ECE 598 – Advanced Operating Systems
Lecture 8

Vince Weaver

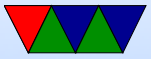
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

11 February 2016

Announcements

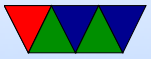
- Homework #3 Due.
- Homework #4 Posted Soon



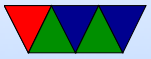
HW#3 Comments

- Sorry about the bug, the manual is a bit unclear about if you need to enable the FIFO even if you aren't using interrupts.

- Problems with serial ports? Why I said to start early. Did have one unexplained problem with screen on OSX. If you are getting corrupted chars or lost messages probably a serial setting. Bitter experience. Why some get through and others not? You can check the error bits in the FIFO but what do you do about it?

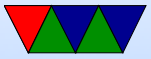


- Setting the speed dividers – there was a nice example in class / in the notes. Most people having problems were doing it the same odd way (be careful sharing code). Hard coding the final result is simplest, especially if you can't sanity check due to no early printk. If you're going to use floating point, promotion, and casting in C you better be sure you know how it works. Also typically you try to avoid using floating point in an operating system kernel (why?). True it might be nice to have some sort of generic set bps function, but it requires some division and what did we learn about non-constant division last

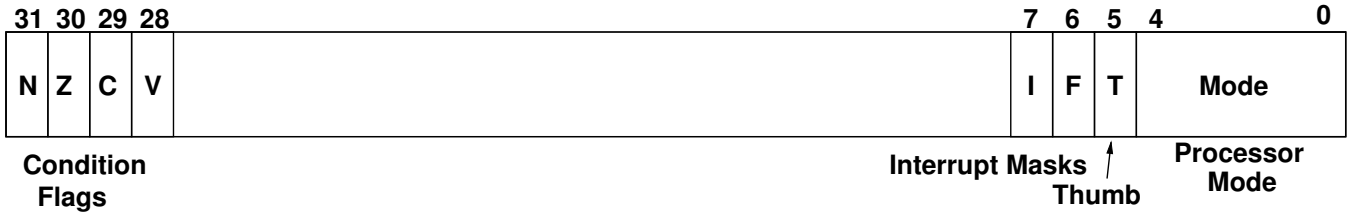


time?

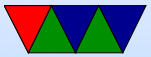
- Printing colors. Didn't want to make it *too* easy. How do you print escape char? Various ways. Implement %c. Manually set the offset in your string to 27. Or in C you can print \033 which is octal for ASCII 27



(Review) ARM CPSR Register

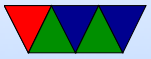


- Current Program Status Register



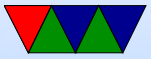
(Review) ARM Interrupt Handling

- 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



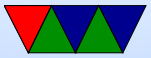
(Review) Vector Table

Type	Type	Offset	LR	Priority
Reset	SVC	0x0	–	1
Undefined Instruction	UND	0x04	lr	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



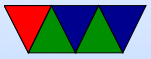
Getting Interrupt to Happen

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

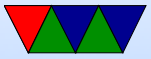


Raspberry Pi Interrupts

- See Section 7
- Up to 64 possible, but only subset available to ARM chip (rest belong to GPU)
- Basic pending register: 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 registers, to set which interrupts are enabled
- Disable registers
- You also have to enable interrupts on the device too

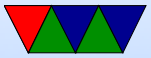


Initializing

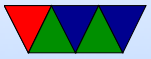
- How do we get the vector to 0x0?
  Copy it there after the fact. Hard part is if we want the routines to be C code.

- 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.

```
_start:
    ldr pc, reset_addr
    ldr pc, undefined_addr
    ldr pc, software_interrupt_addr
    ldr pc, prefetch_abort_addr
```

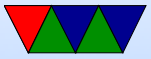


```
    ldr pc, data_abort_addr
    ldr pc, unused_addr
    ldr pc, interrupt_addr
    ldr pc, fast_interrupt_addr
reset_addr:             .word   reset
undefined_addr:         .word   undefined_instruction
software_interrupt_addr:    .word   software_interrupt
prefetch_abort_addr:        .word   prefetch_abort
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}
```

