Chapter 10/11/12/13

High Level Programming Languages

Variables and Operators

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 do-able in assembly!

Compilation vs. Interpretation

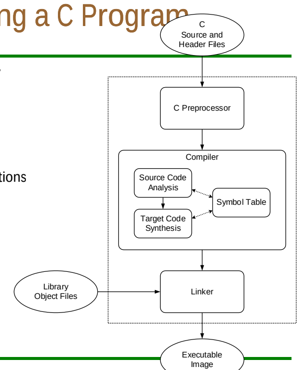
- Different ways of translating high-level language
- *Interpretation*
  - interpreter = program that executes program statements
  - generally one line/command at a time
  - limited processing
  - easy to debug, make changes, view intermediate results
  - languages: BASIC, LISP, Perl, Java, Matlab, 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

Get W from the keyboard.
X = W + W
Y = X + X
Z = Y + Y
Print Z to screen.

How many arithmetic operations when interpreted? When compiled with optimization?

Compiling a C Program

- Entire mechanism is usually called the "compiler"
- Preprocessor
  - macro substitution
  - conditional compilation
  - "source-level" transformations
    - output is still C
- Compiler
  - generates object file
    - machine instructions
- Linker
  - combine object files (including libraries) into executable image



Compiler

- Source Code Analysis
  - "front end"
  - parses programs to identify its pieces
    - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)
- Code Generation
  - "back end"
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
    - Consider automated HTML generation...
  - very dependent on target machine
- Symbol Table
  - map between symbolic names and items
  - like assembler, but more kinds of information

A Simple C Program

```
#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\n", 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 needed by the program.
    - no restrictions -- could be any C source code
- `#define STOP 0`
  - Before compiling, replace all instances of the string "STOP" with the string "0"
  - Called a *macro*
  - Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)
- Every C program must have exactly one function called `main()`.
  - Be careful with what you #include!
  - main() determines the initial PC.

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\n", startPoint - counter);`
- Print multiple expressions with a single statement
- `printf("%d %d\n", counter, startPoint - counter);`
- Different formatting options:
- `%d` decimal integer
- `%x` hexadecimal integer
- `%c` ASCII character
- `%f` floating-point number

Examples of Output

- This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 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.
```


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", &length, &width);`
  - 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 LC-3, that's 16 bits -- 32 bits for most modern processors

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

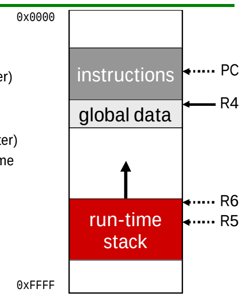
- Addition/Subtraction: If mixed types, smaller type is "promoted" to larger.
  - `x + 4.3` answer will be float
- Division: If mixed type, the default result is a truncated signed integer
  - For int x = 5: (x / 3 = = 1) is true! Not 1.6! Not 2! 1!
  - For float f = 5: (f / 3 = = 1) is false!
  - For int x = 5: ((float) x / 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 si = 2.5 / 3 is 0
  - float f = 2.5 / 3 is 0.833333 [Note automatic typecasting of 3]
- 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
- Local: 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 { }
  - can access variable declared in any "containing" block

Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
  - R4 points to beginning (global pointer)
- Run-time stack
  - Used for local variables
  - R6 points to top of stack (stack pointer)
  - R5 points to top frame on stack (frame pointer)
  - New frame for each block (goes away when block exited)
- Offset = distance from beginning of storage area
  - Global: LDR R1, R4, #4
  - Local: LDR R2, R5, #-3

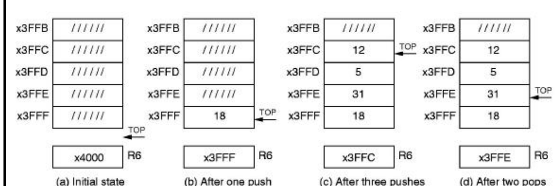


Stack Data Structure

- 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)

