ECE 271 – Microcomputer Architecture and Applications Lecture 9

Vince Weaver

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

vincent.weaver@maine.edu

15 February 2022

Announcements

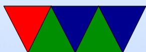
- Read Chapter 3.5 to 3.7
- Lovebyte Update
- Lab schedule update

  Finish Lab#3 (keypad) as soon as possible

  Start Lab#4 as soon as possible

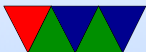
  Next week is catch-up lab to finish any outstanding labs

  Note no Monday lab on Monday due to President's day



Lab #4 Notes

- Redo Lab#1, but in Assembly language



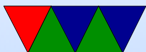
Lab #4 Notes

We want to convert the following C to assembly

```
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
```

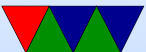
A lot is going on in provided code. Some low-level C and preprocessor directives are used to make RCC be a pointer to address 0x40021000. A cast is used to make this pointer be of type RCC_Typedef which has the offsets for the various sub-registers (see next slide)

```
#define PERIPH_BASE      ((uint32_t)0x40000000)
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x00020000)
#define RCC_BASE         (AHB1PERIPH_BASE + 0x1000)
#define RCC              ((RCC_TypeDef *)(RCC_BASE))
```



Lab #4 Notes

- Do we remember how structs work in C?
- What's the difference between `a.b` vs `a->b`
- The latter is indexing from a pointer

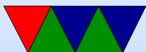


Lab #4 Notes

The RCC_Typedef is in the provided stm32l476xx.h

```
typedef struct {
   __IO uint32_t CR;          // Control Register, offset 0x00
....
   __IO uint32_t AHB2ENR;     // AHB2 periph control, offset 0x4c
....
} RCC_Typedef;
```

The key understanding is that we want to access the 32-bit value that's at offer 0x4c from the beginning of the base

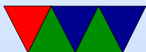


Lab #4 Notes

The code that implements the register setting can be done like this:

```
ldr     r1,=RCC_BASE              ; r1=&RCC
ldr     r3,[r1,#RCC_AHB2ENR]      ; r3=RCC->AHB2ENR
orr     r3,#RCC_AHB2ENR_GPIOBEN   ; r3 = r3 | RCC_AHB2ENR_GPIOBEN
str     r3,[r1,#RCC_AHB2ENR]      ; RCC->AHB2ENR = r3
```

We provide defines in assembly which you can use for value/masks rather than having to do the raw hex codes.



Clearing values

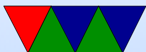
- use the BIC instruction to clear bits
  Why not just use AND? Because a mask has lots of 1s might not fit in available constant room

```
and r3,#0xfffffffe          ; wont fit in instruction, too big
bic r3,#1                   ; same as: and r3,#~1 but fits in constant
```

- You can usually include complicated C-stye constant manipulations, things like

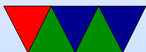
```
and r3,r3,#(GPIOBEN + 0x5 | (1<<3))
```

  though note that the Keil assembler might not support the full range



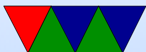
Something-cool Notes / BSRR

- Using the BSRR register to set/reset GPIO pins without having to do a read/modify/write.
- Atomic operation?
- Bit set/reset Register.
  - Write a bit pattern of 0 or 1
  - 0 means leave alone
  - In bottom 16-bits, 1 means set that GPIO to 1
  - In the top 16-bits, 1 means reset that GPIO to 0



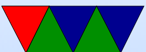
Assembler – Code comments

- Can use C and C++ style comments
- Keil: can use ; (makes rest of line a comment)
- Linux/gas: Can use @ for beginning of line



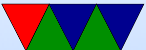
Functions/Subroutines

- Why use them?



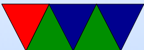
Sample C

```c
int sum(int a, int b) {
   return a+b;
}

int main(int argc, char **argv) {
   int result;
   result=sum(1,2);
}
```



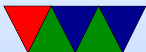
Sample Assembly

```
sum
     add    r0,r1,r2    ; result=arg1+arg2
     bx     lr          ; jump to saved address in link register

main
     mov    r0,#1       ; set arg1 to 1
     mov    r1,#2       ; set arg2 to 2
     bl     sum         ; call sum function, put current
                        ; program counter+4 into link register
```



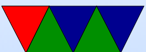
Subroutines on ARM

- `bl` branch and link instruction
  - Sets the link register LR (r14) as the memory address of the next instruction immediately after the BL (PC+4 on Thumb-2)
  - Adjust the PC to be the memory address of the first location of where we want to transfer execution
- After executing, LR has the return address



Returning from a Subroutine

- Use the `BX LR` instruction, which says to branch to the address located in the LR register. (the X is for exchange; historical THUMB reasons)



Saving/Restoring values in functions

- To preserve registers at start of function can

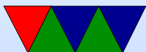
```
push {r0, r1, lr}
```

- You will want to save the link register if not a leaf function (meaning, you call another function from inside)
- At end you can

```
pop {r0, r1, lr}
```

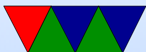
  before using `bx lr` to return
- Alternately, if the LR register was pushed on the stack, you can use the clever hack of doing

```
pop {r0, r1, pc}
```



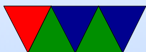
to pop the link register directly into the PC to return without the extra branch

; p164

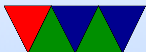


The ABI – The Application Binary Interface

- A Document, produced often by the processor maker
- An agreement of how functions / code talk to each other
- A common standard so compilers, libraries, and code can call each other and know how to set things up
- Useful to have for your own code. Might be slightly less efficient, but better than for every function you call having to save/setup different registers
- What kinds of things are included?
  - What registers to put things in? Register allocation?

