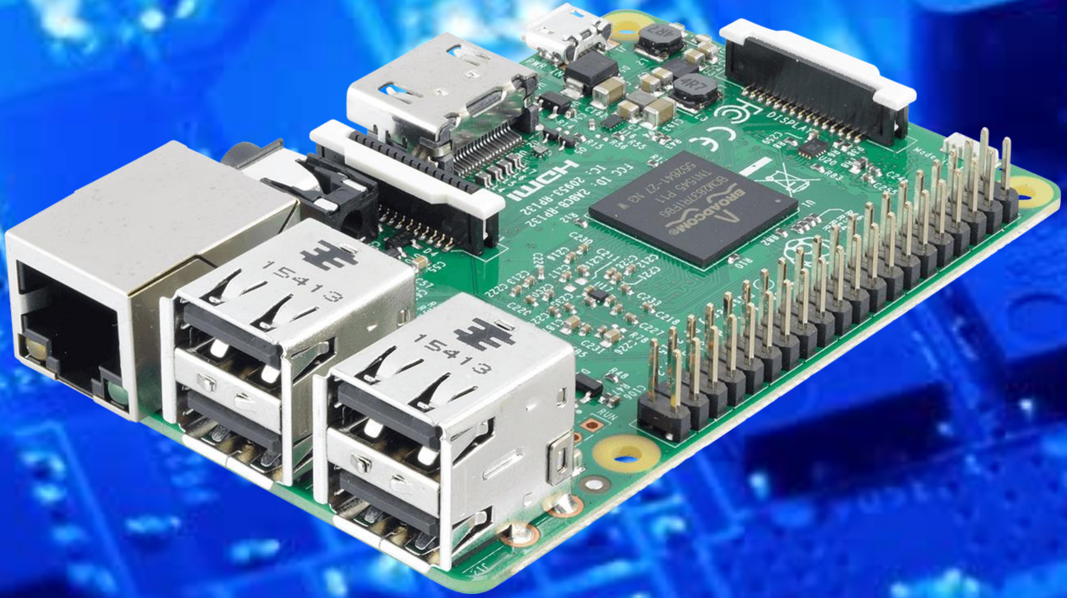
EEE3096S



Advanced Interrupts

L19

Embedded Systems II

Dr Simon Winberg

Electrical Engineering
University of Cape Town

Outline of Lecture

- When to use interrupts
- Execution speed and code density
- ARM Interrupts
- Specialized Interrupt Behaviours

When to Use Interrupts

- Interrupt generation can be enabled or disabled on many peripherals
- Interrupts add complexity to code
- Interrupts can improve performance compared to polling – i.e. resources (like CPU cycles and reading IO pins) are not wasted
- But there are compromises:
  - Usually a CPU has a limited number of interrupt lines. The ARM has essentially 2 only FIQ and IRQ …

Code Density Execution Speed and Interrupts

Recommended additional reading to better understand intricacies of interrupts: L19-Interrupts Reading.pdf in folder “Supplementary Reading” on Vula course webiste.

Speed and Code Density

- For RISC computers (ARM is one), execution speed is closely linked to code density

  Where 'code density' =

  The amount of space that an executable program takes up in memory

- Compilers are getting very good at optimizing this – no need to use special techniques
- Usually don't need to worry about speed except for your inner loops
- Measure where your performance is lost – it's often not where you think

Further optional reading

If you're interested in further reading on this see: Richard Phelan (2003) "Improving ARM Code Density and Performance: New Thumb Extensions to the ARM Architecture" Available: http://www.arm.com/pdfs/Thumb-2CoreTechnologyWhitepaper-Final4.pdf

Relevance of Code Density to Interrupts

- Code Density may be a factor for interrupt service routines (ISRs)
- Typically you want ISRs to take as little space, so you may want to

