ECE 598 – Advanced Operating Systems
Lecture 4

Vince Weaver

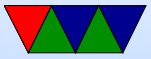
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 February 2018

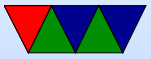
Announcements

- HW#1 was due
- HW#2 was posted, will be tricky, can work in groups?
- Let me know if you need to borrow a Pi
- E-mail from Linus
- Perpetual debug issues in vmwOS



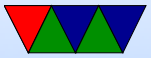
Raspberry Pi

- One of the first *cheap* ARM development boards
- Put out by the Raspberry Pi Foundation in England
- Meant for educational use, widely used by hobbyists
- Designed to get students interested in low-level computing like the old days.
- Model names based on old BBC Micro computers

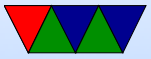


Raspberry Pi-1 A/B/A+/B+/zero

- BCM2835 SoC
- ARM1176 – v6, older than the v7 (Cortex A8, A9, A15)
- 700MHz (overclock?), 1-issue in-order, VFPv2 (no neon), DSP
- 256MB-512MB RAM
- 16k 4-way l1 i/d cache, 128kb L2 controlled by vcore (linux configs for cpu)
- VideoCore IV (24Gflops) GPU
- ARM32 and THUMB; no THUMB2

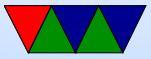


- Powered by USB-micro connector
- A models lack Ethernet and have fewer USB
- Plus models have 40 pin GPIO header (instead of 26), better power converter, more USB, combined composite/sound port, re-arranged some internal GPIOs
- Zero model even more stripped down, mini-HDMI



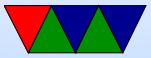
Raspberry Pi Model 2 (original)

- SoC (mostly) the same, but much faster processor
- Board layout just like B+
- BCM2836 (newer models have BCM2837)
- quad-core ARMv7 Cortex A7
- 1GB RAM (so all peripheral addresses move)
- Has THUMB2 support

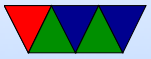


Raspberry Pi Model 3

- SoC still mostly the same
- BCM2837 – Quad-core 64-bit ARMv8 Cortex A53
- Higher performance, but draws more power, overheats and can even crash if you run it too hard.
- Has bluetooth, uses up the primary serial port (pl011) so serial output now stripped-down mini-uart
- Wireless Ethernet, many internal GPIOs used by this so

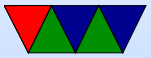


a VC-controlled i2c GPIO extender used for some things (like ACT light)

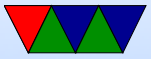


BCM2835 SoC features

- Peripherals start at 0x20000000 (512MB) phys address as seen by CPU (GPU seems them at 0x7e000000)
  On 1GB Models (pi2+pi3) at 0x3f000000 (just below 1G)
- GPIO
- Primary pl011 UART
- bsc, aka i2c
- dma controller
- emmc

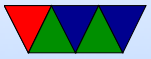


- gpio
- interrupts (mailbox, doorbell)
- pcm/i2s audio
- pwm
- spi
- spi/i2c slave
- system timer
- mini-uart (serial port)
- arm timer
- usb
- video: HDMI?,composite



Writing a standalone (bare-metal) Program

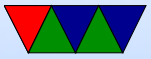
- Easy in assembler
- Some Extra work in C. Why?



Entry Point from Bootloader

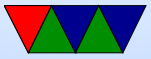
- Execution starts at 0x8000

- 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 system config, device-tree (newish) or ATAGS (arm TAGS). ATAGS usually loaded at 0x100



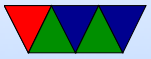
Building

- You will need to set up a cross-compiler
  Link to directions. Windows, OSX, Linux.
  Have done the Linux and OSX versions. People last year successfully did on Windows.
  Old days much harder, had to compile gcc cross-compiler from scratch, quite a pain.
- Then edit your code, then cross compile.
- Once the image is built, you will copy it to a memory key that has Linux on it. On the /boot partition, over-write



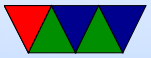
the `kernel.img` file with your image.

- Then reboot.
- There are various ways/tools you can use to edit your code so you might want to experiment. You can also build the code "natively" on a Pi but that would involve some manner of transferring the file to get it on an SD card.
- 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. Can install uboot if you want.



Blinking an LED – Should be Easy?

- On Model B, GPIO16 is connected to the ACT LED (active low)
- On Model B+/A+/2, it is GPIO47 (active high instead)
- On Model 3 it's connected to an i2c GPIO extender controlled by the VideoCore :(
- For homework purposes, we will blink GPIO18 (on the expansion header) and either hook an LED or Digilint or similar to verify.

