EEE3096S: Embedded Systems II

LECTURE 21:

INTERRUPTS AND EXCEPTIONS ON ARM

Presented by:

STANLEY MBEWE

Electrical Engineering
University of Cape Town

OVERVIEW

- Interrupts on the ARM (the IRQ and FIQ)*
- Interrupt service routines
- ARM Exceptions
- ARM Advanced Interrupt Handlers and marshalling many interrupts to IRQ and FIQ

Connection to Text Book*
The Textbook introduces the concept of Exceptions and Interrupts on page 207, in terms to why these are operations that a processors provides.
This slides provide a brief reminder of the concept of polling, interrupts, and interrupt service routines.

*Recommended textbook for this course is "Computer Organization and Design - ARM Edition" by A. Patterson, John L. Hennessy

ARM HAS 2X MODES FOR INTERRUPT HANDLING

- In previous lectures, it was shown that in ARM there is a:
  - IRQ (Interrupt Request) mode and a
  - FIQ (Fast Interrupt) mode
- These are used for handling interrupts:
  - If the IRQ input to the ARM is raised, it changes to IRQ mode (if IRQ enabled)
  - If the FIQ input to ARM is raised, it changes to FIQ mode (if FIQ enabled)
- The ARM modes are recapped on the next slide

HAL GENERATED CODE IRQ ROUTINE

```
void EXTI0_1_IRQHandler(void)
{
   // you need to add code here to handle the IRQ …

  HAL_GPIO_EXTI_IRQHandler(B1_Pin); // Clear interrupt flags
}
```

In the pracs, one of the things you need to think about is debouncing, which is essentially having a timestamp at the first interrupt and ignoring any interrupts that happen within a certain period after the first interrupt.

Why don't we just do a delay, like delay_ms(100) in the interrupt handler?? Because that cause the processing to halt for a while, stopping all operations until the delay is complete… short answer is doing a hard delay would be rather bad programming practice.

HAL GENERATED CODE IRQ ROUTINE

The HAL also generates some assembly code in startup_stm32f051r8tx.s, if you look for EXTI0_1_IRQHandler you'll notice it appears in the g_pfnVectors segment which sets up various exception handlers:

```
/*******************************************************************************
* The minimal vector table for a Cortex M0.  Note that the proper constructs
* must be placed on this to ensure that it ends up at physical address
* 0x0000.0000.
*******************************************************************************/
  .section .isr_vector,"a",%progbits
  .type g_pfnVectors, %object
  .size g_pfnVectors, .-g_pfnVectors

g_pfnVectors:
  .word  _estack
  .word  Reset_Handler
...              // various unused exception handlers, just assigned NULL
  .word  0
  .word  0
...
  .word  RCC_CRS_IRQHandler              /* RCC and CRS
  .word  EXTI0_1_IRQHandler             /* EXTI Line 0 and 1
  .word  EXTI2_3_IRQHandler             /* EXTI Line 2 and 3
...
```

i.e. it gets set up on memory when the program is loaded, there is no call. In the main.c, there is no explicit placement of a pointer to function EXTI0_1_IRQHandler into the exception handler memory location.

What's happening… why does the interrupt work?!

First a reminder on the interrupts and exception handling on the ARM…

7 MODES OF ARM OPERATION & INTERRUPTS

1. User Mode (applications run in this mode)
2. Interrupt ReQuest (**IRQ**) – mode switched to when low priority interrupt is signalled
3. Fast Interrupt reQuest (**FIQ**) – Used when a high priority interrupt is signalled
3. Supervisor – on software interrupt / reset
4. Abort – Memory exceptions
6. Undef – Handles undefined instructions
7. System Mode - Privileged mode (for OS). Similar to user mode with fewer restrictions.

The IRQ and FIQ modes are used for handling interrupts.

