Chapter 10/11/12/13

High Level Programming Languages

Variables and Operands

The runtime stack

Emphasis on how C-like languages are converted to LC-3 assembly

A High-Level Languages

- Gives symbolic names to values
- Don't need to know which register or memory location
- Provides abstraction of underlying hardware
- Operations do not depend on instruction set
- Example: can write "a = b * c", even though LC-3 doesn't have a multiply instruction
- Provides expressiveness
- Use meaningful symbols that convey meaning
- Simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
- Can enforce rules or conditions at compile-time or run-time
- If it can be specified in a high-level language then it MUST be doable in assembly!

Compilation vs. Interpretation

- Different ways of translating high-level language
- Interpretation
  - Interpreter = program that executes program statements
  - Generally one line/command at a time
  - Limited processing
  - Easy to debug, make changes, view intermediate results
  - Languages: BASIC, LISP, Perl, Java, Matlab, C-shell
- Compilation
  - Translates statements into machine language
  - Does not execute, but creates executable program
  - Performs optimization over multiple statements
  - Change requires recompilation
  - Can be harder to debug, since executed code may be different
  - Languages: C, C++, Fortran, Pascal

Compiling a C Program

- Entire mechanism is usually called the "compiler"
- Preprocessor
  - Macro substitution
  - Conditional compilation
  - "Source-level" transformations
  - Output is ELF
- Compiler
  - Generates object file
  - Machine instructions
- Linker
  - Combines object files (including libraries)
  - Into executable image

A Simple C Program

```c
#include <stdio.h>
#define STOP 0

/* Function: main                                   */
/* Description: counts down from user input to STOP */
main()
{
  /* variable declarations */
  int counter;
  /* an integer to hold count values */
  int startPoint;
  /* starting point for countdown */

  printf("Enter a positive number: ");
  scanf("%d", &startPoint);
  /* output count down */
  for (counter=startPoint; counter >= STOP; counter--)
  {
    printf("%d
", counter);
  }
}```
Preprocessor Directives

- #include <stdio.h>
  - Before compiling, copy contents of header file (stdio.h) into source code.
  - Header files typically contain descriptions of functions and variables used by the program.
- #define STOP 0
  - Before compiling, replace all instances of the string "STOP" with the string "0".
  - Called a macro.
- Every C program must have exactly one function called main().
  - main() determines the initial PC.

Examples of Output

- This code:
  - printf("%d is a prime number.\n", 43);
  - printf("%d plus 59 in decimal is %d.d", 43, 59);
  - printf("%d plus 59 in hex is %x.d", 43, 59);
- produces this output:
  - 43 is a prime number.
  - 43 + 59 in decimal is 102.
  - 43 + 59 in hex is 66.
  - 43 + 59 as a character is f.

Output with printf

- Variety of I/O functions in C Standard Library.
- Must include <stdio.h> to use them.
- printf: Can print arbitrary expressions, including formatted variables
  - printf("%d %d\n", startPoint - counter);
  - Print multiple expressions with a single statement
  - printf("%d %d %d\n", counter, startPoint - counter);
- Different formatting options:
  - %d decimal integer
  - %x hexadecimal integer
  - %c ASCII character
  - %f floating-point number

Input with scanf

- Many of the same formatting characters are available for user input.
- scanf("%c", &nextChar);
  - reads a single character and stores it in nextChar
- scanf("%f", &radius);
  - reads a floating point number and stores it in radius
- scanf("%d %d %d\n", &length, &width, &counter);
  - reads two decimal integers (separated by whitespace), stores the first one
    in length and the second in width
- Must use address of operator (&) for variables being modified.
  - We'll revisit pass by reference/value in a future lecture

Data Types

- Variables are used as names for data items.
- Each variable has a type, type qualifiers, and a storage class which tells
  the compiler how the data is to be interpreted (and how much space it
  needs, etc.).
- int counter;
- Basic data types:
  - Integral: int (at least 16 bits) Qualifiers: signed, unsigned, long
  - Floating-point: float (at least 32 bits), double
  - Character: char (at least 8 bits)
  - Enumerated: enum hobbits {bilbo, frodo, samwise, pippen, merry}
  - Storage class: automatic, static, register
  - Derived data types: pointers, arrays, structures
- Exact size can vary, depending on processor
  - int is supposed to be "natural" integer size;
  - for 32-bit, that’s 4 bytes – 32 bits for most modern processors

High level languages have rules that specify the data type of result

- Addition/Subtraction: If mixed types, smaller type is "promoted" to larger
  - int x = 5
  - int y = 4.3
  - answer will be float
- Division: If mixed type, the default result is a truncated signed integer
  - For int x = 5
  - (x / 3 == 1) is true! Not 1.6! Not 2! 1!
  - For float y = 5
  - (y / 3 == 1) is false!
- The rules can be overridden by typecasting the operands or result!
  - the compiler does this for you automatically to match the destination
    type!
  - int i = 2.5 / 3
  - i is 0
  - float f = 2.5 / 3
  - f is 0.833333
  - Note automatic typecasting of int
- Without typecasting you are stuck with the limitations of the data type
  the compiler assigned for the storage calculation of your intermediate
  value!
Variables and Scope

- Where are variable stored? Where can they be accessed? Why?
  - All C variables are defined as being in one of two storage classes:
    - Automatic storage class (on the stack, uninitialized)
    - Static storage class (in the global memory area, initialized to 0)
  - Compiler infers scope from where variable is declared unless specified
  - programmer doesn’t have to explicitly state (but can!)
  - automatic int x;
  - static int y;
- Global: accessed anywhere in program (default static)
  - Global variable is declared outside all blocks
  - Variable is local to the block in which it is declared
  - Variable is only accessible in a particular region (default automatic)
  - Variable is local to the block in which it is declared
  - Block defined by open and closed braces { }
  - Global: accessed anywhere in program (default static)
  - Global variable is declared outside all blocks
- Offset = distance from beginning

Variable is local to the block in which it is declared

Global: accessed anywhere in program (default static)

- Global variable is declared outside all blocks
- Variable is local to the block in which it is declared
- Variable is only accessible in a particular region (default automatic)
- Variable is local to the block in which it is declared
- Block defined by open and closed braces { }
- Global: accessed anywhere in program (default static)
- Global variable is declared outside all blocks

Stack Data Structure

- Abstract Data Structures
  - are defined simply by the rules for inserting and extracting data
- The rule for a Stack is LIFO (Last In First Out)
  - Operations:
    - Push (enter item at top of stack)
    - Pop (remove item from top of stack)
  - Error conditions:
    - Underflow (trying to pop from empty stack)
    - Overflow (trying to push onto full stack)
- We just have to keep track of the address of top of stack (TOS)

Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
    - R4 points to beginning (global pointer)
    - R5 points to top of stack (stack pointer)
    - New frame for each block
  - Offset = distance from beginning of storage area
    - Global: LDR R1, #4
      - Local: LDR R2, #4

- Implemented in memory
  - The Top Of Stack moves as new data is entered
    - Here R6 is the TOS register, a pointer to the Top Of Stack

- In the assembler
  - = 4; /* change global variable

- In the compiler
  - = 1; /* local to main */
  - = 2; /* local to this block */

Example

