VII. *Exceptions & Interrupts*

Jalal Kawash

Outline

1. Exceptions
2. Interrupts

Section 1

Exceptions

Section 1 Objectives

At the end of this section you will

1. Understand what exceptions are
2. List and entertain the 5 exception modes in ARM

Exceptions

- An *exception* is a condition that needs to halt normal execution
  - E.g.: Core reset, failed memory access, software interrupt (O.S.), attend to I/O, etc…
- An exception triggers the execution of an associated *exception handler*
  - A subroutine that attends to the needed work

ARM Exceptions

Exception	Mode	Main Purpose
Fast Interrupt Request	FIQ	FIR handling
Interrupt Request	IRQ	IR handling
Supervisor (SWI & Reset)	SVC	O.S. protected mode
Pre-fetch/Data Abort	Abort	Memory protection/virtual memory handling
Undefined Instruction	Undefined	Hardware coprocessors

Banked Registers

- Each processor mode has a set of registers
  - Called *register banks*
  - Banks partially overlap
  - Some registers are available in all modes
- When changing modes, registers that are banked are saved before mode changes

level view	System level views							
		Privileged modes						
			Exception modes					
	User mode	System mode	Supervisor mode	Monitor mode ‡	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon ‡	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon ‡	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon ‡	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

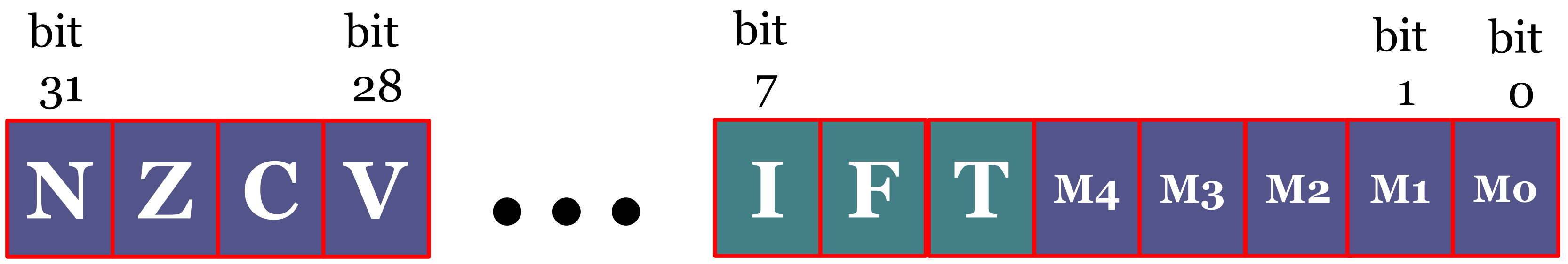
‡ Monitor mode and the associated banked registers are implemented only as part of the Security Extensions

From the ARM compiler toolchain reference

ARM Execution Modes

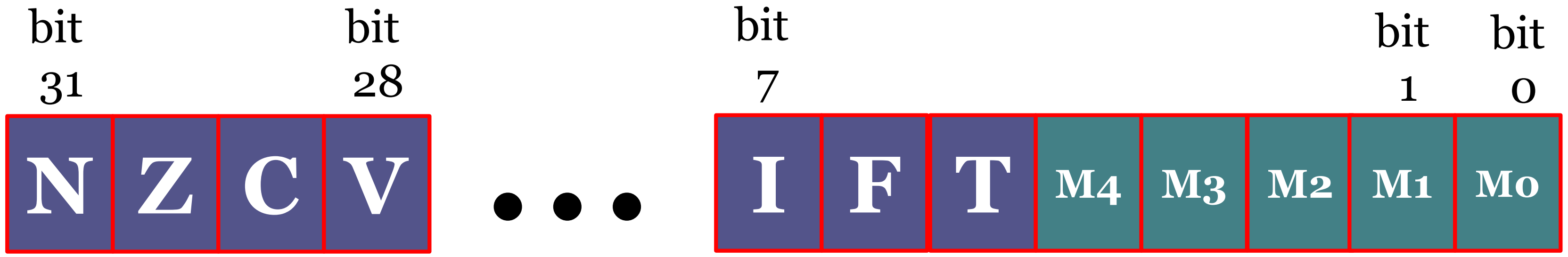
- An exception causes ARM to enter a corresponding mode
  - Except for user and system modes
    - These two modes are entered by manually changing CPSR
- An exception changes mode to *md*, the core:
  - SPSR_*md* = CPSR
  - LR_*md* = PC
  - Set CPSR to *md*
  - PC = address of exception handler

Recall: CPSR



- Flags: changed as a result of an ALU operation
- I/F : interrupt masks
  - Disable IRQ and FIQ when set
- T : status bit
  - If 1 => THUMB instructions / execution mode

Recall: CPSR – Mode Bits



- The combination of these 5 bits determine the mode
  - 10000 User Mode
  - 10001 FIQ Mode
  - 10010 IRQ Mode
  - 10011 Supervisor Mode
  - 10111 Abort Mode
  - 11011 Undefined Mode
  - 11111 System Mode
  - Any other value: unpredictable “mode”

SPSR

- Saved Program Status Register
- When an exception takes place, CPSR is saved in SPSR
- User and System are not entered by exceptions
    - No need for SPSR
    - If read in these modes, an unpredictable value results
    - If written, write is ignored