  Keep ISRs / frequently used functions and memory in fast memory

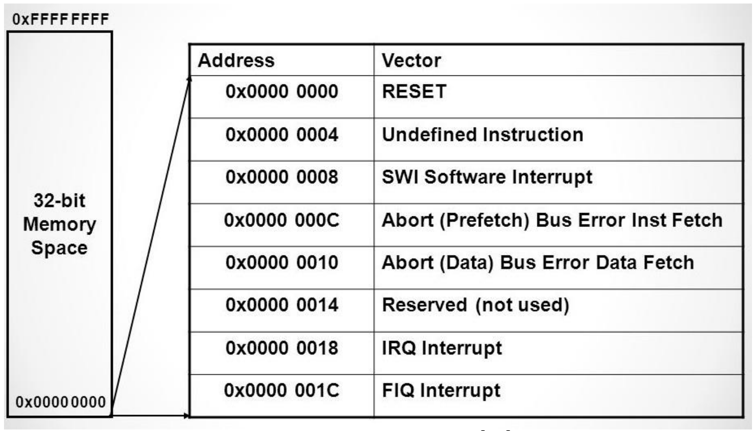
    - e.g. a computer may have a small amount of SRAM for frequently used functions and slower DRAM for less frequent code
    - CPU may support "No Evict Mode" or "Cache-as-RAM“ cache operation where basically parts of the cache can be locked or just used as RAM instead of cache
- So you generally *want the ISR to have a high code density* if possible – and this may involve choosing instructions wisely:
  – ARM Thumb instructions are register-only half-word instructions (two instructions per address)
  – Or ARM ‘multiop’ e.g. r0=r1[r2]; r2=r2+1 in one go

Interrupts on the RPi

- There are
  - Interrupt Service Routines (ISR)
  - Interrupt functions
  - Threaded callback handlers
- These are similar but not necessarily the same. Let’s look at these concepts quickly…

Interrupt Service Routines (ISR) on ARM

- An ISR is usually a function that is separate from an application (for one run by an OS) and runs in supervisor mode.
- It is usually linked to from the Interrupt Vector Table (IVT)
  - But on the ARM it is a more complicated setup, peripheral interrupt handling is usually a two-jump process (see AIC later!)



Address	Vector
0x0000 0000	RESET
0x0000 0004	Undefined Instruction
0x0000 0008	SWI Software Interrupt
0x0000 000C	Abort (Prefetch) Bus Error Inst Fetch
0x0000 0010	Abort (Data) Bus Error Data Fetch
0x0000 0014	Reserved (not used)
0x0000 0018	IRQ Interrupt
0x0000 001C	FIQ Interrupt

Arm vector table

Threaded Callback Handler

- A threaded callback is essentially:
  - One (IO master) thread is in charge of IO handling
  - Callback functions are implemented to do stuff under certain IO operations (e.g. print “PB hit” when pushbutton pressed)
  - This IO master needs to be passed callback functions
- Important issue is:
  - There are at least two threads involved: the application thread(s) and the IO thread
  - Communication between these threads are needed so that the user application can tell the IO master to add or remove IO callback handlers.