SETTING UP THE IRQ / FIQ

- In order to handle an IRQ or FIQ, the ARM jump tables, and status register (CPSR) must be set up appropriately.
- But usually*, the peripheral interrupt controller coordinating interrupts also needs to be configured (explained soon).
- The jump table should be configured first before the program status register is set.
  (hence why its in the assembly start code module, and therefore done during start-up while loading the program)

* This is for many of the more fancy ARM cores, such as the ARM Cortex

INTERRUPT SERVICE ROUTINES (ISR) ON ARM

- Setting up interrupts, usually during startup:
    1. Set up ARM exception vector
    2. Configure the interrupt controller
    3. Set status flags as needed
    4. On ARM processor: clear IRQ and FIQ bits of CPSR

ARM EXCEPTION VECTOR

- Just 32 bytes in size
- Starts at address 0x00000000 (can be set to 0xFFFF0000 for Windows CE)
- *NB:* Each entry point to the vector contains an ARM 32bit instruction

FIQ	0x1C
IRQ	0x18
Reserved	0x14
Data Abort	0x10
Prefetch Abort	0x0C
Software Interrupt	0x08
Undefined Instruction	0x04
Reset	0x00

NOTE: each 32-bit element in this vector is an **instruction**, it is not an address that is to be jumped to.
Often, jump addresses are put in after address 0x20, and each instruction in the vector is just: `LDR pc, (pc+0x20)`

THE ISR FUNCTION ON ARM

- On function entry:
  - Disable Interrupts (unless re-entry required)
  - Adjust return address (Link Register(LR) -= 4 on ARM9)
  - Save context (partly automatic)
- In the ISR body:
  - Handle the interrupt (often is just a call to a C function)
  - Reset external interrupt device / peripheral (if necessary)
- When the ISR is ready to exit:
  - Restore context (which was saved on entry)
  - Restore the program counter

NESTED INTERRUPTS

- Most CPUs (including ARMs) support priority interrupts
- Interrupt nesting refers to the ability of a high-priority interrupt to preempt a lower priority interrupt
- Simple systems seldom use interrupt nesting; this is done by the ISR simply disables all interrupts until it returns

REENTRANT INTERRUPTS

- A reentrant function is one that can be called asynchronously from multiple threads without concern for synchronization or mutual access.
- A reentrant ISR is one that can be suspended (e.g. by a higher priority interrupt) and later resumed.

REENTRANT INTERRUPTS

Three rules are used to determine whether a function really is reentrant:

1. A reentrant function cannot use variables in a non-atomic way (i.e. use local variables only)
2. A reentrant function cannot call any non reentrant functions
3. A reentrant function cannot use the hardware in a non-atomic* way

* Atomic operations are operations visible to all parties and are uninterruptible

EXCEPTIONS AND EXCEPTION HANDLERS

- Lecture 8 discussed how the first few memory addresses of ARM, 0x00 – 0x1C, are exception vectors. Note: FIQ and IRQ are also exceptions on ARM.
- An exception can be considered a trigger to the CPU, e.g. happening when a peripheral raises an interrupt line, or when an arithmetic error (e.g. divide by zero) occurs.
- When the ARM experiences an exception (if it is in a mode to handle exceptions), it does the following: 1) suspend current operation, 2) loads the 32-bit value from the appropriate address in the exception table (e.g. if IRQ is raised it reads the value from address 0x18), and 3) executes (i.e. feeds into the instruction processor) the loaded value. Usually, instructions stored in the exception vector are branch instructions.

(see Lecture 8 slides)

Introducing the AIC…

The ARM processor itself has only two interrupt lines, FIQ and IRQ that you now know about. But that is far too few for sophisticated embedded system. This is where the Advanced Interrupt Controller (the AIC) comes in to play. The AIC is essentially a peripheral itself that is on the chip, connected with the ARM processor.
The AIC provides 32 interrupt connections that can be used to hook up peripherals, either internal or external to the ARM processor itself.

