ECE 531 – Advanced Operating Systems
Lecture 10

Vince Weaver

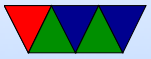
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

24 September 2025

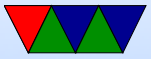
Announcements

- Don't forget HW#3
  - Serial ports can be tricky, don't put the assignment off
  - You'll be using the serial port for all the future homeworks
- HW#2 still being graded (UMaine network troubles last night delayed things)



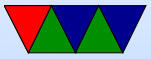
HW#2 Review – Code

- Still finishing grading this part
- A bit slow to grade with a lot of SD-card swapping



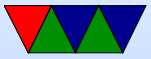
HW#2 Review – Filesize

- Size: C code 180 bytes, assembly 84 bytes
- Note if you use a global variable C code will bump up to 4k because the data segment gets aligned
- Linux kernel is 7megabytes or so



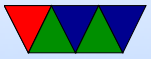
HW#2 Review – Why is C bigger?

- Can look at .dis files for disassembly
- Init: C has 60 bytes to set things up, assembly has none
- Delay: C 64 bytes due to pessimization from volatile (has to load/store load/store i over and over) asm 12 bytes
- C also saves/restores LR and registers to maintain calling convention.



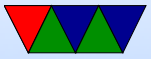
HW#2 Review – Other questions

- volatile – have C compiler not optimize away stores
- C array of 32-bit ints vs actually byte-wise access
- ALT4 on GPIO18 is SPI1_CEN_0
  - This is chip select for an SPI bus
  - Many, many people got this wrong, instead reporting the ALT0 or ALT5 values. Not sure why as it's a simple table lookup in the manual



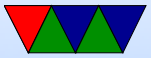
ARM has various Modes

- Modes:
- States
  - ISA: ARM (normal), Thumb, Jazelle, ThumbEE
  - Execution state (?)
  - Security: Secure and Non-secure
- Privilege Level
  - If secure: PL0 = user, PL1 = kernel
  - If non-secure: PL0 = user, PL1 = kernel, PL2 = hypervisor



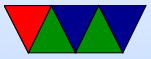
ARM Modes

User	PL0	
FIQ	PL1	fast interrupt
IRQ	PL1	interrupt
SVC	PL1	supervisor
MON	PL1	monitor (only if security extensions)
ABT	PL1	abort
UND	PL1	undefined instruction
SYS	PL1	system
HYP	PL2	hypervisor (only if virtual extensions)



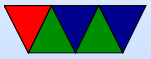
ARM Modes – continued

- User mode – unprivileged, restricted. Can only move to higher level by exception.
- System Mode – like USER, but no restrictions on memory/registers. Sort of like running as root, cannot enter by exception.
- Supervisor – kernel mode. SVC (syscall) instructions take you here. Also at reset (boot).
- Abort – called if a memory or prefetch causes an exception

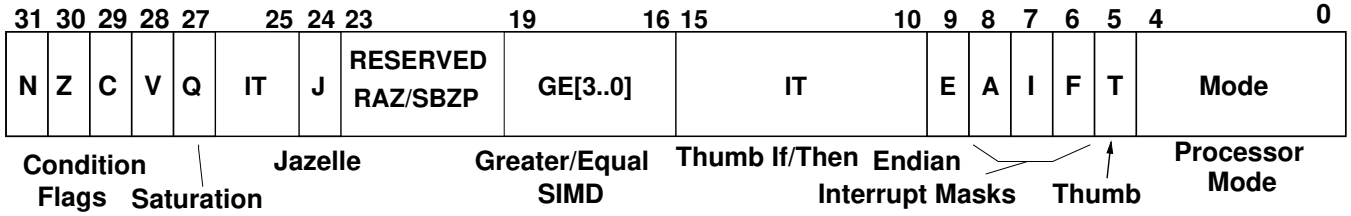


why is this useful? Virtual memory.

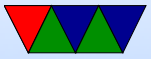
- Undefined – called when undefined instruction happens
  why is this useful? Emulator?
- FIQ/IRQ – fast or normal interrupt
- HYP – hypervisor, for virtualization. A bit beyond this class
  Due to change in firmware newer Pis boot into this mode
- Secure – secure mode, can lock things down.