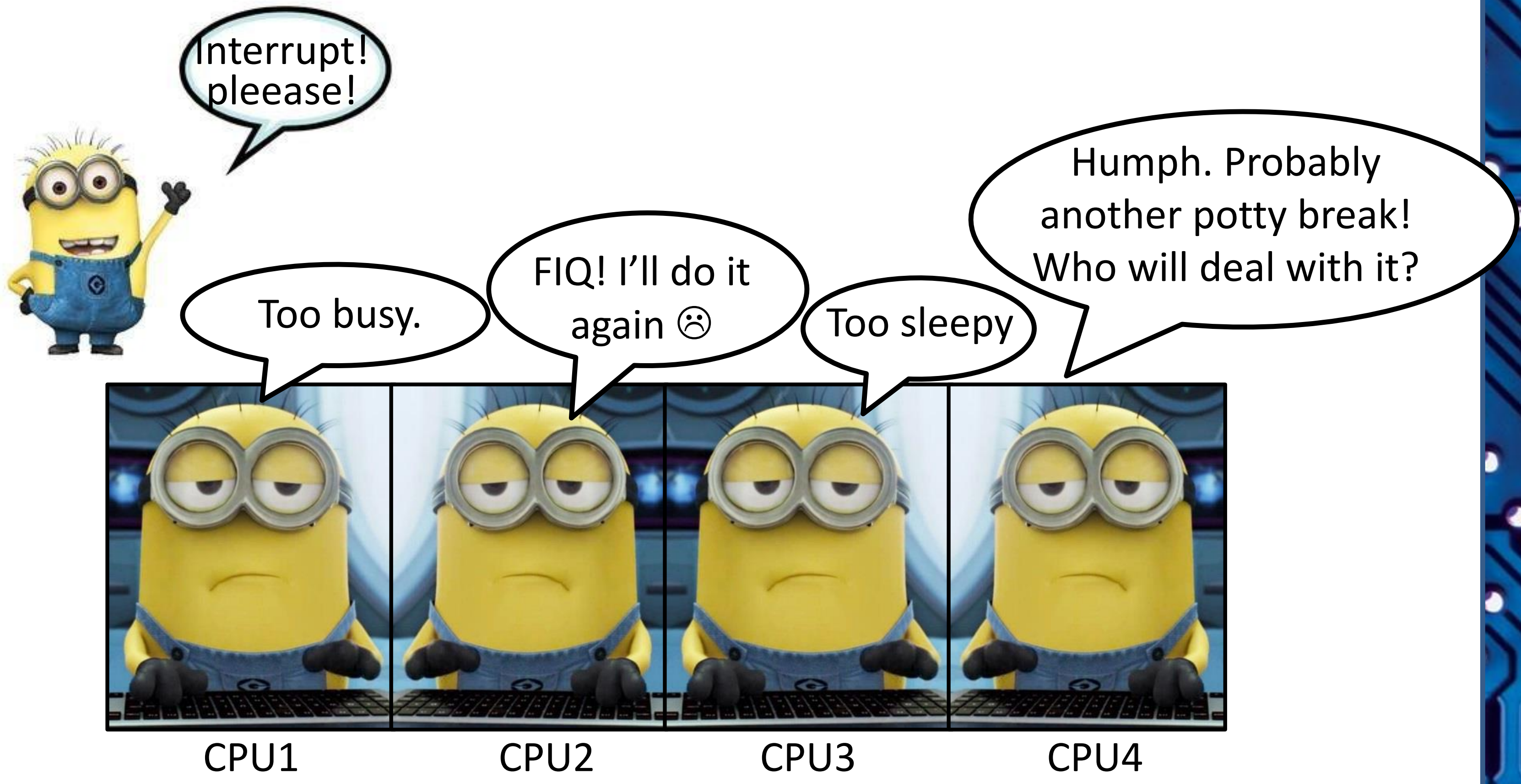
Concept of how the threaded callback handler works

Interrupt Function

- This is just a very general term, it might refer to any kind of interrupt handing ☺

ARM Interrupts

More advanced but necessary knowledge for knowing ARM-based processors/SoC design approach