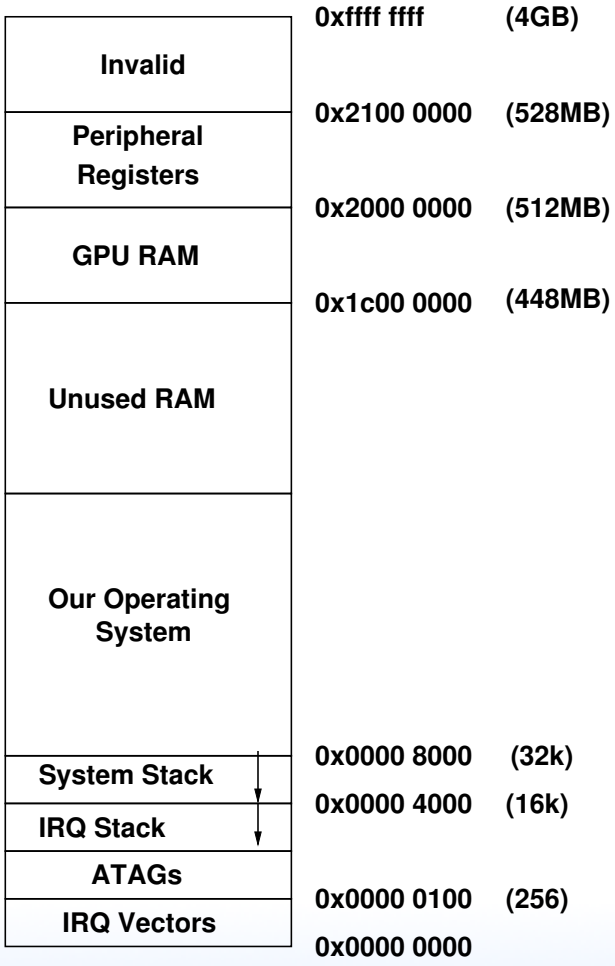


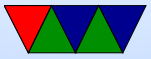
Setting up the Stacks

- Need chunk of memory for each stack
- Need to 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)
- Luckily system boots up in SVC mode so we can change CPSR



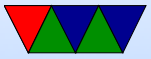
Our Memory Map





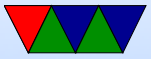
Setting up the Stacks

```
        /* Set up the Interrupt Mode Stack      */
        /* First switch to interrupt mode, then update stack pointer */
        mov     r3, #(CPSR_MODE_IRQ | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISA
BLE )
        msr     cpsr_c, r3
        mov     sp, #0x4000

        /* Switch back to supervisor mode */
        mov     r3, #(CPSR_MODE_SVR | CPSR_MODE_IRQ_DISABLE | CPSR_MODE_FIQ_DISA
BLE )
        msr     cpsr_c, r3
```



Clearing the Interrupt Status Bit

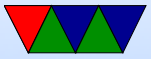
```
_enable_interrupts:
    mrs     r0, cpsr
    bic     r0, r0, #0x80 ; bit clear
    msr     cpsr_c, r0

    mov     pc, lr
```



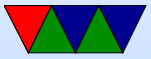
Configuring a Timer

- Section 14 of peripheral manual.

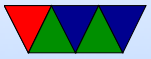
- It is similar but not exactly the same as an ARM SP804 Timer
- There are also the system timers (4 timers described in Section 12).
- Note that the timer we use is based on the APB clock which



```
 /* Timer is based on the APB bus clock which is 250MHz on Rasp-Pi */

int timer_init(void) {

        uint32_t old;

        /* Disable the clock before changing config */
        old=mmio_read(TIMER_CONTROL);
        old&=~(TIMER_CONTROL_ENABLE|TIMER_CONTROL_INT_ENABLE);

        /* First we scale this down to 1MHz using the pre-divider */
        /* We want to /250.  The pre-divider adds one, so 249 = 0xf9 */
        mmio_write(TIMER_PREDIVIDER,0xf9);

        /* We enable the /256 prescalar */
        /* So final frequency = 1MHz/256/61 = 64.04 Hz */

        mmio_write(TIMER_LOAD,61);

        /* Enable the timer in 32-bit mode, enable interrupts */
        /* And pre-scale the clock down by 256 */
        mmio_write(TIMER_CONTROL,
                TIMER_CONTROL_32BIT | /* typo 23 */
```

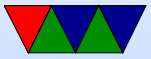


```
                TIMER_CONTROL_ENABLE |
                TIMER_CONTROL_INT_ENABLE |
                TIMER_CONTROL_PRESCALE_256);

        /* Enable timer interrupt */
        mmio_write(IRQ_ENABLE_BASIC_IRQ,IRQ_ENABLE_BASIC_IRQ_ARM_TIMER);

        return 0;

}
```

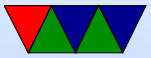


Sample Interrupt Handler

- CPU disabled interrupts and switches CPSR to correct mode
- Save registers (no need to save SPSR unless nested)
- Interrupt handler checks and sees which interrupt was triggered (in a register)
- Interrupt Status Routine (ISR) called which services the routine and then acknowledges interrupt

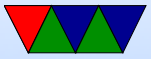


- Handler restores context, returns
- CPU restores execution



Sample Interrupt Handler

```
void __attribute__((interrupt("IRQ"))) interrupt_vector(void) {
    static int lit = 0;
    int which;

    /* Check to see what interrupt we had */
    which=mmio_read(IRQ_BASIC_PENDING);
    if (which&0x1) {

        /* Clear the ARM Timer interrupt */
        mmio_write(TIMER_IRQ_CLEAR,0x1);

        /* Flip the LED */
        if( lit ) { led_off(); lit=0; }
        else {led_on(); lit=1; }
    }
}
```



Enabling Interrupts

```
static inline uint32_t get_CPSR(void) {
        uint32_t temp;
        asm volatile ("mrs %0,CPSR":"=r" (temp):) ;
        return temp;
}

static inline void set_CPSR(uint32_t new_cpsr) {
        asm volatile ("msr CPSR_cxsf,%0"::"r"(new_cpsr) );
}

/* enable interrupts  */
static inline void enable_interrupts(void){
        uint32_t temp;
        temp = get_CPSR();
        set_CPSR(temp & ~0x80);
}
```