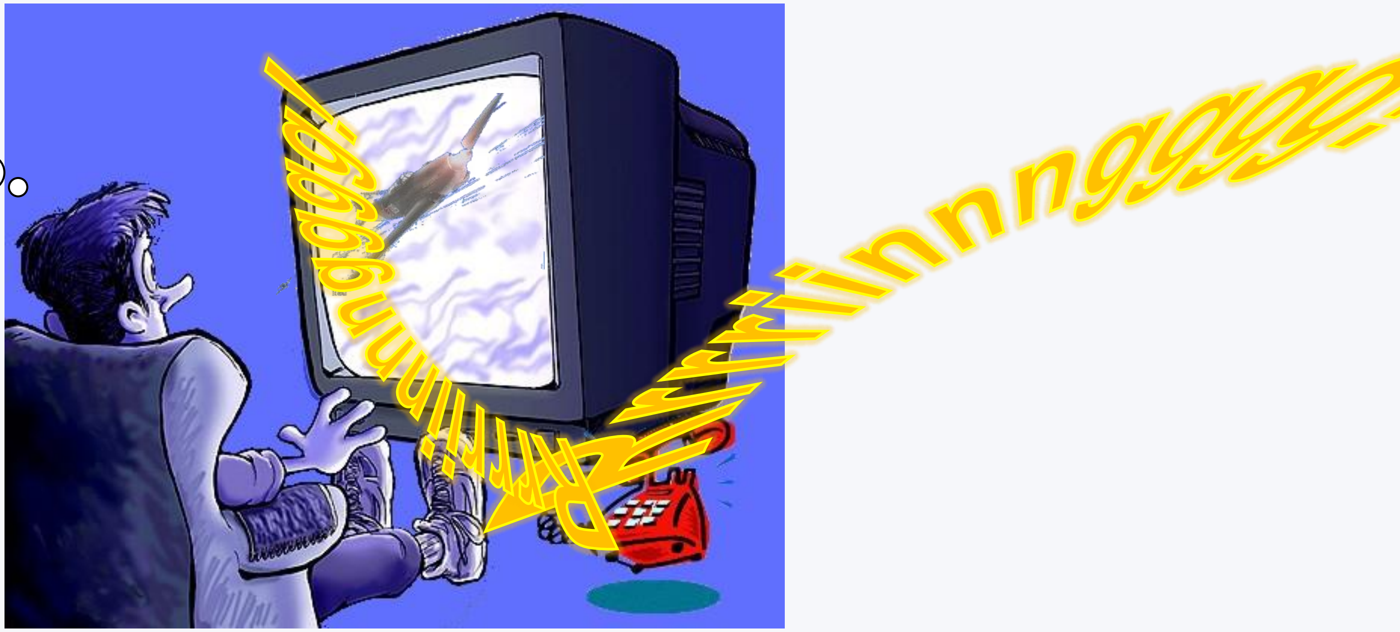
The AIC connects directly (indeed takes over sending signals) to the ARM's two interrupt lines (the IRQ and FIQ).

Use of the AIC can become quite complex, these slides aim to provide an overview of what is involved and what benefits the AIC provides, it is not necessary to go deeply into these aspects for this course.

The BCM2835 Interrupt Controller (i.e. the Pi's SoC) is explained in Section 7 (pp. 109-118) of Broadcom's BCM2835 ARM Peripherals datasheet.

INTERRUPT MARSHALLING WITH THE ADVANCED INTERRUPT CONTROLLER (AIC)

I just hate it when my favorite program is interrupted.



ARM AIC FOR MARSHALLING ADDITION IRQS

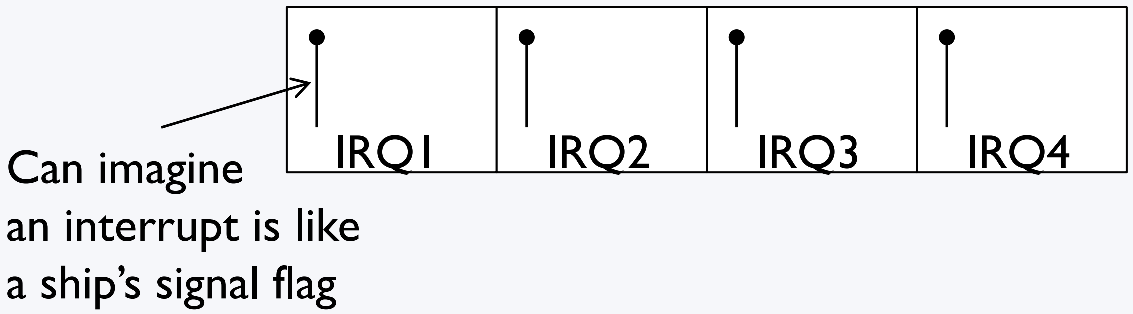
- Features
  - 8-level priority, individually maskable, vectored
  - Up to thirty-two interrupt sources.
  - Reduces software and real-time overhead in handling internal & external interrupts.
- AIC drives both the nFIQ (fast interrupt request) and nIRQ (standard interrupt request) inputs of the ARM processor.
- Inputs of AIC either
  - Internal peripheral interrupts or external interrupts from pins.