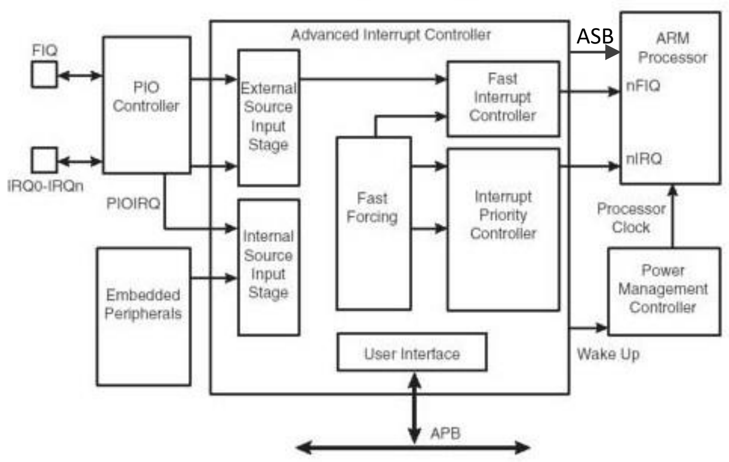
ARM Interrupts

- The ARM CPU has two interrupt lines
  - IRQ = normal interrupt request line
  - FIQ = fast interrupt request line
- Operation:
  - IRQ is essentially normal priority
  - FIQ is high priority – design usually supress other interrupts while FIQ is busy, so your FIQ ISR needs to be fast!!
- BUT: In any real system there will be many more sources of interrupts, not just two… so how does one handle this?? Here's how…

ARM – having more than 2 Interrupts

- Usually an ARM-based processor, i.e. which contains the ARM CPU IP as the core, has other hardware facilities included (often called "the ARM ecosystem")…
- One such element of the ARM ecosystem (which is usually external to the CPU but internal to the chip) is mechanisms for additional interrupt
- The ATMEL "Advanced Interrupt Controller (AIC)" is one such example (shown on right, for which different versions are available)
- As shown, a basic AIC allows for 32 interrupts, that can be linked to either FIQ or IRQ. Also allows masking (turning on/off sensitivity), prioritization etc.

Advanced Interrupt Controller



FIQ
IRQ0-IRQn
PIO Controller
PIOIRQ
Embedded Peripherals
Advanced Interrupt Controller
External Source Input Stage
Internal Source Input Stage
Fast Forcing
Fast Interrupt Controller
Interrupt Priority Controller
User Interface
APB
ASB
ARM Processor
nFIQ
nIRQ
Processor Clock
Power Management Controller
Wake Up

How it works:

Setup:

1. There are 32 interrupts, linked to on-chip or external peripheral
2. Interrupt lines to use much be linked to a 'jump address', a priority, and IRQ or FIQ channel

Operation:

1. One of 32 interrupts happen
2. AIC raises IRQ or FIQ
3. AIC sends 'jump address' to CPU

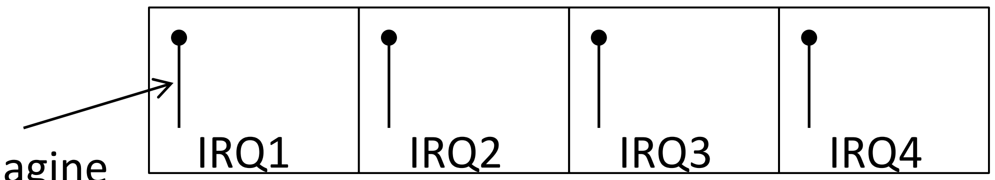
Further details available at: http://ww1.microchip.com/downloads/en/DeviceDoc/doc1779.pdf

Interrupt Control

- The 8-levels of priority
  - Different priority levels for interrupt source
  - Higher priority interrupts serviced while lower priority interrupt may be in process
- Internal interrupt triggering:
  - Level sensitive or edge triggered
- External interrupts triggering:
  - Positive-edge / negative-edge (we use negative edge in prac4) or
  - High-level / low-level sensitive

Masking interrupts

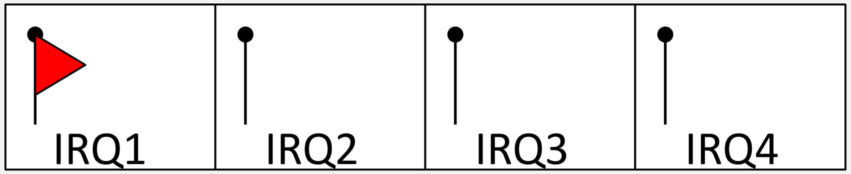
- An interrupt mask is simply a means to tell the CPU to ignore, or not respond to, certain interrupts
- Works the same as a bitmap mask