```c
#include <stdio.h>
int itsGlobal = 0;
main()
{
   int itsLocal = 1; /* local to main */
   printf("Global %d Local %d", itsGlobal, itsLocal);
   int itsLocal = 2; /* local to this block */
   itsGlobal = 4; /* change global variable */
   printf("Global %d Local %d", itsGlobal, itsLocal);
   printf("Global %d Local %d", itsGlobal, itsLocal);
}
```

Variables and Memory Locations

- In our examples, a variable is always stored in memory.
  - For each assignment, one must get the operands (possibly requiring memory loads), perform the operation, and then store the result to memory.
- Optimizing compilers try to keep variables allocated in registers.
  - Compiler infers scope from where variable is declared unless specified
  - All global variables stored here (actually all static variables)
    - R4 points to beginning (global pointer)
    - R5 points to top of stack (stack pointer)
    - New frame for each block
  - Offset = distance from beginning of storage area
    - Global: LDR R1, #4
      - Local: LDR R2, #4
```

Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
    - R4 points to beginning (global pointer)
    - R5 points to top of stack (stack pointer)
    - New frame for each block
  - Offset = distance from beginning of storage area
    - Global: LDR R1, #4
      - Local: LDR R2, #4

- Implemented in memory
  - The Top Of Stack moves as new data is entered
    - Here R6 is the TOS register, a pointer to the Top Of Stack

- In the assembler
  - = 4; /* change global variable

- In the compiler
  - = 1; /* local to main */
  - = 2; /* local to this block */
```

Example

```c
#include <stdio.h>
int itsGlobal = 0;
main()
{
   int itsLocal = 1; /* local to main */
   printf("Global %d Local %d", itsGlobal, itsLocal);
   int itsLocal = 2; /* local to this block */
   itsGlobal = 4; /* change global variable */
   printf("Global %d Local %d", itsGlobal, itsLocal);
   printf("Global %d Local %d", itsGlobal, itsLocal);
}
```

Variables and Memory Locations

- In our examples, a variable is always stored in memory.
  - For each assignment, one must get the operands (possibly requiring memory loads), perform the operation, and then store the result to memory.
- Optimizing compilers try to keep variables allocated in registers.
  -Compiler infers scope from where variable is declared unless specified
  - All global variables stored here (actually all static variables)
    - R4 points to beginning (global pointer)
    - R5 points to top of stack (stack pointer)
    - New frame for each block
  - Offset = distance from beginning of storage area
    - Global: LDR R1, #4
Example: Compiling to LC-3

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>inLocal</td>
<td>int</td>
<td>0</td>
<td>local</td>
</tr>
<tr>
<td>inGlobal</td>
<td>int</td>
<td>1</td>
<td>global</td>
</tr>
<tr>
<td>outLocalA</td>
<td>int</td>
<td>2</td>
<td>main</td>
</tr>
<tr>
<td>outLocalB</td>
<td>int</td>
<td>3</td>
<td>main</td>
</tr>
</tbody>
</table>