THE AIC

- The 8-level Priority Controller allows
  - Different priority levels for interrupt source
  - Higher priority interrupts to be serviced while lower priority interrupt in process
- Internal interrupt triggering:
  - Level sensitive or edge triggered
- External interrupts triggering:
  - Positive-edge / negative-edge or
  - High-level / low-level sensitive

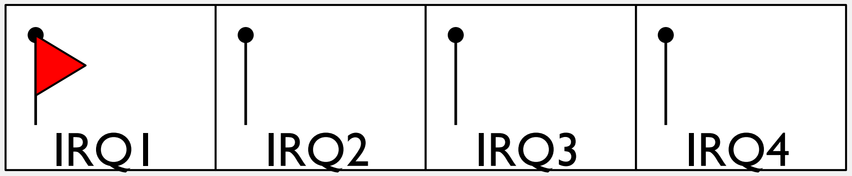
MASKING INTERRUPTS

- An interrupt mask is simply a means to tell the CPU to ignore, or not respond to, certain interrupts
- Works the same as a bitmap mask



MASKING INTERRUPTS

CPU in user mode

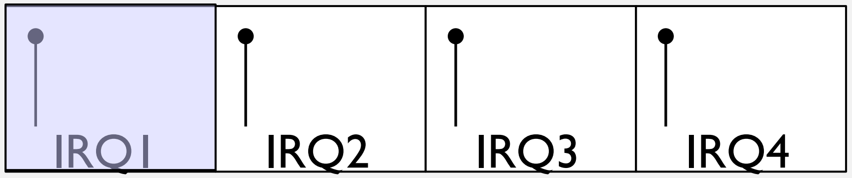


IRQ1 occurs!

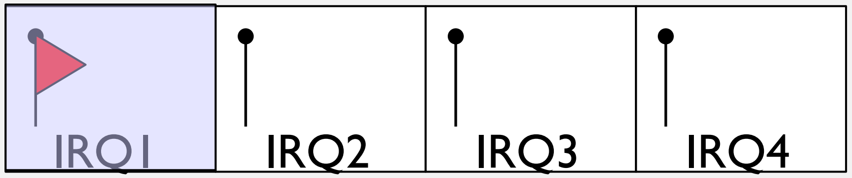
IRQ1 interrupt service routine (ISR)

CPU in IRQ mode

Starts by **masking** out IRQ1 and acknowledging/clearing IRQ1



Inside the ISR, another IRQ1 might occur…



The IRQ1 will only be serviced once the IRQ1 mask is removed

AIC VECTORING

- “Interrupt vectoring” →
  - Each type of interrupt (i.e. one of the 32 interrupts) can have a different service routine address
- A standard ARM without AIC integration has only 2 interrupts – IRQ and FIQ (with separate addresses)
- ARM with AIC has in vector table a read instruction that reads address to jump to from an AIC peripheral register

