ECE 271 – Microcomputer Architecture and Applications Lecture 14

Vince Weaver

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

vincent.weaver@maine.edu

7 March 2019

Announcements

- Read Chapter 10
- Midterm, Tuesday, 12 March
- Reminder, no new lab next week, tie up loose ends in old labs

Lab #6 Update

- Hopefully it is going OK.
- Please try to catch up on the labs...

Midterm Review

- Closed book/notes.
- Short answer.
- Will be on mostly C and assembly language
- I expect you to know at least basic C
- This includes being able to set/clear bits using the bitwise logic operations
- Assembly language, I will provide a table of THUMB2 instructions so no need to memorize.
- Things like, what does this code do? Or, add comments

to this code, or, what is wrong with this code.

- GPIO, LCD, Scanning, Stepper
- Basic understanding of what the hardware is doing, but not super detailed
- No need to memorize all of the MODER register fields, etc.
- Questions may be similar to those on pre/post-lab
- One thing we did not have a question on but important to know is twos complement, and how calculation of overflow flag works
- Also stack. Where do local vars go?

- ABI, know why we have one. Where args go.

Chapter 10 – Mixing C and Assembler

- More ABI: data sizes
- Data alignment in structs
  what happens if unaligned?
- Can you force alignment to be packed? Why? `__packed`
  or other attribute

Var types

- static – make act as global
- volatile
- Local vars. Why encouraged not to use globals?

Symbols in Other Files

- When you use a name (variable name, function name, etc) the compiler/assembler doesn't necessarily resolve the value right away
- This can happen at link time. The object code might just have a placeholder value
- The linker resolves this when making the final executable
- When compiling/assembling you do have to let the code know the symbols are external and what they are like (often in #include files)

- As long as you follow the ABI though you can link against any object file, even one that was compiled long ago, or one you don't have the source for, or a system library.

Inline Assembly

- Inline assembly
- Note, gcc/Linux does this a lot more annoyingly (include example)

```
__asm int sum4(int a, int b, int c ,int d) {
    push {r4,lr}
    mov  r4,r0
    add  r4,r4,r1
    add  r4,r4,r2
    add  r0,r4,r3
    pop  {r4,pc}
```

```
int sum4(int a, int b, int c, int d) {
    int t;
    __asm {
        add t,a,b
        add t,c
        add t,d
    }
    return t;
}


ULong amd64g_dirtyhelper_RDPMC ( ULong counter) {
    UInt  eax, edx;
    __asm__ __volatile__("movq␣%[c],%%rcx;␣rdpmc"
                       : "=a" (eax), "=d" (edx) : [c] "r" (counter));
    return (((ULong)edx) << 32) | ((ULong)eax);
}
\begin{lstlisting}
```

Calling Assembly from C (10.4)

```
/* main.c */

extern int sum4(int a, int b, int c, int d);

int main(int argc, char **argv) {

    x=sum4(1,2,3,4);

    while(1);
}
```

This is where the ABI excels.

Linux directives are a bit different. `.globl`

```
/* sum4.s */

    EXPORT sum4
sum4    PROC
    push {r4,lr}
    mov  r4,r0
    add  r4,r4,r1
    add  r4,r4,r2
    add  r0,r4,r3
    pop  {r4,pc}
    ENDP
```

Strong and weak symbols?

Calling C from Assembly (10.4)

```
/* sum.c */

/* note, not declared static */
int sum2(int x, int y) {

    return x+y;

}



/* main.s */

    IMPORT sum2

    ENTRY

__main  PROC
```

```
    mov r0,#1
    mov r1,#2
    bl  sum2

stop    b   stop

    ENDP
```