ECE 598 – Advanced Operating Systems
Lecture 8

Vince Weaver

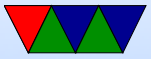
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

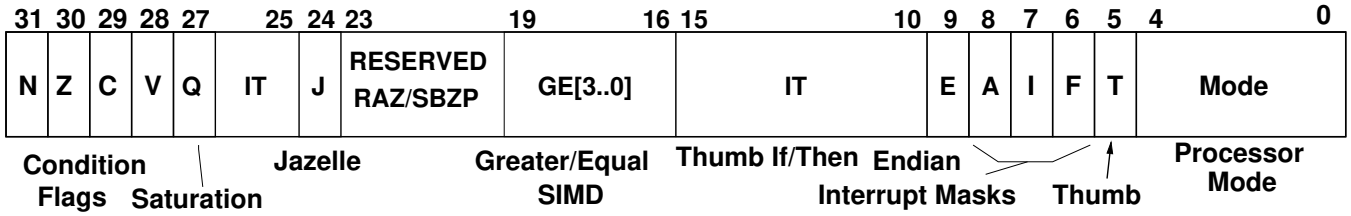
15 February 2018

Announcements

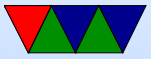
- Homework #3 Due.
- Homework #4 Posted Soon



(Review) ARM CPSR Register

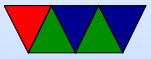


- Current Program Status Register



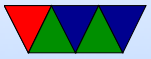
(Review) ARM Interrupt Handling

- 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



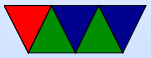
(Review) Vector Table

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



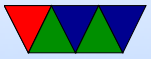
Getting Interrupt to Happen

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

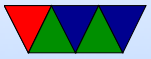


Raspberry Pi Interrupts

- See Section 7 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

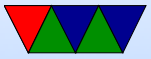


  - FIQ register: can pick which one is FIQ
  - Enable register: to set which interrupts are enabled
  - Disable register
- You also have to enable interrupts on the device too



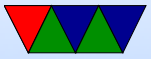
Initializing

- 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

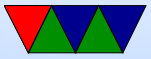


we jump to reset

```
_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:
```

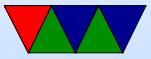


```
ldr r3, =_start
mov     r4, #0x0000
ldmia   r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia   r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
ldmia   r3!,{r5, r6, r7, r8, r9, r10, r11, r12}
stmia   r4!,{r5, r6, r7, r8, r9, r10, r11, r12}
```

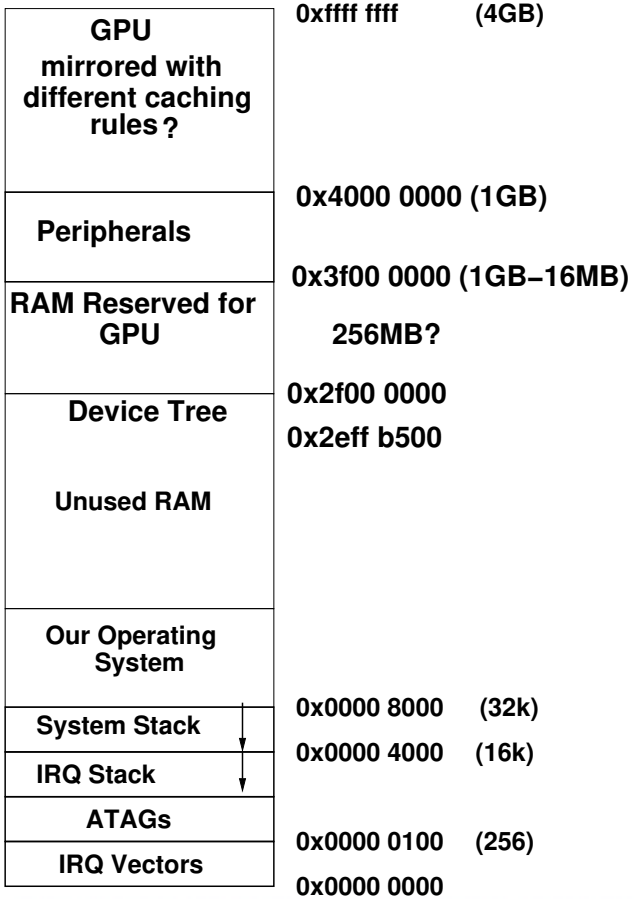


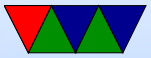
Setting up the Stacks

- 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/pi3 HYP mode) but we can get to a mode where we can change CPSR



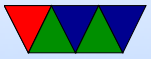
Pi2/Pi3 Memory Map





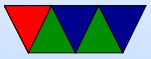
Setting up the Stacks

```
/* Set up the Interrupt Mode Stack      */
/* First switch to interrupt mode, then update stack pointer */
/* cpsr_c means just change the config (mode) registers */
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
```



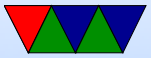
Clearing the Interrupt Status Bit

```
/* or use the "cpsid i" instruction? */

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

    mov     pc, lr
```

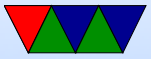


Configuring a Timer

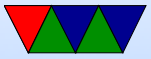
- Section 14 of BCM2835 Peripheral manual.
- Similar, but not exactly the same, as an ARM SP804
- There are also the system timers (4 timers described in Section 12).
- Note that the timer we use is based on the APB clock which ticks at 250MHz
- Limitations: it scales with the system clock, so frequency might change
- Important registers



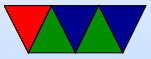
- TIMER_LOAD: set a value and it will count down on each tick and give interrupt when zero. Automatically re-loaded after interrupt.
- TIMER_CONTROL: start/stop, interrupts on/off, scaling
- TIMER_RELOAD: queue a different value to be loaded into TIMER_LOAD automatically when current hits zero
- TIMER_IRQ_CLEAR: clears the interrupt
- TIMER_PRE_DIVIDE: another divider, as original design was for 1MHz clock



```
 /* 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);
        mmio_write(TIMER_CONTROL,old);

        /* 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 */

    /* The value is loaded into TIMER_LOAD and then it counts down */
    /* and interrupts once it hits zero. */
    /* Then this value is automatically reloaded and restarted */

        mmio_write(TIMER_LOAD,61);
```

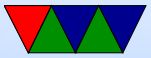


```
        /* Enable the timer in 32-bit mode, enable interrupts */
        /* And pre-scale the clock down by 256 */
        mmio_write(TIMER_CONTROL,
                TIMER_CONTROL_32BIT | /* typo 23 */
                TIMER_CONTROL_ENABLE |
                TIMER_CONTROL_INT_ENABLE |
                TIMER_CONTROL_PRESCALE_256);

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

        return 0;
}
```



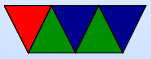
Sample Interrupt Handler

- CPU disables interrupts, 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

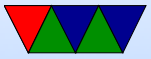
```
void __attribute__((interrupt("IRQ"))) interrupt_vector(void) {
    static int lit = 0;
    int which;

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

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

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



IRQ Handlers in C

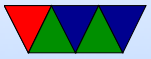
In gcc for ARM, you can specify the interrupt type with an attribute. Automatically restores to right address.

```
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 */
        }
}
```



Enabling Interrupts in C

```
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);
}
```



Writing a Command Parser

- Read in values one character at a time intro string buffer.
- Read until Enter (slash r)
- Be sure to NUL terminate! Also be sure to not overflow buffer!
- How to parse? strtok()? stcmp()?
  Who provides these string routines?
- Simple way to do things is to manually check, like
  `if ((buffer[0]=='l') && (buffer[1]=='s')) somet`