



**Trinity College Dublin**

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

# 07 – Exceptions and Interrupts

CS1022 – Introduction to Computing II

Dr Jonathan Dukes / [jdukes@tcd.ie](mailto:jdukes@tcd.ie)

School of Computer Science and Statistics



**Trinity College Dublin**

Coláiste na Tríonóide, Baile Átha Cliath

The University of Dublin

# PART I – Exceptions

CS1022 – Introduction to Computing II

Dr Jonathan Dukes / [jdukes@tcd.ie](mailto:jdukes@tcd.ie)

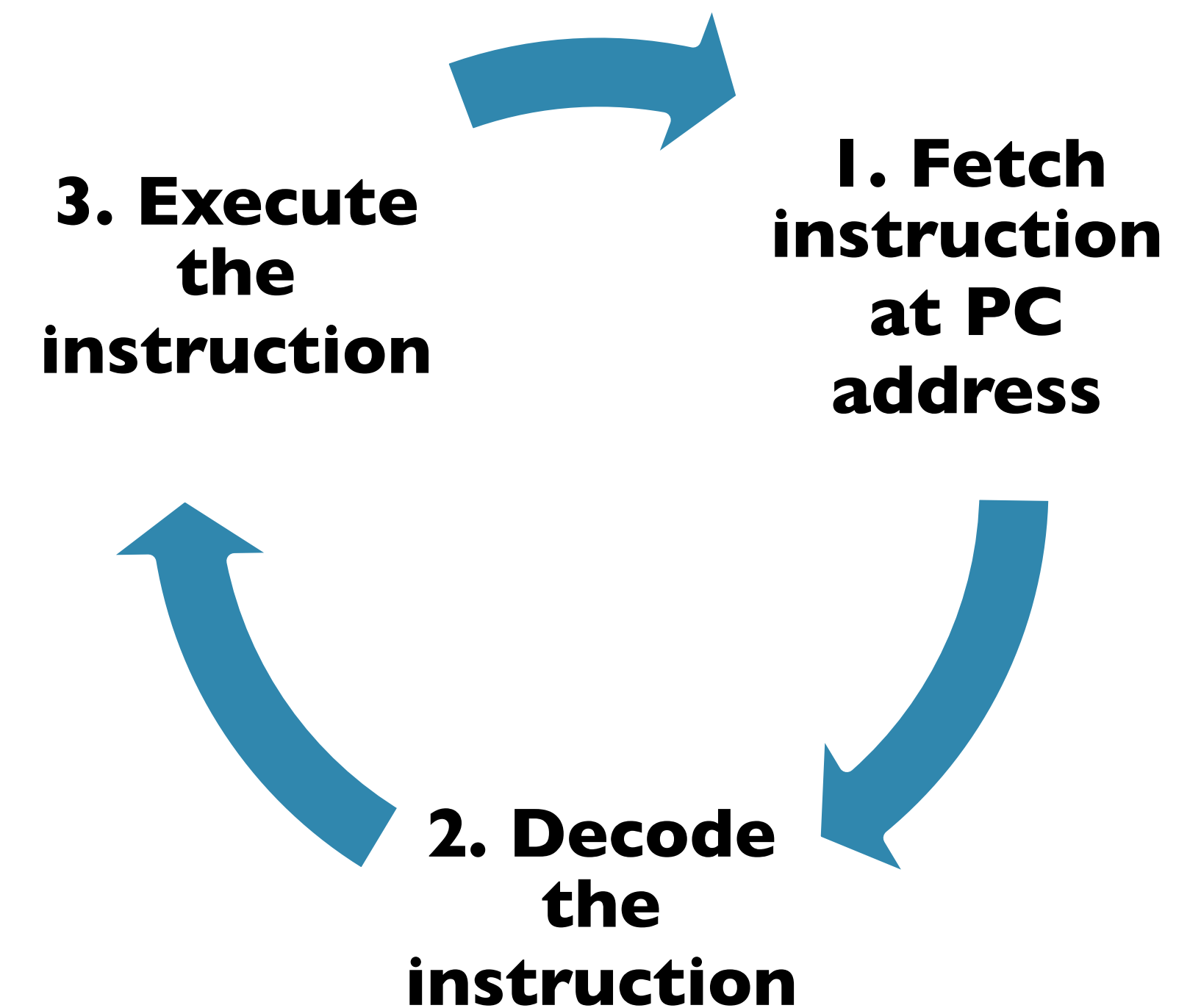
School of Computer Science and Statistics

