ECE 271 – Microcomputer Architecture and Applications Lecture 13

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

5 March 2019

Announcements

- Read Chapters 8 + 9
- Midterm, Tuesday, 12 March
  more info on that as it gets closer

Lab #6 Update

- Stepper motor, but in assembly
- Mostly learning to write functions in assembly

Lab #6 – Making code into a function

- Delay code in C

```
    for(i=0;i<6000;i++) ;
```

- An implementation

```
    mov r5,#6000
delay_loop:
    subs    r5,r5,#1
    bne delay_loop
```

- A more literal one (it takes longer, why?)

```
    mov r5,#0
```

```
delay_loop:
    add r5,r5,#1
    cmp r5,#6000
    bne delay_loop
```

- Conversion to function

```
    // Delay, with amount in r0
    // Can we keep using r5?  What if we didn't save r5?
    //  what value would it have on return?
    // what happens if we forget to pop?
Delay   PROC            // PROC not needed Linux
    push    {r5,lr}
    mov r5,r0
delay_loop:
    subs    r5,r5,#1
    bne delay_loop

    pop {r5,lr}
    bx  lr      // return
    ENDP
```

Using Arrays in Assembly

```
int steps[4]={0x00480084,0x00880044,0x00840048,0x00440088};
int current_step,i;

for(i=0;i<4;i++) {
    current_step=steps[i];
}
```

```
    mov r0,#0
loop
    ldr r1,=steps
    ldr r2,[r1,r0, LSL 2]

    add r0,r0,#1
    cmp r0,#4
    bge loop
```

```
steps
    DCD 0x00480084,0x00880044,0x00840048,0x00440088
```

Note on Linux use `.word` instead of `DCD`

Recursion

- Very CS thing to do
- Function calls itself
- ECE / embedded not like to do it much. Why? What happens when run out of stack?
- Can be useful. Think compilers?
- You'll see it in Google interviews

Factorial Example

- n! = n * (n-1) * (n-2) ... * 1
- Any sane person would implement it like

```
int factorial(int n) {
   int result=1;
   for(i=1;i<=n;i++) result*=i;
   return result;
}
```

Factorial via Recursion

- factorial(0) $=$ 1
- factorial(1) $=$ 1 $=$ 1*factorial(0)
- factorial(2) $=$ 2 $=$ 2*factorial(1)
- factorial(3) $=$ 6 $=$ 3*factorial(2)

Factorial Example – C

```c
int factorial(int n) {
   if (n<2) return 1;
   return (n*Factorial(n-1));
}
```

Factorial Example – Assembler

```
factorial
   push {r4,lr} // save r4 (why?) save lr (why?)
   mov  r4,r0   // copy input arg to r4
   cmp  r4,#2
   bge  else    // if 2 or greater skip ahead
   mov  r0,#1   // otherwise return 1
   b    factorial_exit
else
   sub  r0,r4,#1   // arg is oldarg-1
   bl   factorial
   mul  r0,r4,r0   // return value in r0
                   // multiply by r4 (which was saved across call)
factorial_exit
   pop  {r4,pc}    // why have only one exit to function?


_start
   mov r0,\#0x3
   bl factorial
```

```
stop
   b stop
```

TODO: draw diagram of stack?

Alignment

- Structs and alignment
- Why align variables in memory?
  - Memory is usually byte-addressable
  - ints are multi-byte (2, 4, 8 bytes?)
  - Can you have ints that start at odd addresses?
  - Older machines – no, caused an alignment fault. Either a crash, or else software had to slowly work around issue (do multiple loads, shifts, and ors)
  - x86 always supported unaligned loads, so to be

compatible more systems support it

  - it can still be bad for performance, especially if cross a cache line

- If you have something like

```
struct {
    int a;
    int b;
    int c;
} something;
```

  you can see alignment is easy. Also you can picture what the assembly looks like to load something.a, something.b or something.c

- What about

```
struct {
    int a;
```

```
        char b;
        int c;
    };
```

The compiler might add padding so int c is properly aligned.

- What is wrong with padding?

  Takes more RAM?

  Security (what ends up in padding? old data?)

  What if you are trying to match hardware registers or a file format w/o padding?
- You can force no padding. On Keil with `__packed` attribute.

- On Linux it is `struct __attribute__((__packed__))`

Chapter 9 – 64 bit values

- Adding – use carry bit
- Subtracting
- Multiply?
- Divide?
- Shifting – single shift, through carry
  Logical or Arithmetic
- Shifting by arbitrary, shift and mask.

```
0xdeadbeef  0xc001cafe

shift right by 16
0xdead, 0xbeef, 0xc00l, throw away cafe
0 | (0xdead>>16), (0xbeef<<16)|(0xcool>>16)
```