ECE 531 – Advanced Operating Systems
Lecture 5

Vince Weaver

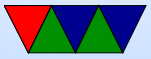
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

12 September 2025

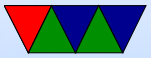
Announcements

- HW#1 due today
- HW#2 will be posted (we'll discuss in class today)



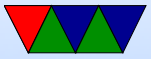
Kernel booting Summary

- Initializes hardware. First part asm. Transition to C as quickly as possible. First thing to initialize. Memory. Then simple in/out. Enable keyboard, simple VGA, serial console. So printk can work.
- Relocates decompression code
- Decompresses
- Parse the resulting ELF file.
- Apply any relocations
- Jump to entry point



Raspberry Pi Booting

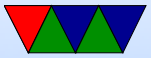
- Unusual (and has changed over the past few years)
- Small amount of firmware on SoC
- Pi is actually a large GPU chip with helper ARM chips
  - VPU – Dual Core Videocore IV chip CPU, SIMD, Parallel Processor, ThreadX OS, co-ordinates everything, as well as video codecs, power, etc.
  - ISP – Image Sensor pipeline, processing for the cameras
  - QPU – Quad Processor Unit – 24 GFLOP compute pipeline, co-ordinate and vertex shader.



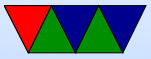
Raspberry Pi Booting – First Stage

- Power on – First stage – Boot ROM in GPU starts up GPU runs things. Draws rainbow pattern, lightning bolts, etc
- First stage tries to load `bootcode.bin` second stage
  - On original Pi this was just SD card
  - Pi3 can boot off of Secondary SD, SPI, NAND, USB or network.

    Also some complex hack to flash the OTP (?) to allow GPIOs to be dedicated to boot-selection



- Pi4 and later this might live on eeprom
  If fails, special rescue image on SD card can fix
- bootcode.bin loaded into L2 cache (shared CPU/GPU?) and executed

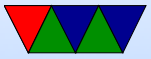


Raspberry Pi Booting – Second Stage

- bootcode.bin – binary blob that is loaded from the SD card and run by GPU. 1 million+ lines of code? Mostly written and maintained by one guy at Broadcom?
  - Has own non-ARM assembly language
  - Efforts to reverse engineer:

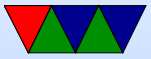
    `https://github.com/christinaa/rpi-open-firmware`

    `https://github.com/hermanhermitage/videocoreiv`
  - Inits SDRAM, gets ARM chips ready (if multiple, puts them in low-power sleep loop)

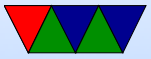


Raspberry Pi Booting – Third Stage

- Third stage – start.elf (cd, db, x)
  (used to be an additional stage before this)
  Loads and parses `config.txt`
  Lots of settings in config.txt
  - cd = cut down, if only 16MB of GPU memory specified
    ARM and GPU share RAM. Cuts out OpenGL, etc.
  - x = extra (have things like video codecs, camera)
  - db = debug, extra asserts
  - Also fixup.dat (cd, db, x) – used for configuring

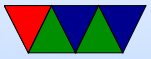


memory split?



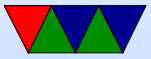
Raspberry Pi Booting – Kernel

- Now 4 kernels
  - kernel.img = old 2835 ARMv6 systems
  - kernel7.img = 2836 ARMv7 systems (32-bit)
  - kernel7l.img = 2711 (pi4, but 32-bit compat)
  - kernel8.img = pi3/pi4 64-bit, set this in config.txt
- Device tree files, .dtb.



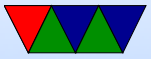
Raspberry Pi Booting – Device Tree

- Need way to describe hardware. Originally ARM ATAGS
- New way is common across all ARM systems, flattened device tree
- Server-class ARM systems might have ACPI (x86/windows compatible instead)
- On boot r2 pointer points to device tree, needs to be parsed
- Have things like location of hardware in memory, which IRQ to use, which GPIOs to use, how much memory is



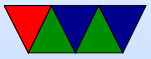
free and where it is, what type of CPU, etc.

- By standardized format, can have “generic” linux kernel that can run on any ARM generation without having to hard code that all.