Can imagine
an interrupt is like
a ship’s signal flag

Masking interrupts

CPU in user mode

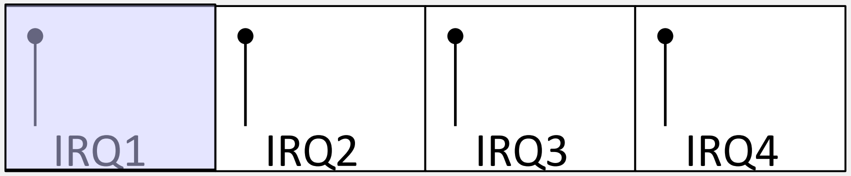


IRQ1 occurs!

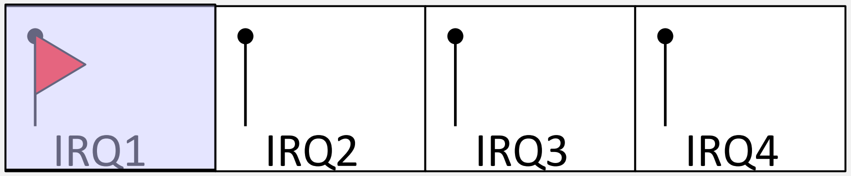
IRQ1 interrupt service routine (ISR)

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 again 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* the vector table a *read instruction* that reads address to jump to from an AIC peripheral register

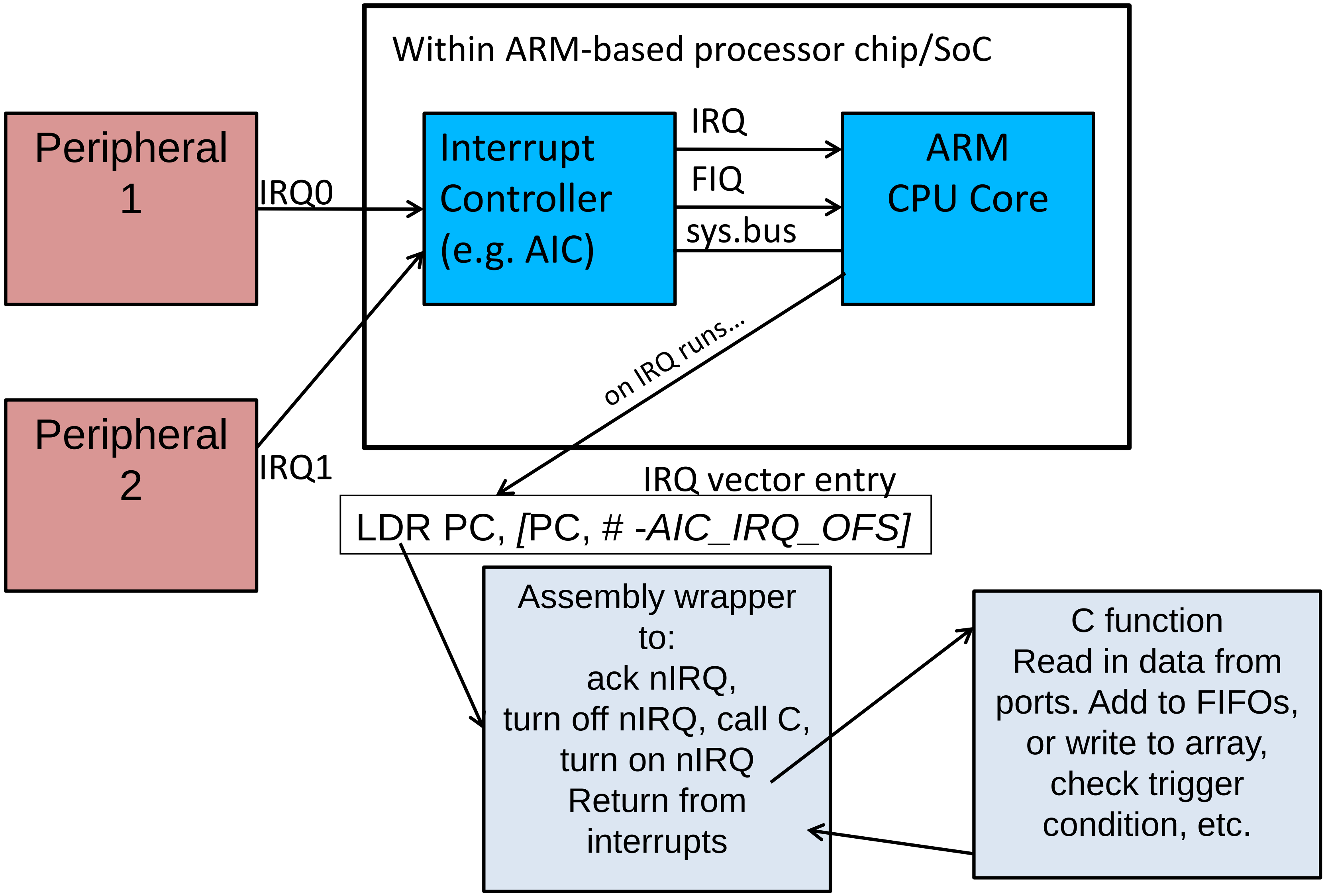
(this design concept is important to know because it has shown to be a very efficient and scalable means to implement interrupt logic while keeping the core simpler)