Section 2

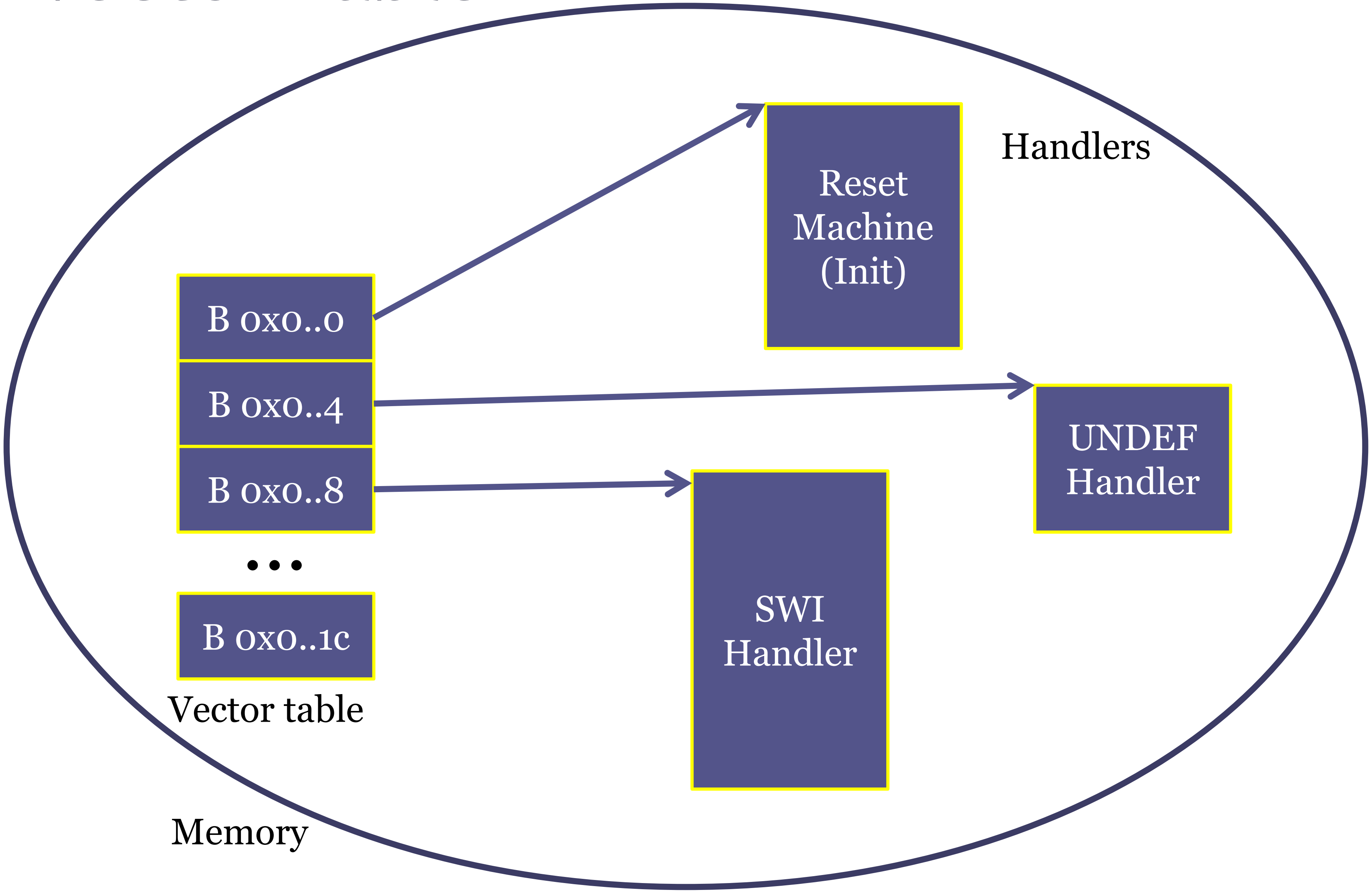
Interrupts

Section 2 Objectives

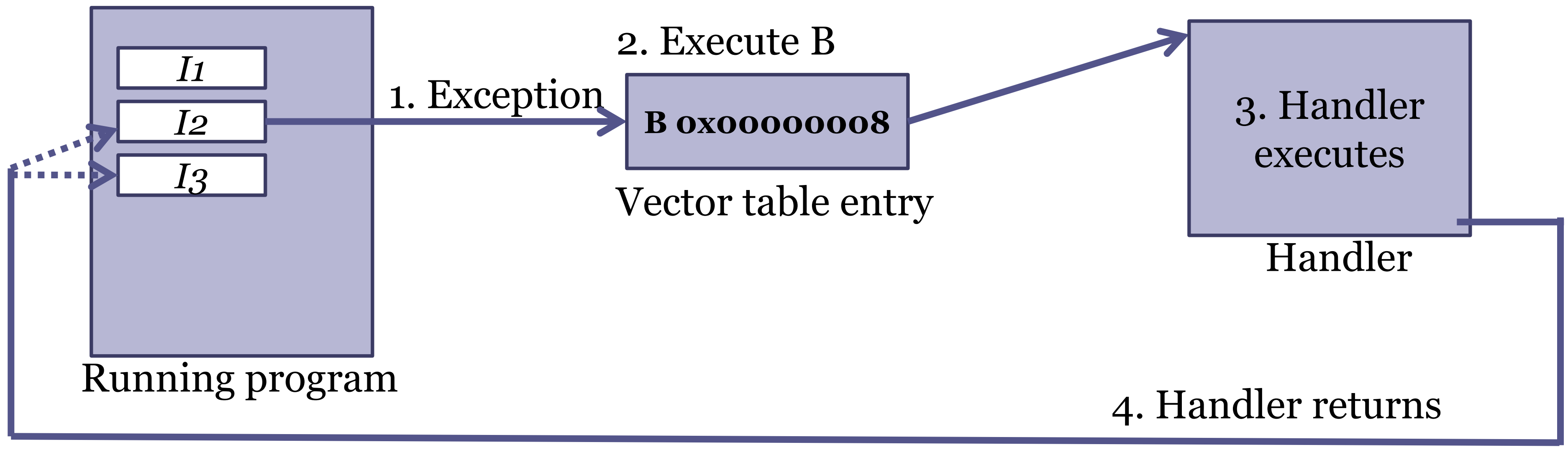
At the end of this section you will

1. Understand the purpose of the interrupt vector table and know how it is implemented
2. Understand what interrupts are
3. Be able to manually enable/disable interrupts
4. Write an interrupt handler and steal an existing interrupt