ARM CPSR Register

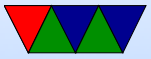


- Current Program Status Register
- Contains flags in addition to processor mode
- Six privileged modes
- One non-privileged: user (cannot write CPSR), now APSR?
- Interrupts and exceptions automatically switch modes



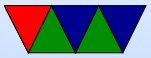
ARM Interrupt Registers

User/Sys	Hyp	Fast	IRQ	Supervisor	Undefined	Abort	Monitor
r0 r1 r2 r3 r4 r5 r6 r7							
r8 r9 r10 r11 r12		r8_fiq r9_fiq r10_fiq r11_fiq r12_fiq					
r13/sp r14/lr r15/pc	SP_hyp	SP_fiq LR_fiq	SP_irq LR_irq	SP_svc LR_svc	SP_und LR_und	SP_abt LR_abt	SP_mon LR_mon
apsr							
cpsr	spsr_hyp ELR_hyp	spsr_fiq	spsr_irq	spsr_svc	spsr_und	spsr_abt	spsr_mon



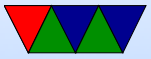
ARM Interrupt Handling

- Unlike other architectures, when switching modes the ARM hardware will preserve the status register, PC and stack and give you mode-specific versions (register bank switching).
- Also for Fast Interrupts r8-r12 are saved as well, allowing fast handlers that do not have to save registers to the stack.



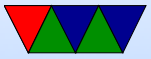
ARM Interrupt Handling

- ARM core saves CPSR to the proper SPSR
- ARM core saves PC to the banked LR (possibly with an offset)
- ARM core sets CPSR to exception mode (disables interrupts)
- ARM core jumps to appropriate offset in vector table

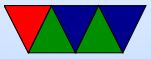


Vector Table

Type	Type	Offset	LR	Priority
Reset	SVC	0x0	–	1
Undefined Instruction	UND	0x04	lr-4/2	6
Software Interrupt	SVC	0x08	lr	6
Prefetch Abort	ABT	0x0c	lr-4	5
Data Abort	ABT	0x10	lr-8	2
UNUSED	–	0x14	–	–
IRQ	IRQ	0x18	lr-4	4
FIQ	FIQ	0x1c	lr-4	3

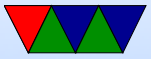


- See ARM ARM ARMv7 documentation for details.
- NOTE: contains a 4-byte instruction, not an address
- Location defaults to 0x000000
  if SCTL.V is 1 “high-vector” 0xffff0000
- If security mode implemented more complex, separate vectors for secure/nonsecure, and on nonsecure the SCTL.V lets you set it anywhere via VBAR
- Interrupts: IRQ = general purpose hardware,
  FIQ = fast interrupt for really fast response (only 1),
  SWI = syscalls, talk to OS
- FIQ mode auto-saves r8-r12.



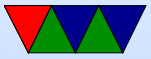
Complications

- What about thumb or endian mode when call into interrupt? Depends on flags in SCTLR register



Interrupt Stacks

- Stack pointer changes when handle interrupt (why?)
- Need to set that up in advance, before interrupts enabled
- Why does kernel have own stack pointer? Why not use the user stack? Does the user stack pointer always have to be valid?

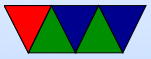


Ways to return from IRQ

- Regular function return not enough, need to change mode and adjust LR
- `subs pc,lr,#4`
  Sneakily branches and gets the right status register (special case when S in SUBS and PC is destination)
- `sub lr,lr,#4`

  `...`

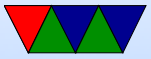
  `movs pc,lr` (or `rfe`)
- Another stores lr and other things to stack, then restores



```
sub lr,lr,#4
stmbd sp!,{r0-r12,lr}
...
ldmfd sp!,{r0-r12,pc}^
```

The caret means to load cpsr from spsr

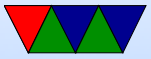
Exclamation point means to update sp after popping.



IRQ Handlers in C

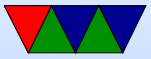
In gcc for ARM, you can specify the interrupt type with an attribute. Automatically restores to right address.

```
void function () __attribute__ ((interrupt ("IRQ")));

/* Can be IRQ, FIQ, SWI, ABORT and UNDEF */

void __attribute__((interrupt("UNDEF"))) undefined_instruction_vector(void) {

    while(1) {
        /* Do Nothing */
    }
}
```



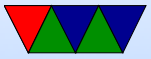
Getting Interrupt to Happen

- Initialize (set up vectors and stacks)
- Enable Interrupt at Pi Level
- Enable Interrupt at Device Level
- Enable Global interrupts at ARM Level

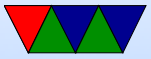


Raspberry Pi Interrupts

- See Section 7 of BCM2835 doc (though it's not well written)
- Up to 64 possible, but only subset available to ARM chip (rest belong to GPU)
- MMIO Registers used to configure:
  - Basic pending: 32-bit field with most common IRQ sources
  - Full pending: two 32-bit registers a bit for each IRQ source and whether triggered

