Recap…

- Introduced von Neumann model and LC3 instruction set
  - Semantics of each instruction
    - How an instruction is executed
  - Simple instructions in machine language
- Today: Introduce Assembly Language programming and the assembly process
  - Next week and week after will also focus on assembly programming
- Circle back to devices and build a Processor Datapath
  - Project 3

Assembly Language: Human-Readable Machine Language

- Computers like ones and zeros…
  0001110010000110
- Humans like symbols…

    ADD    R6, R2, R6

  - How big of a pain was it to write/read machine instructions (last week’s inclass exercises)?
  - Assembler is a program that turns symbols into machine instructions.

Programming in Assembly

- Assembly language level is one-step up from machine
  - All instructions used in Assembly are actual machine instructions…somewhat!
  - Use mnemonics and address labels to make it easier to understand the program
    - Labels converted to addresses and offsets by assembler
  - “macros” and utilities to make it easier
- Assembler directives
  - Tell assembler what to do without the programmer explicitly writing out the machine code to do the task
    - Allocating storage
    - Initializing data
LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assembler directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with ";") one one line are also ignored.
- An instruction has the following format:

  LABEL OPCODE OPERANDS ; COMMENTS

  optional  mandatory

Opcodes and Operands

- **Opcodes**
  - reserved symbols that correspond to actual (LC-3) instructions
  - listed in Appendix A
    - e.g. ADD, AND, LD, LDR, ...
- **Operands**
  - registers -- specified by Rn, where n is the register number
  - numbers -- indicated by # (decimal) or x (hex)
  - label -- symbolic name of memory location
    - separated by comma
  - number, order, and type correspond to instruction format
    - e.g. ADD R1,R1,R3
    - ADD R1,R1,#3
    - LD R6,NUMBER
    - BRz LOOP

Labels and Comments

- **Label**
  - placed at the beginning of the line
  - assigns a symbolic name to the address corresponding to line
    - e.g. LOOP corresponds to some specific memory address
    - LOOP ADD R1,R1,#-1
- **Comment**
  - anything after a semicolon is a comment
  - ignored by assembler
  - used by humans to document/understand programs
  - tips for useful comments:
    - avoid restating the obvious, as "dec increment R1"
    - provide additional insight, as in "accumulate product in R6"
    - use comments to separate pieces of program

Assembler Directives

- **Pseudo-operations.** To make programmer's life easier
  - do not refer to operations executed by program
  - used by assembler
  - look like instruction, but "opcode" starts with dot

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>address</td>
<td>starting address of program</td>
</tr>
<tr>
<td>.END</td>
<td></td>
<td>end of program</td>
</tr>
<tr>
<td>.BLK N</td>
<td>n</td>
<td>allocate n words of storage</td>
</tr>
<tr>
<td>.FILL</td>
<td>A</td>
<td>allocate one word, initialize with value A</td>
</tr>
<tr>
<td>.STRING</td>
<td>n-character string</td>
<td>allocate n+1 locations, initialize w/characters and null terminator</td>
</tr>
</tbody>
</table>
**Trap Codes**

LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them... more on TRAP instructions later...

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>

---

**Example Assembly Program – Add 2 to non-negative number and store into another memory location**

```assembly
.ORIG x3000  ; program starts at address x3000
LD R1, PLACE1 ; PLACE is location in memory
    ; note: offset not specified by programmer
    ; assembler calculates offset needed
BRn Done ; if number is Negative goto end
ADD R3, R1, #2 ; Add 2 store into R3
ST R3, PLACE2 ; store result into PLACE2
Done HALT ; halt program
PLACE2 .BLKW 1 ; reserve/set aside one word in memory
PLACE1 .FILL x0005 ; initialize number to 5
.END ; end of program
```

This code would generate identical machine code as previous with label PLACE1.
Example Assembly Program – Add 2 to non-negative number and store into another memory location
; load number from locations PLACE1.
.ORIG x3000 ;program starts at address x3000
LD R1, PLACE1 ; PLACE is location in memory
; note: offset not specified by programmer
BRn Done ;if number is Negative goto end
ADD R3, R1, #2 ; add 2 store into R3
ST R3, PLACE2 ; store result into PLACE2
Done HALT ;halt program

PLACE2 .BLKW 1
PLACE1 .FILL x0005

Assembler Directive (reserve one location with label 'PLACE2')

Assembly Process

• Assembler: Converts assembly language file (.asm) into an executable file (.obj) ...for the LC-3 simulator in our case.

1st Pass
2nd Pass

Symbol Table

• First Pass:
  a) scan program file
  b) find all labels and calculate the corresponding addresses;
     this is called the symbol table

• Second Pass:
  a) convert instructions to machine language,
     using information from symbol table

First Pass: Constructing the Symbol Table

1. Find the .ORIG statement,
   which tells us the address of the first instruction.
   • Initialize location counter (LC), which keeps track of the current instruction.

2. For each non-empty line in the program:
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
     • NOTE: If statement is .BLKW or .STRINGZ, increment LC by the number of words allocated.