Vector Table



Vector Table Operation



ARM Vector Table

- Starts at address 0x00000000 or 0xffff0000
- Is a branch instruction to a specific address

Exception	Branch to	When
Reset	0	Booting/restart
Undefined Instruction	4	Processor cannot decode instruction
Software Interrupt	8	SWI call (O.S. routine)
Prefetch Abort	c	Attempt to invoke an instruction without access permissions
Data Abort	10	Invalid memory access
Reserved	14	
Interrupt Request	18	Hardware interrupt
Fast Interrupt Request	1c (28d)	Hardware high-priority interrupt

Branch to is relative to start address of vector table. For example, UNDEF is at address 0x00000004 or 0xffff0004

Branching

- Can be:
- B *relativeAddress*
  - Limited to 32 MB range
- LDR pc [pc, #offset]
  - Can branch to any place in memory
  - Slower: one extra memory access
- MOV pc #*immediate*
  - Full address space, but with limited alignment
    - Recall address is 8-bit rotated right by even number of bits
- Other

FIQ vector does not branch

- Since this is a high-priority interrupt, no time should be wasted
- Address 0x1C actually contains the first instruction of the handler
- This means that the FIQ handler is stored starting at address 0x1C

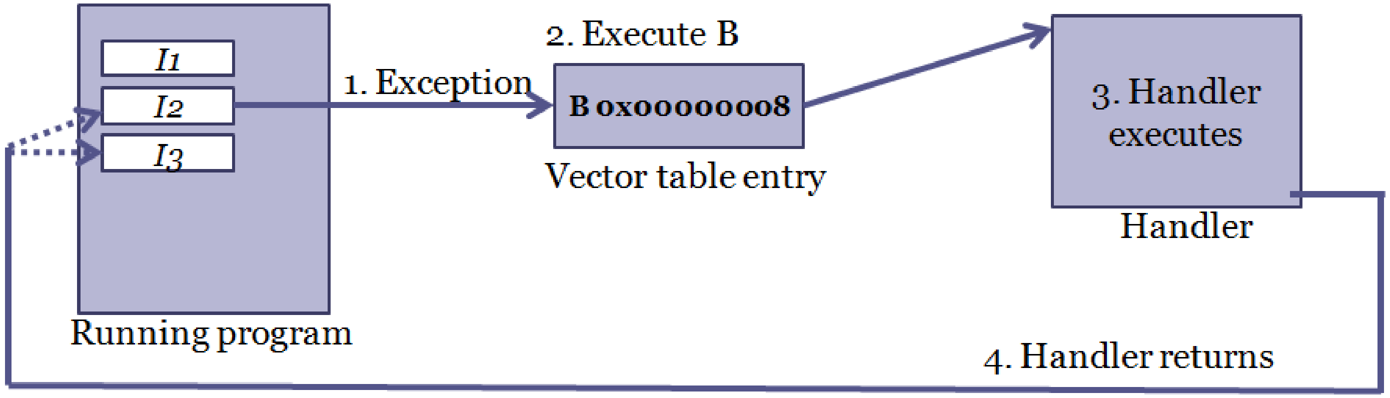
Exception Priorities

- When more than one exception occurs simultaneously, processor executes handler for higher priority exception

Exception	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	
FIQ	3	1	1
IRQ	4	1	
Prefetch Abort	5	1	
SWI(SVC)	6	1	
UNDEF	6	1	

Returning from an Exception

- On an exception, LR is updated
- LR is used to return from the exception handler



- LR must not be stomped on in the handler
- Where to return to depends on exception!

Where to Return to?

Exception	Return Address	Comments
Reset		LR is not defined
Data Abort	LR – 8	Instruction that caused the exception
FIQ	LR – 4	Instruction that caused the exception
IRQ	LR – 4	instruction that caused the exception
Prefetch Abort	LR – 4	Instruction that caused the exception
SWI	LR	Next instruction
Undefined Instruction	LR	Next instruction

Reading the manual

- [The prefetch abort exception] Occurs when the processor attempts to execute an instruction that has prefetched from an illegal address, that is, an address that the memory management subsystem has determined is inaccessible to the processor in its current mode.
- … Instructions already in the pipeline continue to execute *until* the invalid instruction is reached, at which point a prefetch abort is generated.
- **… because the program counter is not updated at the time the prefetch abort is issued, lr_ABT points to the instruction following the one that caused the exception. The handler must return to lr_ABT – 4**

Reading the manual

- [The Data Abort exception] Occurs when a data transfer instruction attempts to load or store data at an illegal address.
- **When a load or store instruction tries to access memory, the program counter has been updated.** A stored value of (pc – 4) in lr_ABT points to the second instruction beyond the address where the exception was generated. When the MMU has loaded the appropriate address into physical memory, the handler should return to the original, aborted instruction so that a second attempt can be made to execute it. **The return address is therefore two words (eight bytes) less than that in lr_ABT**

Returning from a Handler

- Directly (do not stomp on LR)
  - `SUBS pc, r14, #4 // pc = r14 – 14`
    - S & pc => CPSR is restored from SPSR
- Through the stack

```
handler:
  SUB r14, r14, #4 // r14 = r14 – 4
  STMFD r13!, {r0-r3, r14} // save regs on stack
  …
  <handler code>
  …
  LDMFD r13!, {r0-r3, pc}^// return
  // ^ causes CPSR = SPSR
```

Interrupts

- An *interrupt* is special type of an exception
- Two types:
  - IRQ and FIQ
    - Interrupts generated by external peripherals
  - SWI or SVC
    - System call (Call to O.S. routine)

IRQ and FIQ Interrupts

- Occur when a specific interrupt mask is cleared in the CPSR
- Core continues executing the current instruction
  - The instruction in the execution stage of pipeline
- Then, interrupt is handled