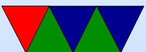


- Alignment of stack (4 bytes? 8 bytes?)
- How to pass 8/16/32/64 byte values
- How to pass floating point values
- Where does the return value go?
- System calls
- Frame pointer



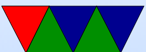
ARM ABI

- On Linux there have been at least 4
- armbe – big endian
- armle – little endian
- EABI – extended (new) ABI
- armhf – EABI but fancier (hard) floating point support



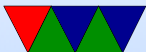
Calling Conventions

- r0/r1/r2/r3
  - parameters/scratch
  - caller saved, so if you want the value in say r3 to be the same after a function call you have to save it/restore it to memory
  - r0/r1 also used as return value from function
- r4/r5/r6/r7/r8/r10/r11 = variables
  - callee saved. You can count on this having the same value after a function as before. If you are in a function



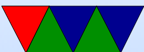
and want to use it, must save/restore it. Often this done at function entry/exit

- r9 – implementation dependent (thread-local register?)
- r12 = reserved by linker?
- r13 = stack pointer
- r14 = LR (link register)
- r15 = PC (program counter)



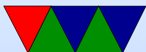
Calling Conventions – Corner Cases

- Return value in r0. Might be in r1 or more if bigger than 32 bits
- What happens if more than 4 arguments?
- What happens if more than 32-bits (use 2 registers, even/odd for 64-bits)



Calling Conventions – Corner Cases

- How do you pass something complicated like an array or struct?
- Call by value or by reference
- Can pass a pointer in a 32-bit register

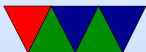


Disassembler

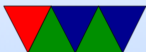
```
#include <stdio.h>
int i;
int main(int argc, char **argv) {
        for(i=0;i<100;i++) {
                printf("Hello!\n");
        }
        return 0;
}
```

```
gcc -Wall -mthumb -march=armv7-a -o test test.c
```

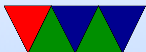
```
Disassembly of section .bss:

0002102c <i>:
   2102c:       00000000        andeq   r0, r0, r0

Disassembly of section .rodata:

000104fc <_IO_stdin_used>:
   104fc:       00020001        andeq   r0, r2, r1
```



```
   10500:       6c6c6548        cfstr64vs       mvdx6, [ip], #-288      ; 0xfffffee0
   10504:       Address 0x00010504 is out of bounds.

0001043c <main>:
   1043c:       b580            push    {r7, lr}
   1043e:       b082            sub     sp, #8
   10440:       af00            add     r7, sp, #0
   10442:       6078            str     r0, [r7, #4]
   10444:       6039            str     r1, [r7, #0]
   10446:       f241 032c       movw    r3, #4140       ; 0x102c
   1044a:       f2c0 0302       movt    r3, #2
   1044e:       2200            movs    r2, #0
   10450:       601a            str     r2, [r3, #0]
   10452:       e010            b.n     10476 <main+0x3a>
   10454:       f240 5000       movw    r0, #1280       ; 0x500
   10458:       f2c0 0001       movt    r0, #1
   1045c:       f7ff ef42       blx     102e4 <puts@plt>
   10460:       f241 032c       movw    r3, #4140       ; 0x102c
   10464:       f2c0 0302       movt    r3, #2
   10468:       681b            ldr     r3, [r3, #0]
   1046a:       1c5a            adds    r2, r3, #1
   1046c:       f241 032c       movw    r3, #4140       ; 0x102c
   10470:       f2c0 0302       movt    r3, #2
   10474:       601a            str     r2, [r3, #0]
```

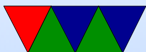


```
10476:    f241 032c    movw   r3, #4140       ; 0x102c
1047a:    f2c0 0302    movt   r3, #2
1047e:    681b         ldr    r3, [r3, #0]
10480:    2b63         cmp    r3, #99 ; 0x63
10482:    dde7         ble.n  10454 <main+0x18>
10484:    2300         movs   r3, #0
10486:    4618         mov    r0, r3
10488:    3708         adds   r7, #8
1048a:    46bd         mov    sp, r7
1048c:    bd80         pop    {r7, pc}
```

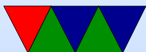


Assembler directives

- AREA – declare a new area (code/data/bss)
  `AREA myData, DATA, READWRITE`
- ENTRY – declare entry point into the code
  You might think this is "main" on C, but actually it is usually somthing called _start, a lot of things happen before main() gets called
- ALIGN – align the current memory address
  for performance, or some things (like variables on stack) must be aligned



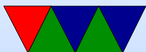
- DCB – reserve space for bytes
  array DCB 1,2,3,4
  hello DCB "Hello World!",0
- DCW – reserve space for 16-bit values
- DCD – reserve space for 32-bit values
- DCFS/DCFB – floating point
- SPACE – restore unreserved data (BSS)
  p SPACE 255 ; reserve 255 zeros
- FILL – reserve space and fill it with a value
  f FILL 20,0xff,1 ; allocate 20 bytes, fill with 0xff
- EQU – like #define in C, lets you set constants.



MAXCPUS EQU 8

On Linux same, but `.equ MAXCPUS, 8`

- RN – alias a register name, if you want to use something like X instead of R3
- EXPORT/IMPORT – export says to make symbol globally visible (.globl on Linux). Import says a symbol is external, like "extern" on C.
- INCLUDE/GET – like include directive in C, includes another file when assembling
- PROC/ENDP – start and end of function. Mostly to make debugging easier? Doesn't actually change



generated code?