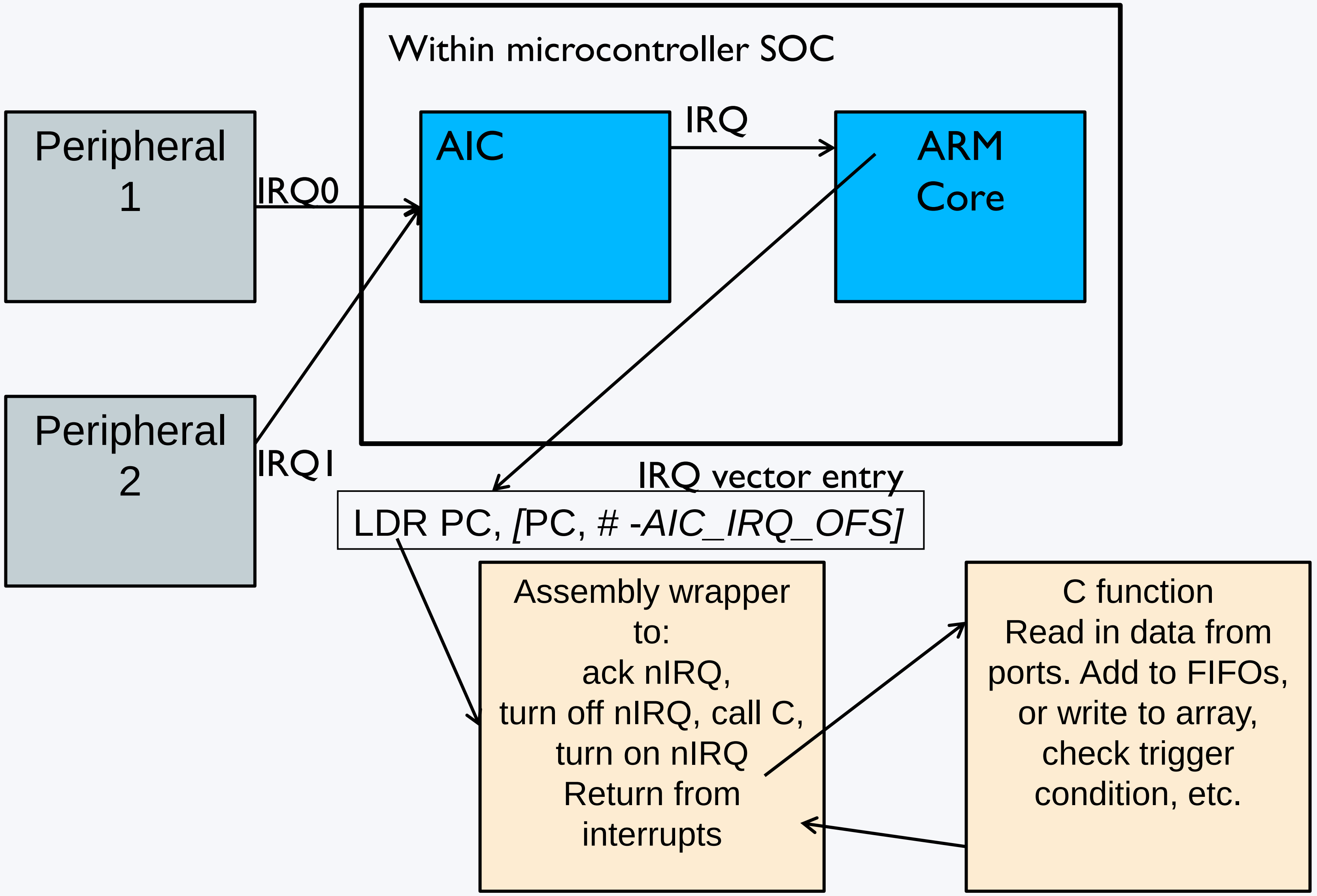
AIC VECTORING

Exception Vector	Address	Contents
FIQ	0x1C	LDR PC, *[PC, # -AIC_FIQ_OFS]*
IRQ	0x18	LDR PC, *[PC, # -AIC_IRQ_OFS]*
Reserved	0x14	Reserved
Data Abort	0x10	…
Prefetch Abort	0x0C	…
Software Interrupt	0x08	…
Undefined Instruction	0x04	…
Reset	0x00	…

The ARM vector table (starting at 0) is configured so that the FIQ has an instruction that reads the FIQ hander from an AIC register. Similarly, the IRQ location contains an instruction to read the IRQ address from an AIC register.

The instruction “LDR PC, [PC, # -AIC_FIQ_OFS]” is loading a word at a 16-bit offset from the PC into the PC. The AIC addresses are memory-mapped input ports, in the upper most memory location (0xFFFF8000 - 0xFFFFFFFF)

AIC INTERRUPT OPERATION


Within microcontroller SOC
Peripheral 1
IRQ0
AIC
IRQ
ARM Core
Peripheral 2
IRQ1
IRQ vector entry
LDR PC, *[PC, # -AIC_IRQ_OFS]*
Assembly wrapper to: ack nIRQ, turn off nIRQ, call C, turn on nIRQ Return from interrupts
C function Read in data from ports. Add to FIFOs, or write to array, check trigger condition, etc.