20-hll-p2-notes

High-Level Languages: Operations and Control Flow

Agenda

- 0. Re-Orienting
- 1. Von Neumann!
- 2. C to ISA Demo
- 3. Control Flow
- 4. The Stack Frame
- 5. The Address Space
- 6. Data Structures
- 7. Getting Strange

Reading:

- 2.0, 2.1.3 through 2.3.8
- 3.2 through 3.6

---

slides.pdf (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/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.

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).

- 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!

1. Von Neumann!

Slightly modifying the example from last class:

types2.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/types2.c)

```
        void subroutine(int x) {

          printf("a subroutine: address of x is %p\n", &x);

        }

        int main() {

          printf("address of the subroutine is %p\n", &subroutine);
          subroutine(42);

          return 0;

        }
```

What's up with the "&subroutine"?

```
        gcc -m32 -g -O0 -o types2 types2.c
```

Then with gdb, then we can stop after the printf.... and look at how the main code, the subroutine code, and the globals all live in the same address space. (Because Von Neumann won.... at least initially).

Or, to get really deranged, try

quine.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/quine.c)

2. C to ISA Demo

Fig 3.8 in the textbook:

ops.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/ops.c)

On my Mac, with -O1

- where's the AND? ("Move Zero-Extended Word to Long")
- what's going with multiplication?

How might bitwise-AND differ from Boolean AND?

and.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/and.c)

---

3. Control Flow

---

if-else

```
// Fig 3.13

int absdiff(int x, int y) {
  if (x < y)
    return y - x;
  else
    return x - y;
}
```

---

while>

```
// Fig 3.15

int fact_while(int n)
{
  int result = 1;
  while (n > 1) {
    result *= n;
```

```
    n = n-1;
  }
  return result;
}
```

for

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

conditional assignment

```
// Fig 3.12

int absdiff(int x, int y) {
  return x < y ? y-x : x-y;
}
```

We don't see a cmp instruction---because one of the subtractions we wanted to do anyway also served as our test.

What if the test hadn't so conveniently lined up?

condB.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20/-hll-p3/demo/condB.c)

switch

```
// Fig 3.18

int switch_eg(int x, int n) {
  int result = x;
```

```
    switch (n) {

    case 100:
      result *= 13;
      break;

    case 102:
      result += 10;
      /* Fall through */

    case 103:
      result += 11;
      break;

    case 104:
    case 106:
      result *= result;
      break;

    default:
      result = 0;
    }

      return result;
    }
```

4. The Stack Frame

Last class, we were looking at how a procedure accessed locals and args, and saw...

- Locals appeared to be negative offsets from %ebp (when they were in memory at all)
- Arguments appeared to be positive offsets from %ebp
- (So...can we use arguments as if they were local variables?)
- There also appeared to be other things going on relative to %ebp

Looking through caller() in my modified version of Fig 3-32

- We see locals off the %ebp
- We see arguments for the subroutine being placed near the top of the stack (in space that was reserved)?
- We see the return value coming back in %eax.

If we walk through the code step by step...

- We see the compiler making space for the locals, the subroutine arguments, the subroutine return address ...and other space, too!
- We see the callee accessing its arguments by reaching into the caller's stack frame

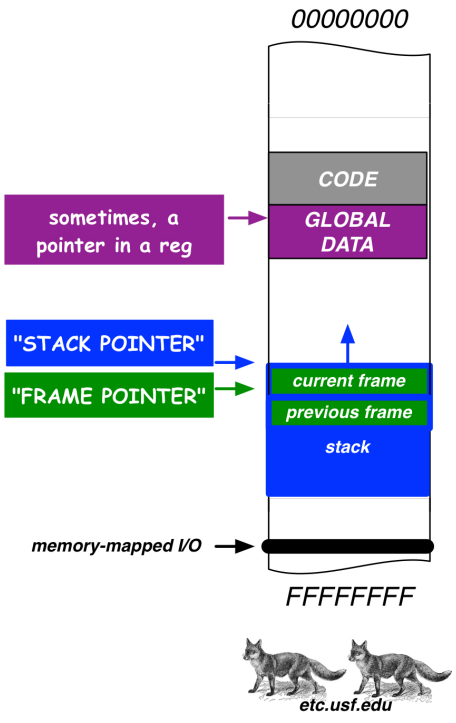
Architecturally...

- why do local variables have valid initial values
- but you shouldn't count on what they are
- and they may not necessarily be the same each time?

---

5. The Address Space

It's all in the same place:



---

6. Data Structures

Programmer's view:

- goal
- struct
- typedef
- foo.bar

examples

```
typedef struct {
  int field0;
  int field1;
  int field2;
} foo_t;

foo_t foo;

main() {

  foo_t *foop = &foo;

  foo.field0 = 1;
  printf("%d\n", (*foop).field0);

}
```

Hardware view

how does the structure get laid out in memory?

```
typedef struct {
  unsigned short short1;
  int int1;
  unsigned char char1;
  int int2;

} foo_t;

foo_t foo;

main() {



  printf("foo        lives at %p\n", &foo);
  printf("foo.short1 lives at %p\n", &(foo.short1));
  printf("foo.int1   lives at %p\n", &(foo.int1));
  printf("foo.char1  lives at %p\n", &(foo.char1));
  printf("foo.int2   lives at %p\n", &(foo.int2));
  printf("sizeof(foo) = %d\n", sizeof(foo));
```

```
        }
```

pack.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/pack.c)

- gcc
- gcc -fpack-struct

---

if foo_t *foop, then we can write foop->bar instead of (*foop).bar

---

7. Getting Strange

Do we have to use %ebp as a frame pointer?

Can locals live in registers?

Can we pass **data structures** as arguments? As return values?

Should we?

struct_arg.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/struct_arg.c)

struct_val1.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/struct_val1.c)

struct_val2.c (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/20-hll-p2/demo/struct_val2.c)