You’ve got to know assembly

Chances are, you’ll never write program in assembly
- Compilers are much better & more patient than you are

Understanding assembly key to machine-level execution model
- Behavior of programs in presence of bugs
  - High-level language model breaks down
- Tuning program performance
  - Understanding sources of program inefficiency
- Implementing system software
  - Compiler has machine code as target
  - Operating systems must manage process state
It’s hard to write code in 1’s & 0’s!

- Assembly language makes it possible to write Machine Language code
  - each line of assembly language is translated into a single ML instruction

- A program called the Assembler does the translation and provides useful tools:
  - use of labels - symbolic names for address locations
  - automatic conversion of binary / hex / decimal
  - pseudo-ops
Assembly Language Instructions

- **Formats**
  - LABEL OPCODE OPERANDS ; COMMENTS
  - LABEL PSEUDO-OPS ; COMMENTS

- **Opcode**
  - Symbolic name for the 4-bit ML opcode

- **Label**
  - Symbolic name for a memory location. It is used to:
    - indicate the target of a branch instruction, e.g. AGAIN in location 0B
    - indicate the location of a stored value or array, e.g. NUMBER and SIX

- **Comments**
  - Intended for humans only: explanation of code, visual display
Pseudo-Ops ...

- ... are directives to the assembler
  - they are not translated into ML instructions

- LC-3 Pseudo-Ops:
  - .ORIG address                     Tells assembler where to locate the program in memory (starting address).
  - .FILL value                      Store value in the next location in memory.
  - .BLKW n                         Set aside a block of n words in memory.
  - .STRINGZ string                 Store the string, one character per word, in memory. Add a word of x0000 after the string.
  - .END                             Marks the end of the source program (not to be confused with the instruction HALT!)
  - .EXTERNAL                       The label so indicated is allocated in another module.
A sample program

01 ;
02 ; Program to multiply an integer by the number 6
03 ;
04 .ORIG x3050
05 LD R1, SIX
06 LD R2, NUMBER
07 AND R3, R3, #0 ; clear R3 to hold the product
08 ;
09 ; The inner loop
0A ;
0B AGAIN ADD R3, R3, R2
0C ADD R1, R1, #-1 ; keep track of iterations
0D BRp AGAIN
0E ;
0F HALT
10 ;
11 NUMBER .BLKW 1
12 SIX .FILL x0006
13 ;
14 .END
From Assembly to bits...

<table>
<thead>
<tr>
<th>x3000:</th>
<th>AND R1, R1, 0 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3001:</td>
<td>ADD R1, R1, 0 1010</td>
</tr>
<tr>
<td>x3002:</td>
<td>LD R2, 0 0000 0010</td>
</tr>
<tr>
<td>x3003:</td>
<td>LD R3, 0 0000 0100</td>
</tr>
<tr>
<td>x3004:</td>
<td>TRAP 0010 0101</td>
</tr>
<tr>
<td>x3005:</td>
<td>0000 0000 0001 0100 ; x0014</td>
</tr>
<tr>
<td>x3006:</td>
<td></td>
</tr>
<tr>
<td>x3007:</td>
<td></td>
</tr>
<tr>
<td>x3008:</td>
<td>0000 0000 0101 0011 ; x0053 = ‘S’</td>
</tr>
<tr>
<td>x3009:</td>
<td>0000 0000 0100 1000 ; x0048 = ‘H’</td>
</tr>
<tr>
<td>x300A:</td>
<td>0000 0000 0110 1001 ; x0069 = ‘i’</td>
</tr>
<tr>
<td>x300B:</td>
<td>0000 0000 0000 0000 ; x0000 = null terminator</td>
</tr>
<tr>
<td>x300C:</td>
<td></td>
</tr>
<tr>
<td>x300D:</td>
<td></td>
</tr>
<tr>
<td>x300E:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORIG x3000</th>
<th>.ORIG x3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND R1, R1, #0</td>
<td>AND R1, R1, #0</td>
</tr>
<tr>
<td>ADD R1, R1, #10</td>
<td>ADD R1, R1, #10</td>
</tr>
<tr>
<td>LD R2, Twenty</td>
<td>LD R2, Twenty</td>
</tr>
<tr>
<td>LD R3, Ess</td>
<td>LD R3, Ess</td>
</tr>
<tr>
<td>HALT</td>
<td>HALT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Twenty</th>
<th>Twenty</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILL x0014</td>
<td>FILL x0014</td>
</tr>
<tr>
<td>.BLKW 2</td>
<td>.BLKW 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ess</th>
<th>.FILL “S”</th>
</tr>
</thead>
<tbody>
<tr>
<td>.STRINGZ “Hi”</td>
<td>.STRINGZ “Hi”</td>
</tr>
<tr>
<td>.BLKW 3</td>
<td>.BLKW 3</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

Wright State University, College of Engineering
Dr. Doom, Computer Science & Engineering

CEG 320/520
Comp. Org. & Assembly
The Assembly Process

- **Objective**
  - Translate the AL (Assembly Language) program into ML (Machine Language).
  - Each AL instruction yields one ML instruction word.
  - Interpret pseudo-ops correctly.