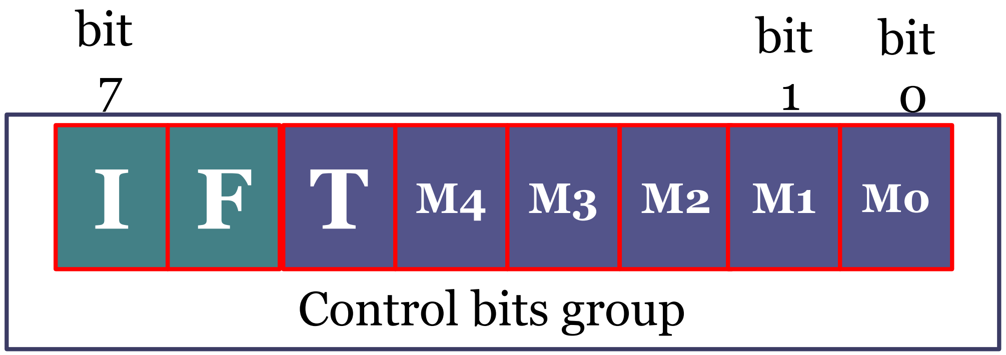
IRQ and FIQ Interrupts

- Core goes through the steps:

1. Core changes to a specific interrupt mode
   1. SPSR_*newMode* = CPSR
   2. LR_*newMode* = PC
2. Disable interrupts (of ≤ priority)
   1. Set the appropriate bits in CPSR
3. Branch to an entry in the Vector table
   1. PC = 0x18 (IRQ) or PC = 0x1c (FIQ)

MRS and MSR instructions

- MRS: Move to Register from Status
  - move CPSR or SPSR_*currentMode* to a GPR
- MSR: Move to Status from Register
  - Move GPRS to CPSR or SPSR_*currentMode*
  - Status suffixes:
    - _c bit 0 to 7 (of control field mask)
    - _x (extension)
    - _s (status)
    - _f (flags)



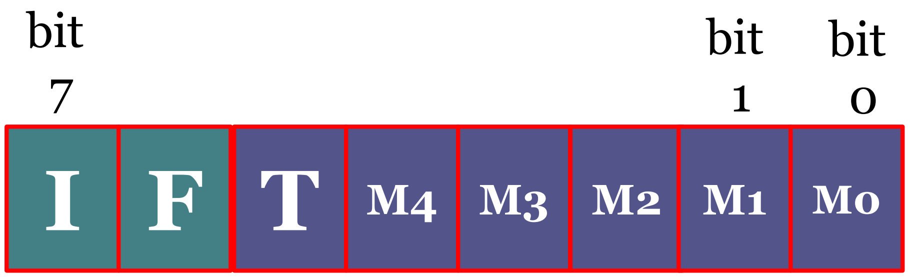
Enabling & Disabling IRQ/FIQ Interrupts

To Enable

```
// 1. copy cpsr to r1
MRS   r1, cpsr
// 2. clear I in r1
// r1 && not(10000000)
BIC   r1, #0x80
// or 0x40 for FIQ
// 3. set the control
//     bit in cpsr
MSR   cpsr_c, r1
```

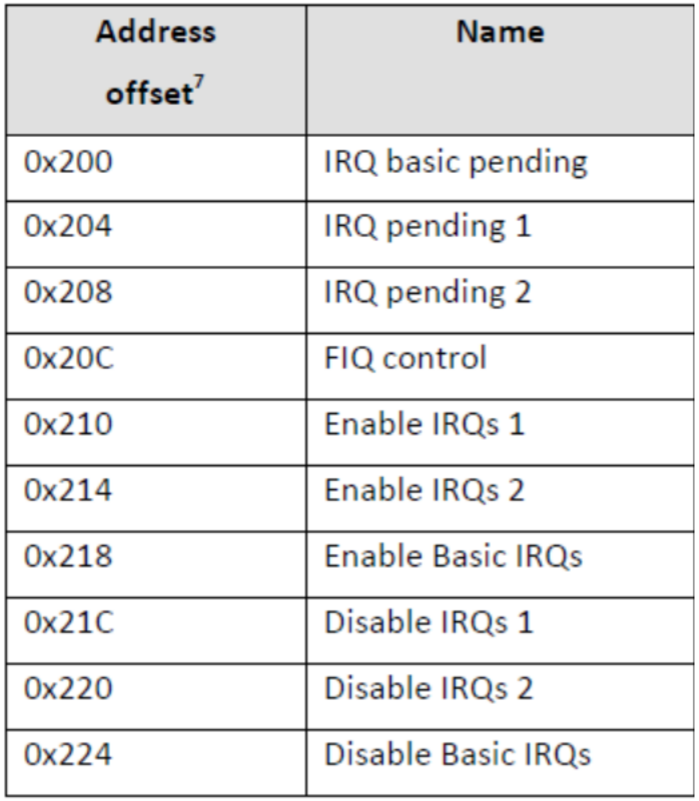
To Disable

```
MRS   r1, cpsr
ORR   r1, #0x80
// or 0x40 for FIQ
// 0x40 is 1000000
MSR   cpsr_c, r1
```



BCM 2835 Interrupt Registers

Memory-Mapped Registers



Address offset[7]	Name
0x200	IRQ basic pending
0x204	IRQ pending 1
0x208	IRQ pending 2
0x20C	FIQ control
0x210	Enable IRQs 1
0x214	Enable IRQs 2
0x218	Enable Basic IRQs
0x21C	Disable IRQs 1
0x220	Disable IRQs 2
0x224	Disable Basic IRQs