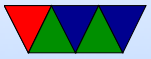
Raspberry Pi kernel/firmware locations

- Usually in /boot (separate FAT partition)
- Various kernel.img files are kernel
- kernel modules (drivers) in /usr/lib/modules
- initramfs (ramdisk) loaded at boot time, holds various files and drivers needed to get system going enough so it can find the rest of the modules on disk (compressed cpio file)
- config.txt (pi-specific) configures system
- .dtb files – device tree files describing system hardware



Writing a standalone (bare-metal) Program

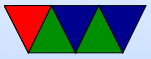
- Easy in assembler
- Some Extra work in C.
  Even in a low-level language like C you have to do some tricks to talk to hardware directly



Entry Point from Bootloader

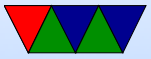
- Execution starts at 0x8000

- 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
  (years ago it pointed to ARM TAGS (ATAGS)



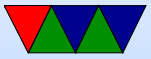
Building / Cross-Compiling

- You will need to set up a cross-compiler
  - You can find directions online, homework will link to some
  - Should be possible from Windows, MacOS, Linux
  - Past years people have gotten all three to work
  - In old days much harder, had to compile gcc cross-compiler from scratch, quite a pain.
- Edit your code, then cross compile.



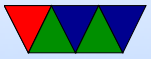
More Cross Compiler Setup

- If you aren't developing on ARM system will have to install a compiler that can generate ARM code
- An architecture to target is specified by triplets
- In this class usually use `arm-none-eabi` though the `arm-none-linux-gnueabihf` might be closer
- If running Linux/Debian installing as simple as `apt-get install gcc-arm-none-eabi`
- For Windows/MacOS can download official ARM toolchain from their website `https://developer.arm.`



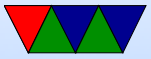
`com/downloads/-/arm-gnu-toolchain-downloads`

- When compiling with the homework if it can't find your cross-compiler you might need to edit Makefile.inc and point it to the right place



Copying to SD Card

- 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 proper kernel image
- For this class on Pi-1B+ it is `kernel.img`
- Note: backup `kernel.img` if you want the possibility of booting back to Linux again
- Then reboot.

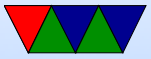


Notes on Code Development

- The most straightforward way is to edit/cross-compile on a development machine and copy the results over
- This involves a lot of SD-card swapping. Are there alternatives?
- One way would be to set up "dual boot" on the pi, compile code natively on Linux, and then reboot and select Linux or your code

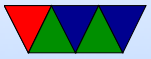
  Unfortunately it is *really* hard to dual boot on a Pi

  In theory you can maybe install uboot, would be



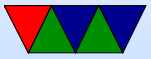
interested in hearing feedback if anyone ever got it working

- Another way on Pi3 or newer would be to network boot and have your kernel image on a server somewhere
- The Pi bootloader does support some limited dual boot with the `auto_boot.txt` file, but this is complex.



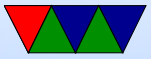
HW#2, Blinking an LED – GPIOs

- On embedded systems there are GPIO ports (general purpose I/O) that you can hook devices to
- By writing the correct registers you can change these output pins high / low
- The pins are multiplexed (a SoC has lots of features, but only so many pins so you have to select what does what)
- The Raspberry Pi has a few LEDs connected to GPIOs that we can blink



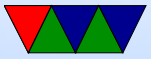
Blinking an LED – Which GPIO?

- Unfortunately different models use different GPIOs
- On Model 1B, GPIO16 is connected to the ACT LED (active low)
- On Model 1B+/1A+/2, it is GPIO47 **(active high instead)**
- On Model 3 it's connected to an i2c GPIO extender controlled by the VideoCore :((GPIO130)
- On Model 4 it's back but connected to GPIO42



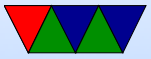
Memory Mapped I/O

- On many computers the peripherals are accessed via special memory accesses
- There's a range of address space that is treated like I/O devices rather than RAM
- The Pi has a complex memory map, but the memory-map region (IO_BASE) is above the RAM



Raspberry Pi IO_BASE

- On a 1B+ this is just above 512MB at 0x2000.0000
- On later Pis with more RAM it can be at 0x3f00.0000 or 0xfe00.0000
  (it's more complex on 64-bit systems)
- To confuse things the documentation is written from the GPU's perspective which lists everything as it it's at 0x7e00.0000

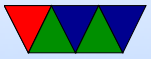


GPIO Interface

- See the peripheral reference available here:

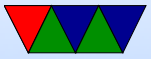
  `https://web.eece.maine.edu/~vweaver/classes/ece598_2015s/BCM2835-ARM-Peripherals.pdf`

- Look in Chapter 6
- The GPIO base is at IO_BASE + 0x200000
  confusingly lists it as 0x7e200000, just replace the leading 0x7e with 0x20, 0x3f, or 0xfe depending on your Pi model).