- **Problem**
  - An instruction may reference a label.
  - If the label hasn’t been encountered yet, the assembler can't form the instruction word

- **Solution**
  - Two-pass assembly
Two-Pass Assembly - 1

- **First Pass - generating the symbol table**
  - Scan each line
  - Keep track of current address
    - Increment by 1 for each instruction
    - Adjust as required for any pseudo-ops (e.g. `.FILL` or `.STRINGZ`, etc.)
  - For each label
    - Enter it into the symbol table
    - Allocate to it the current address
  - Stop when `.END` is encountered
### Symbol Table example

- Using the earlier example:

```assembly
; Program to multiply a number by six

.ORIG x3050
x3050    LD    R1, SIX
x3051    LD    R2, NUMBER
x3052    AND   R3, R3, #0

; The inner loop

AGAIN    ADD   R3, R3, R2
ADD       R1, R1, #-1
BRp       AGAIN

HALT

NUMBER   .BLKW  1
SIX      .FILL  x0006

.END
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Again</td>
<td>x3053</td>
</tr>
<tr>
<td>Number</td>
<td>x3057</td>
</tr>
<tr>
<td>Six</td>
<td>x3058</td>
</tr>
</tbody>
</table>
Two-Pass Assembly - 2

- Second Pass - generating the ML program
  - Scan each line again
  - Translate each AL instruction into ML
    - Look up symbols in the symbol table instruction
    - Ensure that labels are no more than +256 / -255 lines from instruction
    - Determine operand field for the instruction
  - Fill memory locations as directed by pseudo-ops
  - Stop when .END is encountered
Assembled code

- Using the earlier example:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Again</td>
<td>x3053</td>
</tr>
<tr>
<td>Number</td>
<td>x3057</td>
</tr>
<tr>
<td>Six</td>
<td>x3058</td>
</tr>
</tbody>
</table>

```assembly
x3050  0010 001 0 0000 0111 ; LD R1, SIX
x3051  0010 010 0 0000 0101 ; LD R2, NUMBER
x3052  0101 011 011 1 00000 ; AND R3, R3, #0
x3053  0001 011 011 0 00 010 ; ADD R3, R3, R2
x3054  0001 001 001 1 11111 ; ADD R1, R1, #-1
x3055  0000 001 1 1111 1101 ; BRp AGAIN
x3056  1111 0000 0010 0101 ; HALT
X3057   ; .BLKW 1
x3058  0000 0000 0000 0110 ; .FILL x0006
```
Another example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>x3002</td>
</tr>
<tr>
<td>Shift</td>
<td>x3005</td>
</tr>
<tr>
<td>Report</td>
<td>x3007</td>
</tr>
<tr>
<td>Input</td>
<td>x300A</td>
</tr>
<tr>
<td>Output</td>
<td>x300B</td>
</tr>
</tbody>
</table>

```
.ORIG x3000
3000 AND R2, R2, x0 ; clear R2
3001 LDI R1, Input ; load word into R1
3002 Count BRz Report ; if 0, done counting
3003 BRp Shift ; if >0, skip ADD
3004 ADD R2, R2, x1 ; increment count
3005 Shift ADD R1, R1, R1 ; shift left 1 bit
3006 BRnzp Count ; go back up
3007 Report AND R3, R2, x1 ; LSB 1 or 0?
3008 STI R3, Output ; store results
3009 TRAP x25 ; halt program
300A Input .FILL x3200 ; address of input
300B Output .FILL x3201 ; address of output
```
From C to executable

- Preprocessing phase: Modifies original program according to preprocessor directives (these start with the # character). The result is another C program text file (typically with the .i suffix)
  - #include <stdio.h>
  - #define FALSE 0
- Compilation phase: text file converted from high-level language to assembly.
  - Regardless of the original high-level language, all programs handled identically from this point onward
  - One assembly corresponds one-to-one with machine instructions
From C to executable

- Assembly phase: assembly text file converted into machine language binary file and packaged into a relocatable object program.
- Linker phase: multiple object programs are merged to result in an executable object file.
  - For example: a standard library function such as printf might reside in a separate precompiled object file (like printf.o) that exists elsewhere in the system.
Object File

- Each source file is translated into an object file
  - a list of ML instructions including the symbol table.

- A complete program may include several source and/or object files:
  - Source files written in Assembly by the programmer
  - Library files provided by the system (OS or other)
  - Compiled HLL libraries

- The object files must be linked
  - One object file will be the “main”
  - All cross-referenced labels in symbol tables will be resolved
The end result ...

- ... is the executable image (.exe file)
  - this is a file ("image") of the finalized list of ML instructions, with all symbolic references resolved
  - it is *loaded* by copying the list into memory, starting at the address specified in the .ORIG directive
  - it is *run* by copying the starting address to the PC
Representing Instructions

- int sum(int x, int y)
- {
  return x+y;
- }
  
  - For this example, Alpha & Sun use two 4-byte instructions
    - Use differing numbers of instructions in other cases
  
  - PC uses 7 instructions with lengths 1, 2, and 3 bytes
    - Same for NT and for Linux
    - NT / Linux not fully binary compatible

Different machines use totally different instructions and encodings
Problems

- 7.1, 7.2 (tricky), 7.4, 7.5, 7.6