Base physical address is 0x2000B200

From BCM2835 ARM Peripherals

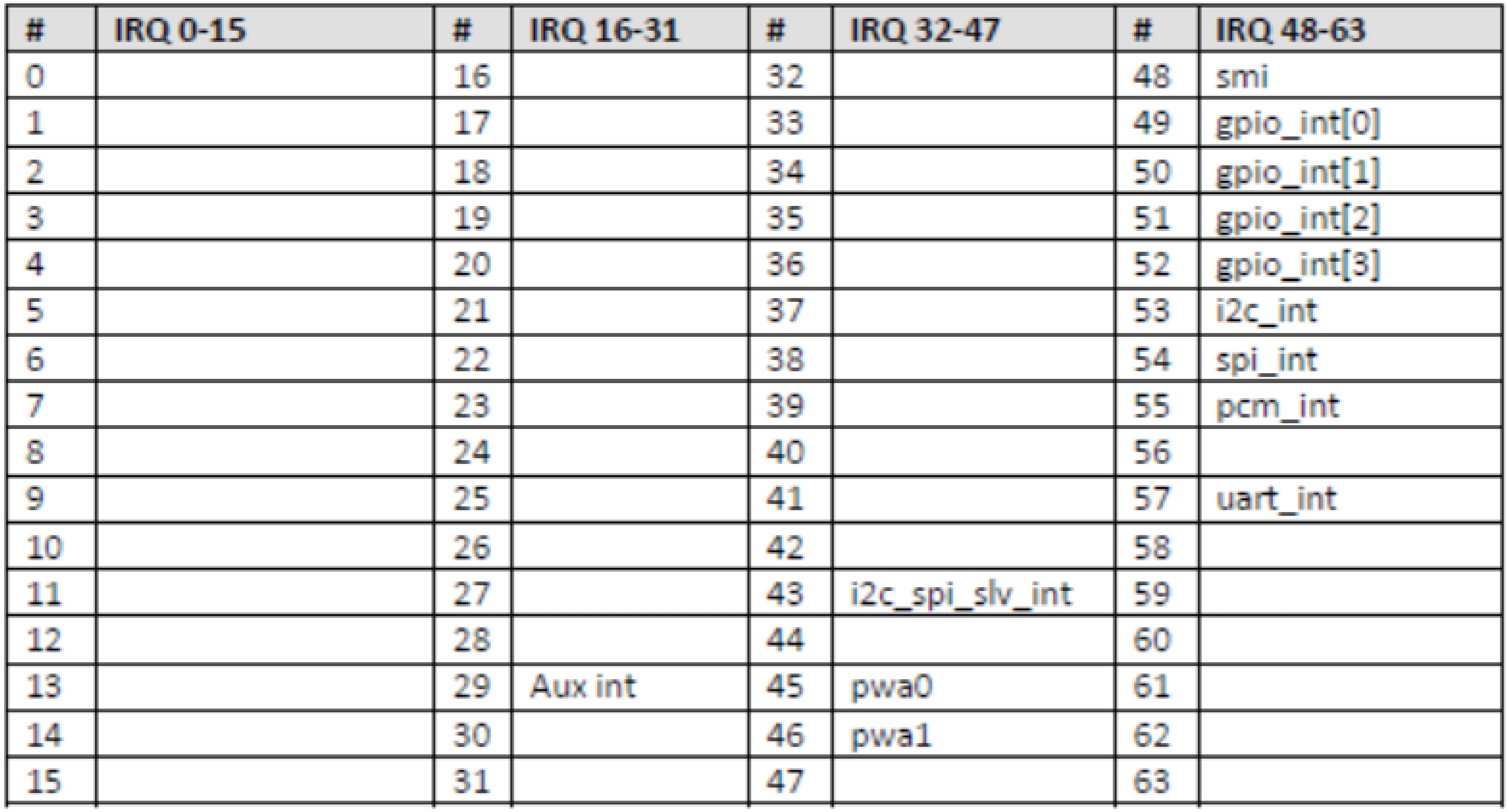
IRQ Basic Pending Register

- If 8th bit (bit # 7) is set, there is a pending interrupt in *pending register 1*
- If 9th bit is set, there is a pending interrupt in *Pending register 2*
- For other bits refer to the document

- Physical address: 0x2000B**200**

IRQ Interrupts

Empty: do not enable;
interfere with GPU operation



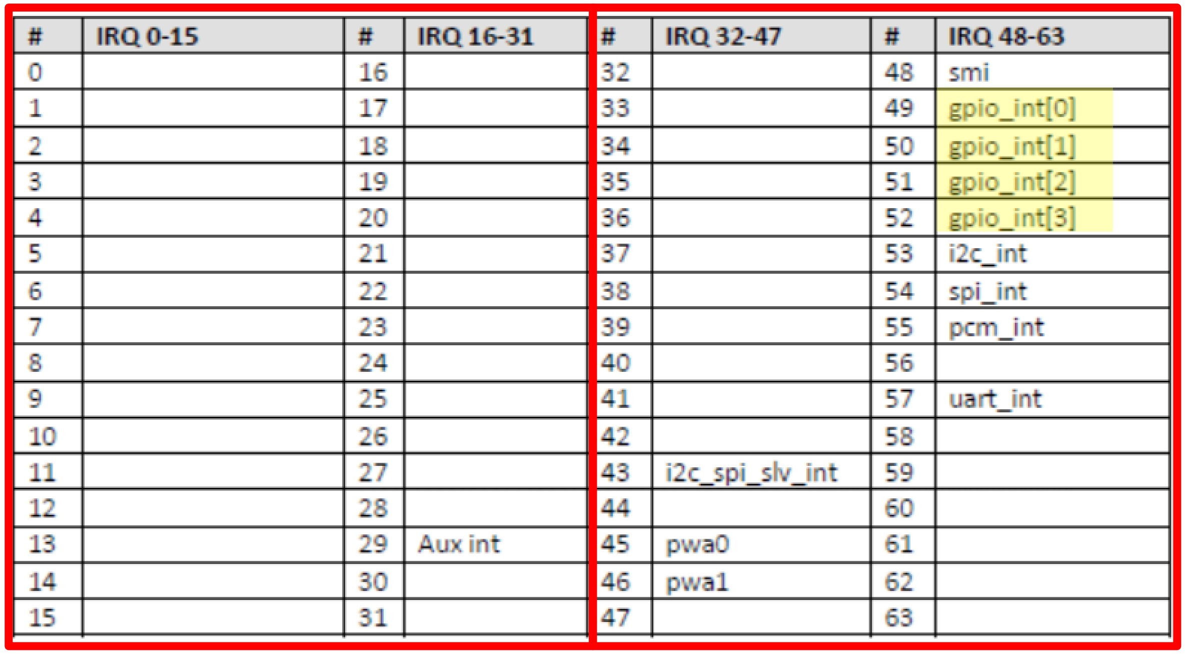
#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

