ECE 531 – Advanced Operating Systems Lecture 6

Vince Weaver

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

vincent.weaver@maine.edu

15 September 2025

Announcements

- Don't forget HW#2

HW#2 Notes – Cross Compilers

- This is often the biggest hurdle
- If on x86 Linux laptop/desktop, ideally you can just install the cross-compile toolchain bundled with your distro
- MacOS using the binaries from the ARM site (see the homework handout) seem to work OK
- Haven't tested Windows, it's definitely possible, I just don't have a test system
- There used to be tutorials on how to use Windows to

cross compile to ARM but they seem to have all gone away? One possible way to do this is install the Windows Subsystem for Linux (WSL) and you can then use the Linux directions after installing the cross-compiler and make

- If someone figures out a good or better way to do this on Windows let me know and I can update things
- Can you use your Pi to develop and not cross-compile at all? Yes if you have 2 SD cards to swap a USB SD-card reader. However it will be a lot card-swapping.

HW#2 Summary Again

- Make simple binary.
- Compile it with ARM toolchain (cross compile?)
- Replace kernel.img on your memory card.
- Boot into it!
- Easier said than done.
- What kind of setup do you have?

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 (ARM1176 also has THUMB but we won't be using that).
- The newer Pis have thumb2 which add their own complications

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               // load 1 into r1
lsl     r1,#24              // logic shift left by 24
str     r1,[r0,#GPIO_GPFSEL2]   // store to address in r0+offset
```

Why not `mov r1,#(1<<24)`?

Immediate constants generally have to be 4096 or less on ARM32 because of lack of bits to hold the constant in a 32-bit instruction.

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
```

How do we start moving to OS?

- Abstraction!
- Create “device drivers”

Abstracting Setting LED

- Instead of open-coding code to turn on / turn off GPIO47, can we make a generic higher level `turn_on_ACT_LED()` function that automatically does it without having to know which Pi type or GPIO address is involved?
- Even inside of that function can we have a nice `int set_gpio(int which);` function?

```
int set_gpio(int which) { // FIXME: add error checking
   int offset=which/32;
   gpio[GPSET0+offset]=(1<<(which%32));
}
```

Abstracting More

- Can do the same with the init code
- Also with delay code, write something using timers that can be better like `usleep()`
- Can be fun to work out how to do it yourself (you can for this assignment if you want) but the provided code for HW#3 will do this for you.

Raspberry Pi Background

- 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 8-bit 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 v1.1

- BCM2836 (v1.2 has BCM2837)
- quad-core ARMv7 Cortex A7
- SoC (mostly) the same, but much faster processor
- Board layout just like B+
- 1GB RAM (so all peripheral addresses move)
- Has THUMB2 support

Raspberry Pi Model 3

- BCM2837 – Quad-core 64-bit ARMv8 Cortex A53
- SoC still mostly the same
- Higher performance, but draws more power, overheats and can even crash if you run it too hard (fixed in 3B+)
- 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)

Raspberry Pi Model 3B+

- 1.4GHz
- Gigabit ethernet (though can only get up to 300MB)
- Power over ethernet
- 802.11ac wifi

Raspberry Pi Model 4B/400

- BCM2711 – Quad-core 64-bit ARMv8 Cortex A72
- Videocore VI / two micro HDMI connectors
- 1/2/4/8GB of LPDDR4
- Bluetooth 5, more advanced Wifi
- Actual gigabit ethernet (behind PCIe)
- Powered by USB-C
- two USB3.0 ports
- Extra i2c/SPI/etc available on GPIO header

Raspberry Pi Model 5

- BCM2712 – Quad-core 64-bit ARMv8 Cortex A76
- Videocore VII / two micro HDMI connectors
- 4/8GB of LPDDR4
- PCIe based
- Peripherals moved to separate custom chip
- Powered by USB-C (5V@5A, not all adapters support)
- Audio jack removed, composite moved to header
- Real time clock (no default battery)
- Power switch

BCM2835 SoC features

- Peripherals mmaped (aside on how that works in the face of caches)
- GPIO
- Primary pl011 UART
- bsc, aka i2c
- dma controller
- emmc (essentially hard-wired SD card)
- interrupts (mailbox, doorbell)
- pcm/i2s audio

- pwm (used for sound on Pi)
- spi
- spi/i2c slave
- system timer
- mini-uart (secondary serial port)
- arm timer
- usb
- video: HDMI?,composite