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.

Advanced Interrupt Operation



Within ARM-based processor chip/SoC
Peripheral 1
Peripheral 2
IRQ0
IRQ1
Interrupt Controller (e.g. AIC)
IRQ
FIQ
sys.bus
ARM CPU Core
on IRQ runs...
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.

Specialized Interrupt Behaviors

Nested Interrupts

- Most CPUs support **priority interrupts**
- **Interrupt nesting** refers to:
  - the ability of a high-priority interrupt to pre-empt a lower priority interrupt
- Simple systems seldom use interrupt nesting (too much logic needed) their solution:
  - the ISR simply disables all interrupts (i.e. masks 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 (in terms of the calling operation).
- A **reentrant ISR** is one that can be suspended (e.g. by a higher priority interrupt) and later resumed.

Reentrant Interrupts - Regulations

These three rules are used to determine whether a function is **safely reentrant**:

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

‘Atomic operations’ are operations which are visible to all parties and are (most importantly) <u>uninterruptible</u>

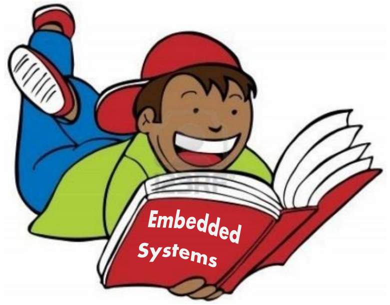
Interrupts, C and Assembly

- We can revisit some of the trickery interrupts issues in C and or Assembly next term

The Next Episode…

Lecture L20

More about DACs, ADCs & sampling



Reminder: Read sections

3.2.2-3.2.4 (S/H and Analog-to-Digital Converters)

3.6.1 (Digital-to-Analog Converters)

(3.3, 3.4 :
above mostly in EEE4084F High Performance Embedded Systems)

References

- https://en.wikipedia.org
- https://en.wikipedia.org/wiki/Carry-lookahead_adder
- Insights on using SPI on the ARM: https://www.voragotech.com/sites/default/files/Vorago%20SPI%20block%20use-%20%20AN%20-%20v1.0.pdf