From BCM2835 ARM Peripherals

IRQ Pending Registers 1 & 2

- IRQ Pending Register 1: Holds interrupts 0 to 31
- IRQ Pending Register 2: Holds interrupts 32 to 63

IRQ Pending Register 1 **IRQ Pending Register 2**



#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1		17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3		19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9		25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

Example Interrupt Service Routine

Steps to create an ISR

Example ISR

- Receive an IRQ interrupt
- ISR checks if SNES button is pressed
  - Updates a shared variable, say *SNESDat*
- If *SNESDat* is set, put the RPi LED on

Recall the ARM Vector Table

Exception	Branch to	When
Reset	0	Booting/restart
Undefined Instruction	4	Processor cannot decode instruction
Software Interrupt	8	SWI call (O.S. routine)
Prefetch Abort	c	Attempt to invoke an instruction without access permissions
Data Abort	10	Invalid memory access
Reserved	14	
Interrupt Request	18	Hardware interrupt
Fast Interrupt Request	1c	Hardware high-priority interrupt

Example ISR – Interrupt Table

```
IntTable:
/* Interrupt Vector Table (16 words) */
      ldr             pc, reset_handler
      ldr             pc, undefined_handler
      ldr             pc, swi_handler
      ldr             pc, prefetch_handler
      ldr             pc, data_handler
      ldr             pc, unused_handler
      ldr             pc, irq_handler
      ldr             pc, fiq_handler

reset_handler:      .word InstallIntTable
undefined_handler:  .word uhISR
swi_handler:        .word swiISR
prefetch_handler:   .word prefISR
data_handler:       .word dataISR
unused_handler:     .word unusedISR
irq_handler:        .word irqISR
fiq_handler:        .word fiqISR
```

If not using use hang
hang: b hang

Installing the Interrupt Table

```
ldr r0, =IntTable // where to read vector table from
mov r1, #0x0000 // where to write vector table to

1. load first 8 words and store to vector table address (from address r0 to r1)
while RIQ FIQ interrupts are disabled
        ...
2. load second 8 words and store to vector table address
        ...
3. Switch to IRQ mode and set stack pointer (for IRQ mode)
        4. Similarly, switch to Supervisor mode and set stack pointer
        ...
5. return
```

Installing the Interrupt Table

1. Load first 8 words and store to vector table address (from address r0 to r1) while RIQ FIQ interrupts are disabled

```
ldmia r0!, {r2, r3, r4, r5, r6, r7, r8, r9}
stmia r1!, {r2, r3, r4, r5, r6, r7, r8, r9}
```

2. Load second 8 words and store to vector table address

```
ldmia r0!, {r2, r3, r4, r5, r6, r7, r8, r9}
stmia r1!, {r2, r3, r4, r5, r6, r7, r8, r9}
```

Installing the Interrupt Table

3. Switch to IRQ mode and set stack pointer (for IRQ mode)

```
mov    r0, #0xD2      // 11010010 IRQ mode
                      // I & F bits high
msr    cpsr_c, r0     // disable interrupts
mov    sp, #0x8000
```

4. Similarly, switch to Supervisor mode and set stack pointer

```
mov    r0, #0xD3
msr    cpsr_c, r0
mov    sp, #0x8000000
```

5. Return

```
bx     lr
```

Example ISR – Checking for interrupts

1. Save registers used in the routine
2. Test if there is an interrupt pending in IRQ pending 2
   - Test 9th bit in *IRQ Basic Pending Reg.* (0x2000B200), if 0 no interrupts go to end of subroutine
   - Else check *IRQ pending* 2 (0x2000B208), check if any of gpio_int[0] to gpio_int[3] is set. If not, go to end of routine
3. Process the interrupt

