ECE 598 – Advanced Operating Systems
Lecture 4

Vince Weaver

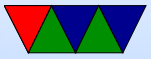
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

22 January 2015

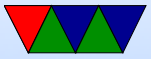
Announcements

- Let me know if you need to borrow a Pi



Writing a standalone Program

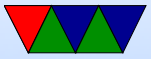
- Easy in assembler
- Some Extra work in C. Why?



Entry Point from Bootloader

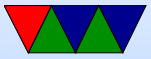
- Execution starts at 0x8000

- Loader passes a few arguments, as in a function call. Three arguments. As per ABI in r0,r1,r2
  r0=device booted from (usually 0)
  r1=arm chip identifier (3138 0xc42 on bcm2835)
  r2=pointer to ATAGS (arm TAGS) which contains info from bootloader, such as memory avail, etc.



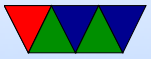
Building

- We're going to use a cross-compiler to build the homework. Then copy the result to the SD card. There are other ways to do this (including building natively on the pi) but those involve a lot of cable and/or card swapping.
- Unfortunately debugging is a pain if not working.
- Would be nice if we had a fancy bootloader that allowed dual boot, but I was unable to find a good one.



Blinking an LED

- On Model B, GPIO16 is connected to the ACT LED.
- On Model B+, it is GPIO47 instead

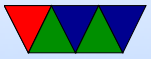


GPIOS

- See the peripheral reference available here:
  `http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf`

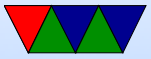
- GPIO base at 0x20200000

- The GPFSEL registers let you enable the GPIO pins.
  9 GPIOs per register (3 bits each)
  GPIO16 is thus set in GPFSEL1
  A value of "1" enables it for output (what do other



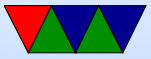
values do?)

- The LED is hooked up to active low, so want to clear the line to turn on (on B+ it is the opposite).
- GPSET registers used to set to 1. So to set GPIO16 to on, set bit 16 of GPSET0 register.
- GPCLR registers used to set to 0
- Can do much fancier things. Set alternate functions for the outputs, pullups, read values, level detect, etc. Much like in ECE271.



Assembly review

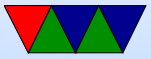
- ARM has 16 registers. r0 - r15. r15 is the program counter. r14 is the stack pointer.
- arm32 has fixed 4-byte encoding (rpi also has THUMB but we won't be using that).



Defines

The .equ assembler directive is the equivalent of a C `#define`

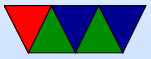
```
.equ GPIO_BASE,         0x20200000
.equ GPIO_GPFSEL1,      4
.equ GPIO_GPSET0,       28
.equ GPIO_GPCLR0,       40
```



Loading a Constant

You can use `mov r0,#2048` to load small constants (# indicates an immediate value). However long constants won't fit in the instruction coding. One way to load them is to put = in front which tells the assembler to put the value in a nearby area and do a PC-relative load.

```
ldr     r0,=GPIO_BASE
```

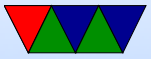


Storing to a Register

There are always multiple ways to generate a constant. In this case we want a 1 shifted left by 18. A simple way to do this is load the value, then logical shift left it to the right position.

The str instruction stores. In this case we have two values, so the value pointed to is the sum of the two.

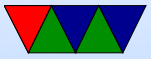
```
mov     r1,#1
lsl     r1,#18
str     r1,[r0,#GPIO_GPFSEL1]
```



Delaying

A simple way to create a delay is to just have a busy loop. Move a value in, and then decrement the counter until it hits zero. You can use a separate `cmp` instruction for the compare, but ARM allows you to put "s" on th end of an instruction to update flags. Thus below the `sub` instruction will update the zero flag after each iteration, and the `bne` branch-if-not-equal will check the zero flag and loop properly.

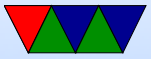
```
    mov r1,#65536
delay_loop:
    subs    r1,r1,#1
    bne delay_loop
```



Looping Forever

Once our program ends we cannot exit like you normally would; there's no operating system to exit to. To prevent the program just running off the end of the address space we have an infinite loop. ARM processors support the `wfe` instruction which will put the CPU in a low-power state while waiting for something to happen. This will use less power (hopefully) than an empty busy loop.

```
finished:
        wfe                     /* wait for event */
        b       finished
```

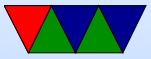


More Blinking, Now in C

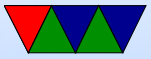
C is easier to program, but has more overhead.

Other things to note:

- Need to compile with `-nostartfiles` as no C library is available.
- You need to provide own C library routines. No printf, strcpy, malloc, anything like that.



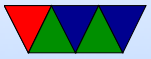
- There needs to be boot code to set up the stack, initialize the BSS, etc.



More Blinking, Now in C

You can set up some useful #define statements to make the code easier to follow.

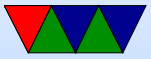
```
#define GPIO_BASE       0x20200000UL
#define GPIO_GPFSEL1    1
#define GPIO_GPSET0     7
#define GPIO_GPCLR0     10
```



Volatile!

The volatile keyword tells the compiler that this address points to something that might change, so should actually be read every time a read is indicated. An optimizing compiler otherwise might notice two reads to an address with no intervening store and optimize away the first read! It may also optimize all but the last store if no intervening reads!

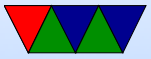
```
volatile uint32_t *gpio;
```



Setting a value

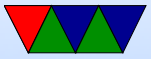
You can treat memory as an array.

```
gpio[GPIO_GPFSEL1] |= (1 << 18);
```



Delays

If you want to use an empty delay loop like we do in asm you'll have to use volatile or otherwise find ways to keep the compiler from optimizing it away.



Building

- Linker script (tells linker where to put things, sets up entry point, etc)
- Objcopy to strip off extraneous ELF header stuff.