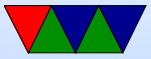


  - FIQ register: can pick which one is FIQ
  - Enable register: to set which interrupts are enabled
  - Disable register
- You also have to enable interrupts on the device too
- On Pi4 need to enable gic-400 interrupt controller too



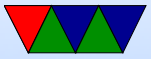
Initializing

- How do we get the vectors to address 0x0?
  Copy it there after the fact. Hard part is if we want the routines to be C code.
- Clever, have the reset vector point to start of code, so you can have the reset vector of beginning of code and it will jump to the right location.
- `ldr` does a PC-relative load, so as long as we copy the vectors at the same offset will work
- Leave at entry point, and first one is reset, so at boot

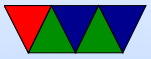


we jump to reset

```
_start:
    ldr pc, reset_addr
    ldr pc, undefined_addr
    ldr pc, software_interrupt_addr
    ldr pc, prefetch_abort_addr
    ldr pc, data_abort_addr
    ldr pc, unused_addr
    ldr pc, interrupt_addr
    ldr pc, fast_interrupt_addr
reset_addr:                 .word   reset
undefined_addr:             .word   undefined_instruction
software_interrupt_addr:    .word   software_interrupt
prefetch_abort_addr:        .word   prefetch_abort
data_abort_addr:            .word   data_abort
unused_addr:                .word   reset
interrupt_addr:             .word   interrupt
fast_interrupt_addr:        .word   fast_interrupt

_start:
    ...
reset:
```

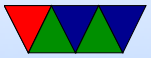


```
ldr r3, =_start
mov     r4, #0x0000
ldmia   r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia   r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
ldmia   r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia   r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
```

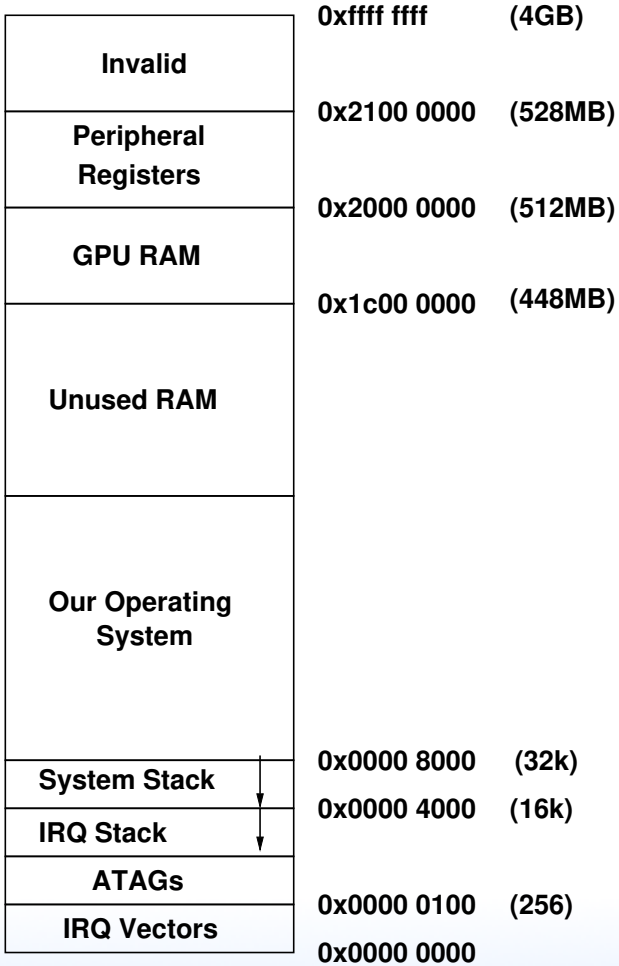


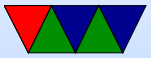
Setting up the Stacks

- Need chunk of memory for each stack
- Temporarily switch to mode, then set the stack pointer
- You can manually (without getting an interrupt) set the CPSR value with a `msr` instruction (move to status register)
- We start in SVC mode (Well, on pi2+newer HYP mode) but we can get to a mode where we can change CPSR



Pi1-B+ Memory Map





Code to Set up the Stacks

```
/* Set up the Interrupt Mode Stack      */
/* First switch to interrupt mode, then update stack pointer */
/* cpsr_c means just change the config (mode) part of the cpsr */
mov     r3, #(CPSR_MODE_IRQ | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISABLE )
msr     cpsr_c, r3
mov     sp, #0x4000

/* Switch back to supervisor mode */
mov     r3, #(CPSR_MODE_SVC | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISABLE )
msr     cpsr_c, r3
```