Checking for Interrupts

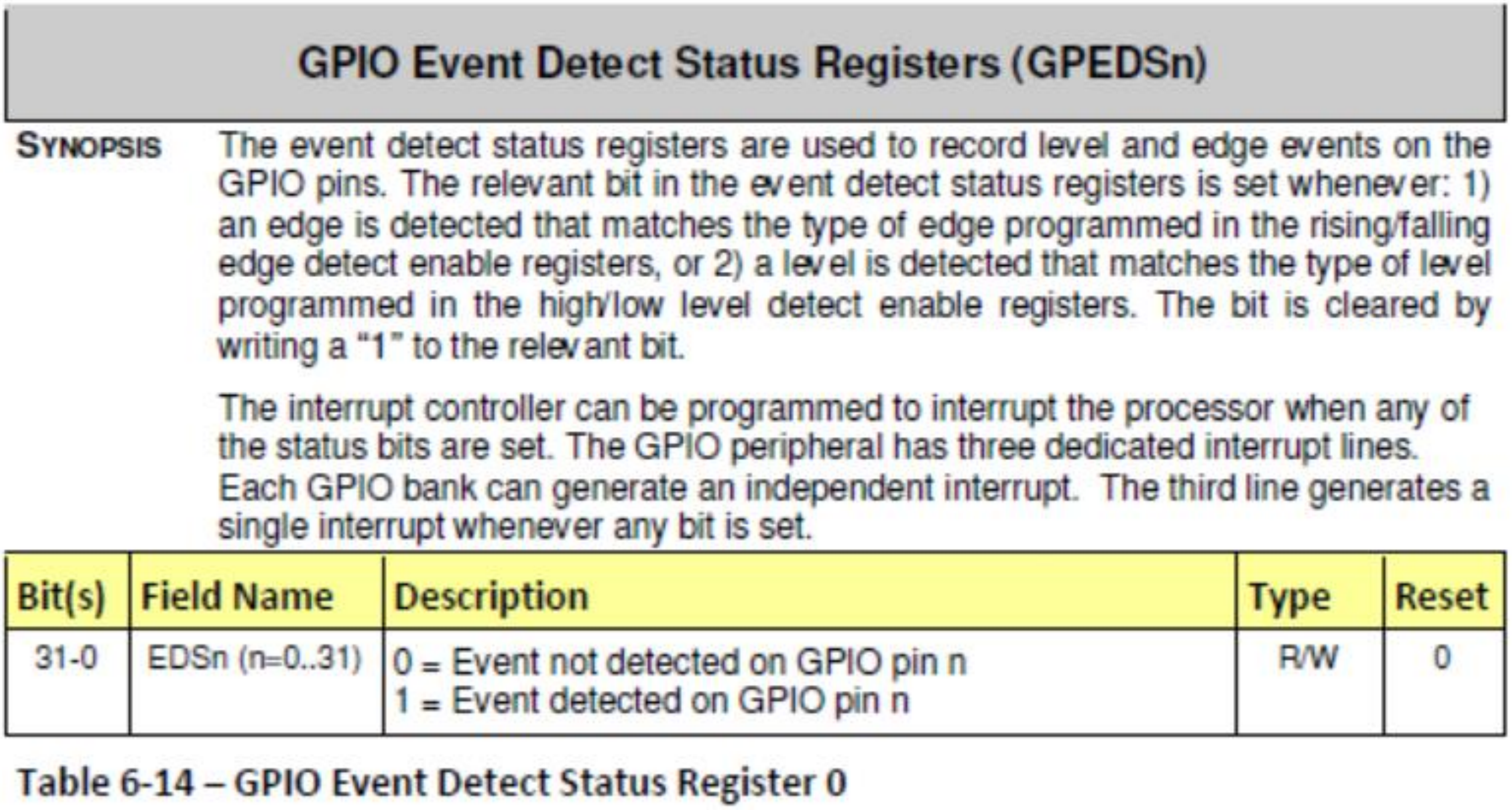
1. Test if there is interrupt pending in IRQ Pending Register 2

```
ldr    r0, =0x2000B200      // IRQ Basic Pending Reg.
ldr    r1, [r0]
tst    r1, #0x200           // bit 9
beq    irqEnd
```

2. Test that at least one of gpio_int[0] to gpio_int[3] is set

```
ldr    r0, =0x2000B208      // IRQ pending 2
ldr    r1, [r0]
tst    r1, #0x001E0000      // bits 17 to 20
beq           irqEnd
```

GPIO GPEDS0 Register



GPIO Event Detect Status Registers (GPEDSn)

SYNOPSIS The event detect status registers are used to record level and edge events on the GPIO pins. The relevant bit in the event detect status registers is set whenever: 1) an edge is detected that matches the type of edge programmed in the rising/falling edge detect enable registers, or 2) a level is detected that matches the type of level programmed in the high/low level detect enable registers. The bit is cleared by writing a "1" to the relevant bit.

The interrupt controller can be programmed to interrupt the processor when any of the status bits are set. The GPIO peripheral has three dedicated interrupt lines. Each GPIO bank can generate an independent interrupt. The third line generates a single interrupt whenever any bit is set.

Bit(s)	Field Name	Description	Type	Reset
31-0	EDSn (n=0..31)	0 = Event not detected on GPIO pin n 1 = Event detected on GPIO pin n	R/W	0

Table 6-14 – GPIO Event Detect Status Register 0

From BCM2835 ARM Peripherals

Example ISR – Processing Interrupt

1. Test GPEDS0 (0x20200040) for pin 10 (SNES DAT line); if high, we have an event; otherwise, there is no event go to the end of the routine

2. If there is an event, store a value in a shared variable indicating so (SNESDat)

3. Clear bit 10 in GPEDS0 to acknowledge the interrupt

4. End of subroutine: restore registers and return
   - `subs pc, lr, #4`

ISR

1. Test if pin 10 (SNES DAT line) is reporting an event in GPEDS0

```
ldr   r0, =0x20200040 // GPEDS0
// GPIO event detect status 0
ldr   r1, [r0]
tst   r1, #0x400    // bit 10
beq   irqEnd
```

2. Invert the LSB in SNESDat

```
SNESDat:  .int 0
```

```
ldr   r0, =SNESDat
ldr   r1, [r0]
eor   r1, #1
str   r1, [r0]
```

