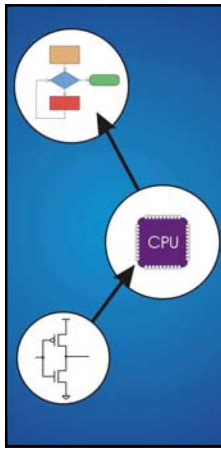
Introduction to Computer Engineering

ECE/CS 252, Fall 2010
Prof. Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin – Madison



Chapter 6
Part II: Debugging

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?

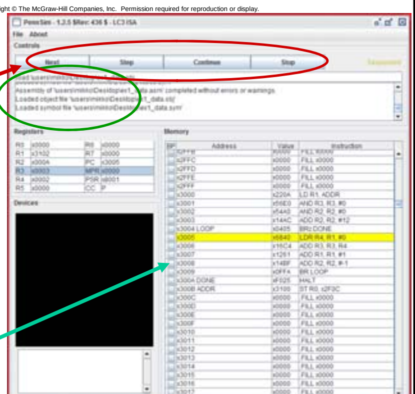
In debugging, the equivalent to looking at a map is *tracing* your program.
- Examine the sequence of instructions being executed.
- Keep track of results being produced.
- Compare result from each instruction to the ***expected*** result.

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:
  - Simulator
    - any universal computing device can emulate another UCD
  - operating system "monitor" tools
  - in-circuit emulators (ICE)
    - plug-in hardware replacements that give instruction-level control

PennSim Simulator



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

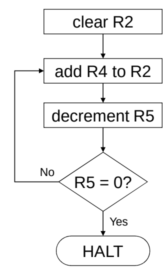
- **Tell the simulator to stop executing when it reaches a specific instruction.**
- **Check overall results at specific points in the program.**
  - **Lets you quickly execute sequences to get a high-level overview of the execution behavior.**
  - **Quickly execute sequences that your believe are correct.**

Watchpoints (not available in PennSim)

- **Tell the simulator to stop when a register or memory location changes or when it equals a specific value.**
- **Useful when you don't know <u>where</u> or <u>when</u> a value is changed.**

Example 1: Multiply

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




```
x3200 0101010010100000
x3201 0001010010000100
x3202 0001101101111111
x3203 0000011111111101
x3204 1111000000100101
```

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

Debugging the Multiply Program

PC and registers at the <u>beginning</u> of each instruction

Single-stepping

PC	R2	R4	R5
x3200	--	10	3
x3201	0	10	3
x3202	10	10	[illegible]
x3203	10	10	[illegible]
x3201	10	10	[illegible]
x3202	20	10	[illegible]
x3203	20	10	[illegible]
x3201	20	10	[illegible]
x3202	30	10	1
x3203	30	10	0
x3201	30	10	0
x3202	40	10	0
x3203	40	10	-1
x3204	40	10	-1
	40	10	-1

Breakpoint at branch (x3203)

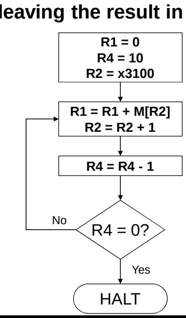
```
x3200 0101010010100000
x3201 0001010010000100
x3202 0001101101111111
x3203 0000011111111101
x3204 1111000000100101
```

Should stop looping here!

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

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.




```
x3000 0101001001100000
x3001 0101100100100000
x3002 0001100100101010
x3003 0010010011111100
x3004 0110011010000000
x3005 0001010010100001
x3006 0001001001000011
x3007 0001100100111111
x3008 0000001111111011
x3009 1111000000100101
```


Debugging the Summing Program

Running the the data below yields R1 = x0024, but the sum should be x8135. What happened?

Address	*Contents*
x3100	x3107
x3101	x2819
x3102	x0110
x3103	x0310
x3104	x0110
x3105	x1110
x3106	x11B1
x3107	x0019
x3108	x0007
x3109	x0004

Start single-stepping program...

PC	R1	R2	R4
x3000	--	--	--

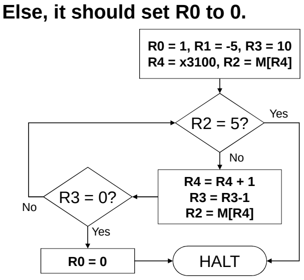
```
...
x3002 0001100100101010
x3003 0010010011111100
x3004 0110011010000000
...
```

Loading <u>contents</u> of M[x3100], not address.
Change opcode of x3003 from 0010 (LD) to 1110 (LEA).

Example 3: Looking for a 5

This program is supposed to set R0=1 if there's a 5 in one of ten memory locations, starting at x3100.

Else, it should set R0 to 0.




```
x3000 0101000000100000
x3001 0001000000100001
x3002 0101001001100000
x3003 0001001001111011
x3004 0101011011100000
x3005 0001011011101010
x3006 0010100000001001
x3007 0110010100000000
x3008 0001010010000001
x3009 0000010000000101
x300A 0001100100100001
x300B 0001011011111111
x300C 0110010100000000
x300D 0000001111111010
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?

Address	Contents
x3100	9
x3101	7
x3102	32
x3103	0
x3104	-8
x3105	19
x3106	6
x3107	13
x3108	5
x3109	61

Perhaps we didn't

Put a breakpoint at

how many times w

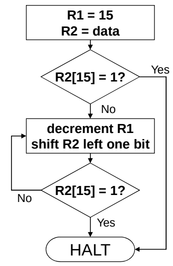
PC	R0	R2
x300D	1	7
x300D	1	32
x300D	1	0
	0	0

```
...
x3007 0110010100000000
x3008 0001010010000001
x3009 0000010000000101
x300A 0001100100100001
x300B 0001011011111111
x300C 0110010100000000
x300D 0000001111111010
x300E 0101000000100000
x300F 1111000000100101
x3010 0011000100000000
```

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.

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.




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


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)

PC	R1
x3007	14
x3007	13
x3007	12
x3007	11
x3007	10
x3007	9
x3007	8
x3007	7
x3007	6
x3007	5

PC	R1
x3007	4
x3007	3
x3007	2
x3007	1
x3007	0
x3007	-1
x3007	-2
x3007	-3
x3007	-4
x3007	-5

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, ...).**