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. (TODO: diagram)
- A value of '000' in GPFSEL makes it an input, '001' enables it for output (what do other values do? see the manual)



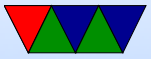
Configuring a GPIO pin for Write in C

- GPIO16 is thus `GPFSEL1`, bits 18-20
  GPIO47 is what? (`GPFSEL4`, bits 21-23)
  GPIO18 is what? (`GPFSEL1`, bits 24-26)
- So to set the value for GPIO16, first clear it to zero something like:

  ```
  gpio[GPFSEL1]&=~(0x7<<18);
  ```

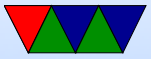
  then set the value:

  ```
  gpio[GPFSEL1]|=(1<<18);
  ```



Setting a GPIO pin value

- To toggle the LED, set the GPIO line high / low
- GPSET registers are used to set a pin to 1.
  For these registers there are 32 bits, so in GPSET0 each bit 0..31 corresponds to GPIO0..GPIO31
- So to set GPIO16 to on, set bit 16 of GPSET0 register.
  `gpio[GPSET0]|=(1<<16);`
- Note: there's no need to load/mask before doing this, the interface is designed so values of 1 are written and values of 0 ignored

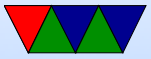


Clearing a GPIO pin value

- GPCLR registers are used to clear a pin to 0.
- So to set GPIO16 to on, set bit 16 of GPCLR0 register.

```
gpio[GPCLR0]|=(1<<16);
```

- As with setting, there's no need to load/mask/save as the interface preserves the values

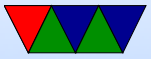


More Blinking in C

C is easier to program, but has more overhead.

Things to note:

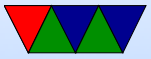
- Need to compile with `-nostartfiles` as no C library is available.
- You need to provide own C library routines. No printf, strcpy, malloc, anything like that.
- There needs to be boot code to set up the stack, initialize the BSS, etc.



C Helper Defines

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

```
// note different on pi models
#define IO_BASE      0x20000000UL
#define GPIO_BASE         (IO_BASE+0x200000)
#define GPIO_GPFSEL1      1
#define GPIO_GPSET0       7
#define GPIO_GPCLR0       10
```

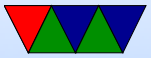


Volatile!

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!

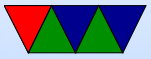
```
volatile uint32_t *gpio;
gpio=(uint32_t *)GPIO_BASE;
```



Setting a value

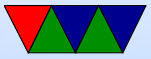
You can treat memory as an array.

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



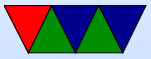
Delays

- The proper way is via timers, but that's beyond what we want to do in this homework
- A regular empty loop will be optimized away by the compiler (even if you put extra volatiles in) gcc keeps getting better at this.
- Currently what seems to work is:
  - tell it to not inline the code with `void __attribute__ ((noinline`
  - Have a loop, but instead of leaving it empty put `asm("");` which the compiler can't optimize away



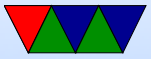
Building our Image

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



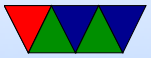
Startup Code

- Even regular C code doesn't jump straight to `main()`, there's code at `_start` that runs first to set up things like libraries and such
- I provide `boot.s`
- Sets up the stack (when you boot the boot loader doesn't necessarily do it for you)
- Clears the bss (un-initialized variables) to zero
- Jumps to `main()`. Still has r0..r2 set



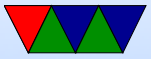
What if you want to do this in Assembly Language

Of course you do!



ARM Assembly review

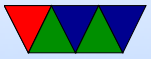
- ARM has 16 registers. r0 - r15. r15 is the program counter. r14 is the stack pointer.
- arm32 has fixed 4-byte encoding (rpi also has THUMB but we won't be using that).



Defines

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

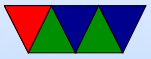
```
.equ GPIO_BASE,         0x20200000

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



Loading a Constant

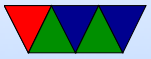
You can use `mov r0,#2048` to load small constants (# indicates an immediate value). However long constants won't fit in the instruction coding. One way to load them is to put = in front which tells the assembler to put the value in a nearby area and do a PC-relative load.

```
ldr     r0,=GPIO_BASE
```



Logical Operations

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

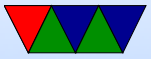


Storing to a Register

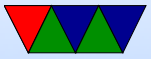
There are always multiple ways to generate a constant. In this example we want to shift 1 left by 24. A simple way to do this is load the value, then logical shift left it to the right position.

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]
```



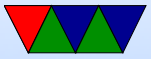
Can you instead do `mov r1,#(1<<24)`?



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
```