ISR

3. Clear bit 10 in GPEDS0 to acknowledge the interrupt

```
ldr   r0, =0x20200040
mov   r1, #0x400
str   r1, [r0]
```

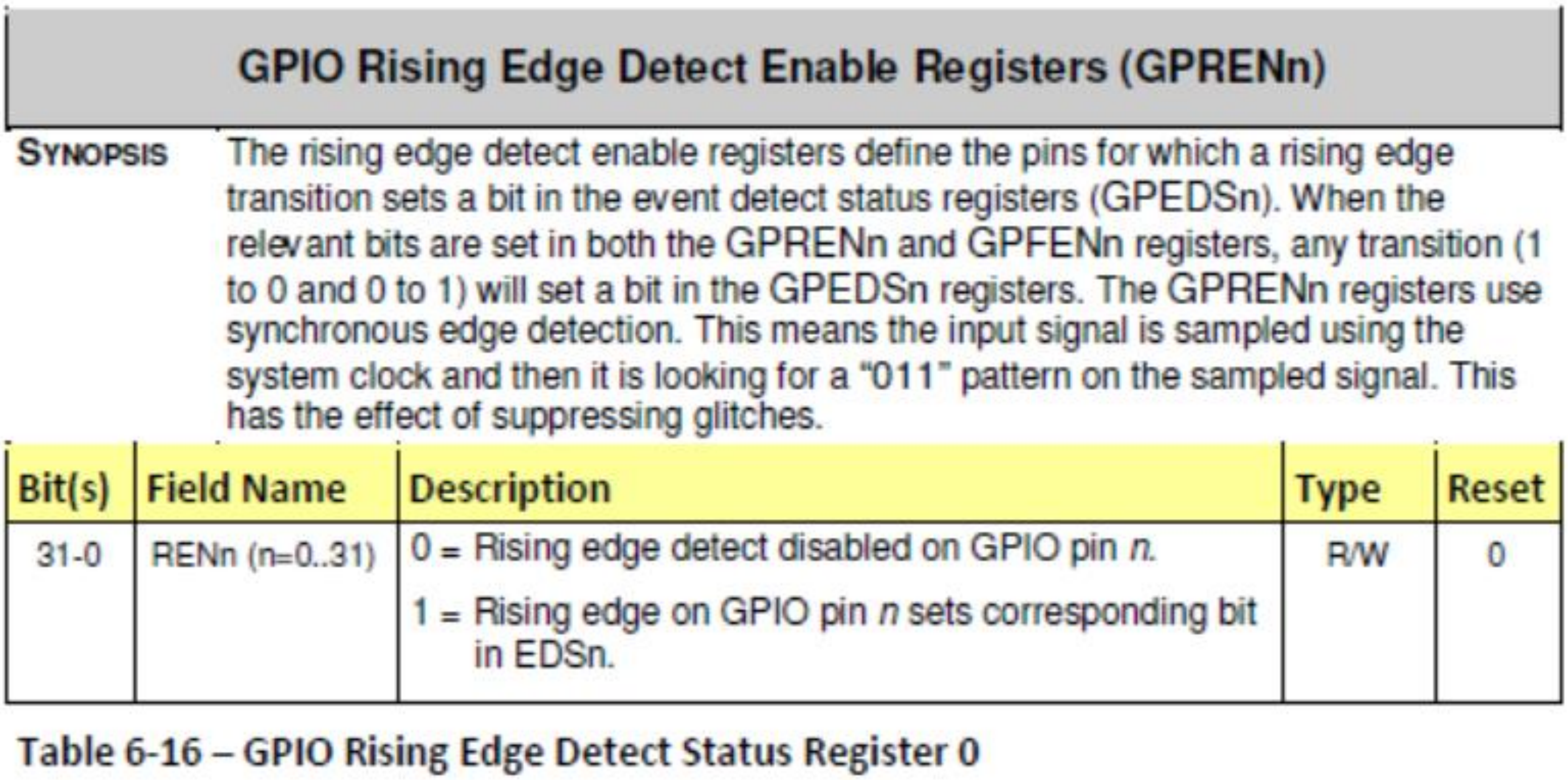
4. Return

```
pop  {r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11,
r12, lr}
subs  pc, lr, #4
```

Example ISR – main Routine

1. Install the interrupt table
2. Enable JTAG
3. Initialize the SNES controller
4. Signal to controller to sample buttons (LAT line)
5. Set pin 16 GPIO function to write

GPIO GPREN0 Register



GPIO Rising Edge Detect Enable Registers (GPRENn)

SYNOPSIS The rising edge detect enable registers define the pins for which a rising edge transition sets a bit in the event detect status registers (GPEDSn). When the relevant bits are set in both the GPRENn and GPFENn registers, any transition (1 to 0 and 0 to 1) will set a bit in the GPEDSn registers. The GPRENn registers use synchronous edge detection. This means the input signal is sampled using the system clock and then it is looking for a "011" pattern on the sampled signal. This has the effect of suppressing glitches.

Bit(s)	Field Name	Description	Type	Reset
31-0	RENn (n=0..31)	0 = Rising edge detect disabled on GPIO pin *n*. 1 = Rising edge on GPIO pin *n* sets corresponding bit in EDSn.	R/W	0

Table 6-16 – GPIO Rising Edge Detect Status Register 0

From BCM2835 ARM Peripherals

Example ISR – main routine

6. Set the Rising Edge Detect bit for pin 10 (DAT line)
7. Enable GPREN0 – set IRQs 49 to 50 (gpio_int[0] to gpio_int[3])
8. Enable IRQ

```
    mrs         r0, cpsr
    bic         r0, #0x80
    msr         cpsr_c, r0
```

9. Check in a loop for the shared variable (SNESDat) to be set; if so, set GPIO pin 16 (LED)

Example ISR – main (1)

```
main:
        bl                  InstallIntTable
        bl                  EnableJTAG
        bl                  InitSNESController

        bl                  SetLATHigh        // Signal to
controller to
                                   // sample buttons

     mov     r0, #16 // use pin 16 to communicate with LED
     mov     r1, #1 // write function
     bl      SetGpioFunction
```

Example ISR – main (2)

```
        // set the Rising Edge Detect bit for pin 10
        ldr                 r0, =0x2020004C // GPREN0
        ldr                 r1, [r0]
        orr                 r1, #0x400 // bit 10
        str                 r1, [r0]

        ldr                 r0, =0x2000B214 // Enable IRQs 2
        mov                 r1, #0x001E0000 // bits 17 to 20
set
        // (IRQs 49 to 52; gpio_int[0] to gpio_int[3])
        str                 r1, [r0]

        // Enable IRQ
        mrs                 r0, cpsr
        bic                 r0, #0x80
        msr                 cpsr_c, r0
```

Example ISR – main (3)

```
ledLoop$:
        // set GPIO pin 16 (ACT LED) based on value in
SNESDat //(changed by irq)
        ldr                 r1, =SNESDat
        ldr                 r1, [r1]
        mov                 r0, #16
        bl                  SetGpio

        b                   ledLoop$



hang:
        b                   hang
```