3. Stop when .END statement is reached.
   • NOTE: A line that contains only a comment is considered an empty line.

Pass 1

• Construct the symbol table for the program

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Done</td>
<td></td>
</tr>
<tr>
<td>PLACE2</td>
<td></td>
</tr>
<tr>
<td>PLACE1</td>
<td></td>
</tr>
</tbody>
</table>
Second Pass: Generating Machine Language

- For each executable assembly language statement, generate the corresponding machine language instruction.
  - If operand is a label, look up the address from the symbol table.

- Potential problems:
  - Improper number or type of arguments
    - ex: ADD R1, R2
  - Immediate argument too large
    - ex: ADD R1, #1023
  - Address (associated with label) more than 256 from instruction
    - can’t use PC-relative addressing mode

Pass 2

- Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, SIX</td>
<td></td>
</tr>
<tr>
<td>BRp AGAIN</td>
<td></td>
</tr>
<tr>
<td>LD R2, NUMBER</td>
<td></td>
</tr>
</tbody>
</table>

LC-3 Assembler

- Using “assemble” (Unix) or LC3Edit (Windows), generates several different output files.

Multiple Object Files

- An object file is not necessarily a complete program.
  - system-provided library routines
  - code blocks written by multiple developers

- For LC-3 simulator, can manually load multiple object files into memory, then start executing at a desired address.
  - system routines, such as keyboard input, are loaded automatically
    - loaded into “system memory,” below x3000
    - user code should be loaded between x3000 and xDFF
  - each object file includes a starting address
  - be careful not to load overlapping object files
Linking and Loading

- **Loading** is the process of copying an executable image into memory.
  - more sophisticated loaders are able to relocate images to fit into available memory
  - must readjust branch targets, load/store addresses

- **Linking** is the process of resolving symbols between independent object files.
  - suppose we define a symbol in one module, and want to use it in another
  - some notation, such as `EXTERNAL`, is used to tell assembler that a symbol is defined in another module
  - linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

Example 1

- Download assembly1.asm
- Open LC3 edit
- Assemble
- Open LC3 simulator
- Load program: assembly1.obj
- Set breakpoint at last instruction of your program– this will stop the simulation after the instruction
  - Set it by choosing it from menu or…
  - Simpler: click on small dot next to where you want to set breakpoint…it goes red to indicate breakpoint set.

Programming in assembly..

- Style guidelines
- Problem decomposition and mapping to assembly

Style Guidelines

1. Provide a program header…standard stuff
2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   1. Mixed upper and lower case for readability.
   2. ASCIlToBinary, InputRoutine, SaveR1
6. Provide comments between program sections.
Recap: Problem Solving and Problem Decomposition

- With an eye towards writing assembly programming/low-level software
- Flowcharts anyone?
- Decomposition:
  - Break problem/solution into sub-problems/modules
  - Structured programming
  - Connect the modules…
    - With conditionals, iterations, sequence,….

Example

- Array of N numbers
- Read length N of the array
- Replace negative numbers by 0
- Add all the (new) numbers
- Print the sum

Three Basic Constructs

- There are three basic ways to decompose a task:

Sequential

- do Subtask 1, then subtask 2, etc.
  - Read value of N from keyboard
  - Process Array of Nums
  - Change –ve to 0 and Compute Sum of nums
  - Print Sum
  - Go through Array
  - Compute Sum
  - Print Sum
### Conditional

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

  - Check if number \( \geq 0 \)
  - Change –ve to 0

  - \( x < 0 \)
  - \( x = 0 \)
  - \( x = x \)

### Iterative

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

  - Check each element in array and compute sum

### 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**

- Test Condition
- Generate Condition
- Instruction
- Unconditional branch to reset condition
- Exact bits depend on condition being tested
- PC offset to address C
- PC offset to address A

Assuming all addresses are on the same page.

**Converting Code to Assembly**

- Can use a standard template approach
- Typical Constructs
  - if/else
  - while
  - do/while
  - for

```c
if(x > 0)
{
    r2 = r3 + r4;
}
else
{
    r5 = r6 + r7;
}
```

```assembly
if(x > 0)       LD R1, X       BRP THEN
{               ADD R5,R6,R7       BRNZP DONE
    r2 = r3 + r4;
}              THEN ADD R2,R3,R4       DONE ...
else
{               DONE ...
    r5 = r6 + r7;
}
```

```assembly
if(x > 0)       LD R1, X       BRNZ ELSE
{               ADD R2,R3,R4
    r2 = r3 + r4;
}               BRNZP DONE
else
{               ELSE ADD R5,R6,R7
    r5 = r6 + r7;
}               DONE ...
```
while

```csharp
x = 0;
i = 10;
while (i > 0)
{
x = x + i;
i--;
}
```

AND R1,R1,#0
AND R2,R2,#0
ADD R1,R1,#10
WHL BRNZ DONE
ADD R2,R2,R1
ADD R1,R1,#-1
BRNZP WHL

Next: Connecting the dots – Datapath Design

- Starting using assembler this week
  - Lab sessions
- How is the instruction set implemented in hardware….
  - Processor datapath
- Assemble a processor datapath using the components we have…