19-hll-p1-notes

High-Level Languages

Agenda

- 0. Re-Orienting
- 1. High-Level Languages
- 2. C Program Structure
- 3. The Architectural Perspective
- 4. Basic Data Types
- 5. C to ISA Demo
- 6. Variables

Reading:

- 2.0, 2.1.3 through 2.3.8
- 3.2 through 3.5

---

slides.pdf (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/slides.pdf)

---

0. Re-Orienting

From thinking about software down to assembly/machine language for an ISA.

From electricity up to a microarchitecture implementing that ISA.

The field has progressed up a messiness/speed curve, trading simplicity/elegance of design for increased performance.

On Friday, we started exploring one component of this trade: pipelining.

In your project this week, you're working through the details of a basic microarchitecture.

Going deeper into pipelining (and other "dirty tricks") now---adding more complexity to the uarch---might be confusing as you're just digesting the basic uarch. So we'll put that off a bit and instead focus on....

---

1. High-Level Languages

What's the point of using one?

How does it relate to this assembler and opcodes and hardware stuff we've been talking about?

- compiled vs interpreted vs...
- **understanding the underlying hardware lets you make better use of the higher-level software**
- and vice-versa

Intro to C

The successor to "B"

Invented by Dennis Ritchie in the early 1970s, because writing his OS (Unix) in assembler got to be too annoying.

Why and where it's still used...

Why it's frowned upon

But why we're using a (simple subset) of it here...

- because it's a high-level language that's fairly close to the metal

- so it's good at at illustrating the hardware/software interface

---

In Practice

The tool chain, in theory

- Pre-processor

- Compilation. (The compiler transforms high-level source to "object code"---a rough draft of the executable)

- Linking (including to library code that provides useful services---and may even trap down to the OS). The linker patches up loose ends (such as unresolved symbols) across multiple object files, and builds the final executable.

e.e....Fig 11.2 from Patt and Patel

The tool chain, in practice (in general)

- typically, a syntax-aware editor, like emacs
- gcc (preproc/compiler/assembler/linker all rolled into one)
- some type of debugger, such as gdb
- or maybe a nifty IDE that packages all this functionality together

---

Ways to look at C-to-assembler mappings

```
gcc -m32 -O0 -S code.s
```

Some differences from book:

- The -m32 says "use the 32-bit address space model"...simpler code
- -O0 says: don't optimize at all! (So more basic structure is visible)

```


gcc -m32 -O0 -g -o code code.c
gdb code
disas main
```

- -g: include symbol data in executable
- -o code: call it "code"
- gdb: run it in the debuggerr
- disassemble!

On Linux, as per textbook:

```
objdump -d add
```

On OSX:

```
otool -tv add
```

The [class resources page (https://canvas.dartmouth.edu/courses/8347/pages/resources)](https://canvas.dartmouth.edu/courses/8347/pages/resources) has links to x86-family instruction set references.

Caveat: "The ever-changing forms of generated code"

2. C Program Structure

Fig 11-3 from Patt & Patel:

```
/*
 *
 *  Program Name : countdown, our first C program
 *
 *  Description  : This program prompts the user to type in
 *  a positive number and counts down from that number to 0,
 *  displaying each number along the way.
 *
 */

/* The next two lines are preprocessor directives */
#include <stdio.h>
#define STOP 0

/* Function    : main                                    */
/* Description : prompt for input, then display countdown */
int main()
{
  /* Variable declarations */
  int counter;         /* Holds intermediate count values */
  int startPoint;      /* Starting point for count down   */

  /* Prompt the user for input */
  printf("===== Countdown Program =====\n");
  printf("Enter a positive integer: ");
  scanf("%d", &startPoint);

  /* Count down from the input number to 0 */
  for (counter = startPoint; counter >= STOP; counter--)
    printf("%d\n", counter);
}
```

talk through basic pieces....

- comments
- preprocessor directives
- main()
- curly braces
- indentation, for readability
- keywords, and function calls

Where We're Going: just enough of the following to see how the hardware does it!

- basic data types
- local and global variables
- arithmetic operators
- if, else, while, for, ? :
- subroutines
- pointers
- recursion
- basic data structures

The I/O we will use (from the surface)

putchar()

getchar()

printf()

scanf() (a bit "mysterious")

---

3. The Architecture Perspective

We're looking at C (as an example high-level language) in order to see how higher-level programming maps on to lower-level hardware (and, indirectly, the influences that flow both ways).

(If you don't know C already, don't worry!)

And keep in mind:

- We're looking at the code generated for the real x86 family, not a toy processor
- and generated by an industrial-strength toolchain, not a toy compiler
- and it's going to make sense!

And this is what you will see in your real life!

A straightforward way to see language/arch interaction: data types.

---

4. Basic Data Types

Typically......

- a char is a byte
  - as a constant, use single quotes. 'x'
- a char string is a consecutive sequence of bytes in memory, terminated by a null byte
  - as a constant, use double quotes. "there is no dark side of the moon, really"
  - the "address" of the string is the address of the zeroth byte... which comes lowest
- an int is a a 4-byte item
- signed things are two's complement

A pointer is just an address. (See pages 39, 43--44, 175 in the textbook)

- foo *fp means that fp contains an address, and what's at that address should be interpreted as a foo
- &x is the address of x
- x = *fp: get the thing at address fp and save it at x
- *fp = x: store x at the address contained in fp

See Figures 2.3 and 2.8.

(Where did the numbers come from in Fig 2.8?)

---

types.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/types.c)

(Explore variables and such with gdb. "print/x", "x/20b")

If time, play with:

- **myst0.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/myst0.c)**
- **myst1.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/myst1.c)**
- **myst1.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/myst1.c)**

---

5. C to ISA Demo

Fig 3.6 in the textbook, but turned to globals

[exchange0.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/exchange0.c)](https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/exchange0.c)

(-O1 is clearer)

Now Fig 3.6 straight on, with

- local variables (x)
- arguments (xp, y)
- return value (x)

[exchange1.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/exchange1.c)](https://ssl.cs.dartmouth.edu/~sws/cs51-s15/19-hll-p1/demo/exchange1.c)

With -O1... you see

- copying the stack pointer to a a "base pointer"
- references to the arguments, as positive offsets from the base pointer

With -O0....

- the stack pointer goes down by 12
- references to the arguments, as positive offsets from the base pointer
- references to "locals" (x, and mysteries) as negative offsets from the base pointer

---

6. Variables

Two main flavors

Global variables

- As we've seen so far...
- Declared at the beginning
- Pre-initialized to 0, unless we specify otherwise
- Visible throughout the program file

- typically, these live within a "global data" region in memory
- In gcc today...seem to be accessed via a global pointer (possibly offset from a register). In other compilers, one sometimes more principled use of offsets from a "global data pointer" register

---

Local variables

- Declared within a {...} code block
- Visible only within that block, after declaration
- Typically, declared at the start of a function, and visible only within that.

If we don't explicitly initialize a local variable...

- it will have an initial value
- but we don't necessarily know what it is
- and it can change each time we run that function
- accessed via a negative offset from the base pointer (or... sometimes kept in a register entirely)

What about ARGUMENTS?

It looks like these are also in one region, but on the other side of the base pointer.