```c
main() {
  int inLocal;
  int inGlobal;

  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal)

  /* print results */
  printf("The results are: outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}
```

The stack frame

- Local variables are stored in a stack frame associated with the current scope
- As we change scope, we effectively change both the top and the bottom of the stack
- R6 is the stack pointer - holds the address of the top of the stack
- R5 is the frame pointer - holds address of the base of the current frame.

Symbol table "offset" gives the distance from the base of the frame.
- A new frame is pushed on the run-time stack each time a block is entered.
- Because stack grows downward (towards memory address 0x000) the base is the highest address of the frame, and variable offsets are negative.

Operators

- Programmers manipulate variables using the operators provided by the high-level language.
- You need to know what these operators assume
  - Function
  - Precedence & Associativity
  - Data type of result
- You are assumed to know all standard C/C++ operators, including bitwise ops:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>bitwise NOT</td>
<td>x</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>x &lt;&lt; y</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>x &gt;&gt; y</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>x &amp; y</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>x ^ y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
</tr>
</tbody>
</table>

Control Structures

- If it can be done in "C" in must be able to be done in assembly
- Conditionals
  - making a decision about which code to execute, based on evaluated expression
  - if
  - if-else
  - switch
- Iteration
  - executing code multiple times, ending based on evaluated expression
  - while
  - for
  - do-while

Implementing IF-ELSE

```
branch_offset:
  LDR R0, R5, #0
  BRA ELSE

[x is not zero]
  LDR R1, R5, #1 ; incr y
  ADD R1, R1, #1
  STR R1, R5, #1
  LDR R1, R5, #02 ; decr z
  ADD R1, R1, #1
  STR R1, R5, #02
  JMP DONE ; skip else code

[x is zero]
  LDR R1, R5, #1 ; decr y
  ADD R1, R1, #1
  STR R1, R5, #1
  LDR R1, R5, #02 ; incr z
  ADD R1, R1, #1
  STR R1, R5, #02

DONE
```

Switch

```
switch (expression) {
  case const1:
    action1; break;
  case const2:
    action2; break;
  default:
    action3;
  }
```

Alternative to long if-else chain. Case expressions must be constant. If break is not used, then case "falls through" to the next.
Implementing WHILE

```c
x = 0;
while (x < 10) {
    printf("%d ", x);
    x = x + 1;
}
```

Example: Compiling to LC-3

```c
#include <stdio.h>

main() {
    int inLocal, inGlobal;
    inLocal = 5;
    inGlobal = 3;
    // perform calculations
    outLocalA = (inLocal + inGlobal) - (inLocal - inGlobal);
    // print results
    printf("The results are: outLocalA = %d, outLocalB = %d
", outLocalA, outLocalB);
}
```

Example: Code Generation

```c
; main
; initialize variables
; inLocal = 5; inGlobal = 3;
AND R0, R0, #0
ADD R0, R0, #5
STR R0, R5, #0
; inLocal = 5
ADD R1, R0, #1
INCR R1
STR R1, R5, #1
; inLocal++, incr R1
ADD R0, R0, R1
STR R0, R5, #0
; inLocal = inLocal + inGlobal
ADD R1, R0, R1
; increment
MOVE R2, R1
LDR R0, R5, #0
; load R0
```

Example (continued)

```c
; first statement:
LDR R0, R5, #0 ; get inLocal
ADD R1, R0, #1 ; increment
STR R1, R5, #0 ; store
```

Example (continued)

```c
; next statement:
LDR R0, R5, #0 ; inLocal
LDR R1, R4, #0 ; inGlobal
ADD R0, R1, #0 ; R0 is sum
LDR R2, R5, #0 ; inLocal
```

```c
; re-init
ADD R0, R0, #1
STR R0, R5, #0
JMP LOOP
```

```c
; test body
AND R0, R0, #0
STR R0, R5, #0
```

```c
; test
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; loop body
LDR R0, R5, #0
ADD R0, R0, #1
STR R0, R5, #0
```

```c
; test again
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; loop body
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; test again
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; loop body
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; test again
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; loop body
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; test again
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; loop body
LDR R0, R5, #0
ADD R0, R0, #1
```

```c
; test again
LDR R0, R5, #0
ADD R0, R0, #1
```
Practice problems

- 10.3, 10.8, 12.1, 12.5