A software stack

- 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



Example

```
#include <stdio.h>
int itsGlobal = 0;

main()
{
  int itsLocal = 1;   /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  {
    int itsLocal = 2;   /* local to this block */
    itsGlobal = 4;      /* change global variable */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
  }
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```


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.
  - C allows the user to provide hints to the compiler
    - register int x;
- Like the assembler, the compiler needs a symbol table
- In the assembler
  - Identifiers (names) are labels associated with memory addresses
- In the compiler
  - Name, Type, Location, Scope

Compiler Symbol Table

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

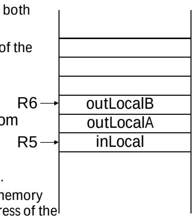
Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main() {
   int inLocal;
   int outLocalA;
   int outLocalB;

   inLocal = 5;
   inGlobal = 3;

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

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

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

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 x0000) 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:

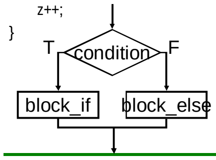
Symbol	Operation	Usage
~	bitwise NOT	~x
<<	left shift	x << y
>>	right shift	x >> y
&	bitwise AND	x & y
^	bitwise XOR	x ^ y
|	bitwise OR	x | y

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

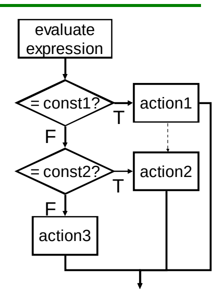
```
int x,y,z;
....
if (x){
     y++;
     z--;
} else {
     y--;
     z++;
}
```




```
        LDR  R0, R5, #0
        BRz  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, #-2
        JMP  DONE  ; skip else code
        ; x is zero
ELSE    LDR  R1, R5, #-1   ; decr y
        ADD  R1, R1, #-1
        STR  R1, R5, #-1
        LDR  R1, R5, #-2   ; incr z
        ADD  R1, R1, #1
        STR  R1, R5, #-2
DONE    ...  ; next statement
```


Switch

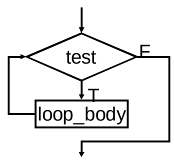
```
switch (expression) {
case const1:
   action1; break;
case const2:
   action2; break;
default:
   action3;
}
```



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

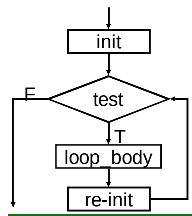
```
x = 0;
while (x < 10) {
   printf("%d ", x);
   x = x + 1;
}
```




```
        AND  R0, R0, #0
        STR  R0, R5, #0  ; x = 0
        ; test
LOOP    LDR  R0, R5, #0  ; load x
        ADD  R0, R0, #-10
        BRzp DONE
        ; loop body
        LDR  R0, R5, #0  ; load x
        ...
        <printf>
        ...
        ADD  R0, R0, #1  ; incr x
        STR  R0, R5, #0
        JMP  LOOP        ; test again
DONE    ; next statement
```


Implementing FOR

```
for (i = 0; i < 10; i++)
  printf("%d ", i);
```




```
        ; init
        AND  R0, R0, #0
        STR  R0, R5, #0  ; i = 0
        ; test
LOOP    LDR  R0, R5, #0  ; load i
        ADD  R0, R0, #-10
        BRzp DONE
        ; loop body
        LDR  R0, R5, #0  ; load i
        ...
        <printf>
        ...
        ; re-init
        ADD  R0, R0, #1  ; incr i
        STR  R0, R5, #0
        JMP  LOOP        ; test again
DONE    ; next statement
```


Example: Compiling to LC-3

```
#include <stdio.h>
int inGlobal;

main() {
  int inLocal;
  int outLocalA;
  int outLocalB;

  inLocal = 5;
  inGlobal = 3;

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

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

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

Example: Code Generation

```
; main
; initialize variables
; inLocal = 5;  inGlobal = 3;
      AND R0, R0, #0
      ADD R0, R0, #5  ; inLocal = 5
      STR R0, R5, #0  ; (offset = 0)

      AND R0, R0, #0
      ADD R0, R0, #3  ; inGlobal = 3
      STR R0, R4, #0  ; (offset = 0)
```


Example (continued)

```
; first statement:
; outLocalA = inLocal++ & ~inGlobal;

      LDR R0, R5, #0  ; get inLocal
      ADD R1, R0, #1  ; increment
      STR R1, R5, #0  ; store

      LDR R1, R4, #0  ; get inGlobal
      NOT R1, R1      ; ~inGlobal
      AND R2, R0, R1  ; inLocal & ~inGlobal
      STR R2, R5, #-1 ; store in outLocalA
                      ; (offset = -1)
```


Example (continued)

```
; next statement:
; outLocalB = (inLocal + inGlobal)
;             - (inLocal - inGlobal);

      LDR R0, R5, #0  ; inLocal
      LDR R1, R4, #0  ; inGlobal
      ADD R0, R0, R1  ; R0 is sum
      LDR R2, R5, #0  ; inLocal
      LDR R3, R5, #0  ; inGlobal
      NOT R3, R3
      ADD R3, R3, #1
      ADD R2, R2, R3  ; R2 is difference
      NOT R2, R2      ; negate
      ADD R2, R2, #1
      ADD R0, R0, R2  ; R0 = R0 - R2
      STR R0, R5, #-2 ; outLocalB (offset = -2)
```

Practice problems

- 10.3, 10.8, 12.1, 12.5