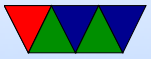


GPIOs

- See the peripheral reference available here:

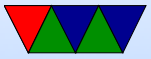
  `http://web.eece.maine.edu/~vweaver/classes/ece598_2015s/BCM2835-ARM-Peripherals.pdf`

- Look in Chapter 6
- On the Pi2/Pi3, memory mapped I/O starts at `0x3f000000` (just before 1GB)
- The GPIO base is at `0x3f200000` (the documentation confusingly lists it as `0x7e200000`, just replace the leading 0x7e with 0x3f).



Enabling a GPIO pin

- The GPFSEL registers let you enable the GPIO pins. 10 GPIOs per register (3 bits each). GPIO0 is `GPFSEL0` bits 0-2, GPIO1 is `GPFSEL0` bits 3-5, etc.
- A value of '0' in GPFSEL makes it an input, '1' enables it for output (what do other values do?)
- GPIO16 is thus `GPFSEL1`, bits 18-20
  GPIO47 is what? (`GPFSEL4`, bits 21-23)
  GPIO18 is what? (`GPFSEL1`, bits 24-27)
- So to set the value, first clear it to zero

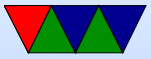


something like:

```
gpio[GPFSEL0]&=~(0x7<<18);}
```

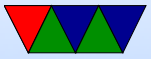
then set the value:

```
gpio[GPFSEL0]|=(1<<18);}
```



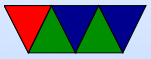
Setting a GPIO value

- We want to toggle the LED, so set the GPIO line high or low.
- GPSET registers used to set to 1. So to set GPIO18 to on, set bit 18 of GPSET0 register.
- GPCLR registers used to set to 0. So to set GPIO18 to off, write '1' to bit 18 of GPCLR0 register.
- Can do much fancier things. Set alternate functions for the outputs, pullups, read values, level detect, etc. Much like in ECE271.



ARM Assembly review

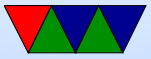
- ARM has 16 registers. r0 - r15. r15 is the program counter. r14 is the stack pointer.
- arm32 has fixed 4-byte encoding (rpi also has THUMB but we won't be using that).



Defines

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

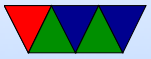
```
.equ GPIO_BASE,         0x3f200000

.equ GPIO_GPFSEL1,      0x04
.equ GPIO_GPSET0,       0x1c
.equ GPIO_GPCLR0,       0x28
```



Loading a Constant

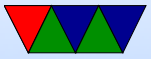
You can use `mov r0,#2048` to load small constants (# indicates an immediate value). However long constants won't fit in the instruction coding. One way to load them is to put = in front which tells the assembler to put the value in a nearby area and do a PC-relative load.

```
ldr     r0,=GPIO_BASE
```



Logical Operations

```
and r1,#1024
orr r2,#2048
```

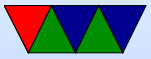


Storing to a Register

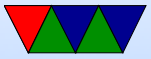
There are always multiple ways to generate a constant. In this example we want to shift 1 left by 24. A simple way to do this is load the value, then logical shift left it to the right position.

The `str` instruction stores a register to memory. The second argument is the address; there are many possible addressing modes, the one we are using adds a constant offset to an address in a register.

```
mov     r1,#1
lsl     r1,#24
str     r1,[r0,#GPIO_GPFSEL2]
```



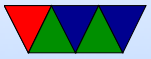
Can you instead do `mov r1,#(1<<24)`?



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 the 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.

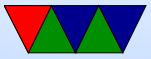
```
    mov r1,#65536
delay_loop:
    subs    r1,r1,#1
    bne delay_loop
```



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
```

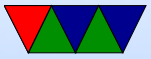


More Blinking, Now in C

C is easier to program, but has more overhead.

Other things to note:

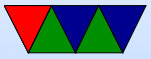
- Need to compile with `-nostartfiles` as no C library is available.
- You need to provide own C library routines. No printf, strcpy, malloc, anything like that.
- There needs to be boot code to set up the stack, initialize the BSS, etc.



More Blinking, Now in C

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

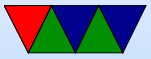
```
#define GPIO_BASE       0x3f200000UL
#define GPIO_GPFSEL1    1
#define GPIO_GPSET0     7
#define GPIO_GPCLR0     10
```



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!

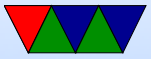
```
volatile uint32_t *gpio;
```



Setting a value

You can treat memory as an array.

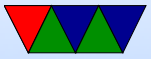
```
gpio[GPIO_GPFSEL1] |= (1 << 18);
```



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.

gcc keeps getting better at this. Currently have to tell it to not inline the code with `void __attribute__ ((noinline)) delay(int length)` and then in the inner loop of your do-nothing function you will want to put something like `asm("");` which tells the compiler not to optimize it away.



Building

- Linker script `kernel.ld` (tells linker where to put things, sets up entry point, etc)
- By default an ELF executable is generated; the `objcopy` program strip off extraneous ELF header stuff leaving just the raw executable.