Exceptions are events outside the normal (expected?) flow of execution of a program

Suppose the processor fetches the next instruction from memory but the instruction word is not a valid instruction

Suppose we try to write a word to address 0xA1000000 but the memory device at that address is a read-only ROM device

Suppose we're executing an instruction and someone presses the RESET butto ...



Expected events that occur at unpredictable times (with respect to the execution of our program!)

Mouse clicks, keyboard presses, touchscreen presses, button presses, ...

Receive network data, finish sending network data, ...

Wait for a timer to count down to zero, ...

Unexpected events that we can try to handle

Eject CD, remove USB key, ...

Battery power low, ...

Loose wireless network signal, network cable unplugged, ...

Read from or write to invalid memory addresses

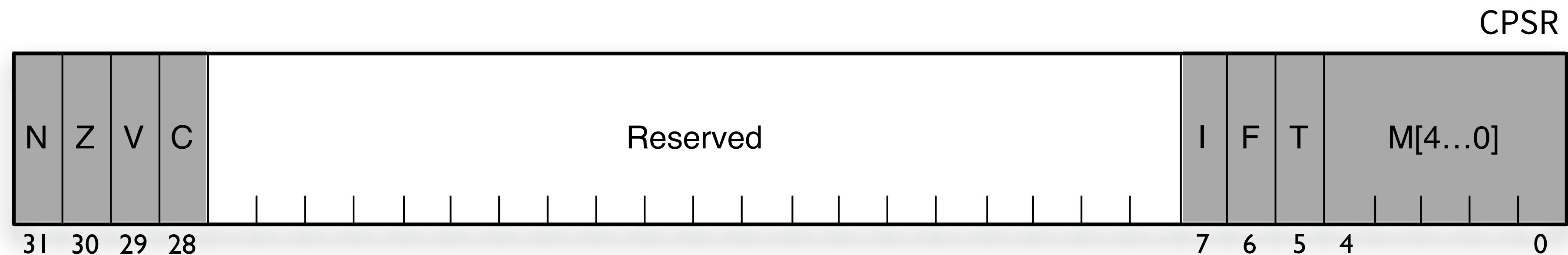
Attempting to execute invalid machine code

Reset

Name	Description
Reset	Occurs at power-on or when RESET button is pressed
Undefined Instruction	Attempt to execute an invalid instruction word
Software Interrupt (SWI)	Caused programmatically by executing SWI instruction
Prefetch Abort	Attempt to fetch an instruction from an invalid address (instructions)
Data Abort	Attempt to load/store from/to an invalid address (data)
(Reserved)	
IRQ	Interrupt ReQuest
FIQ	Fast Interrupt reQuest

When one of these exceptions occurs, it can be “handled” by an “exception handler” (*think **subroutine!***) that takes appropriate action

Processor Mode		Mode #	Description
User	usr	0b10000	Normal program execution
FIQ	fiq	0b10001	Fast Interrupt reQuest handling (low overhead)
IRQ	irq	0b10010	General purpose interrupt handling
Supervisor	svc	0b10011	Protected mode for OS (Reset / SWI)
Abort	abt	0b10111	Prefetch / Data Abort (memory management)
Undefined	und	0b11011	Software emulation of unimplemented instructions
System	sys	0b11111	Privileged mode using user-mode registers (OS)





# ARM Programmers Model Registers

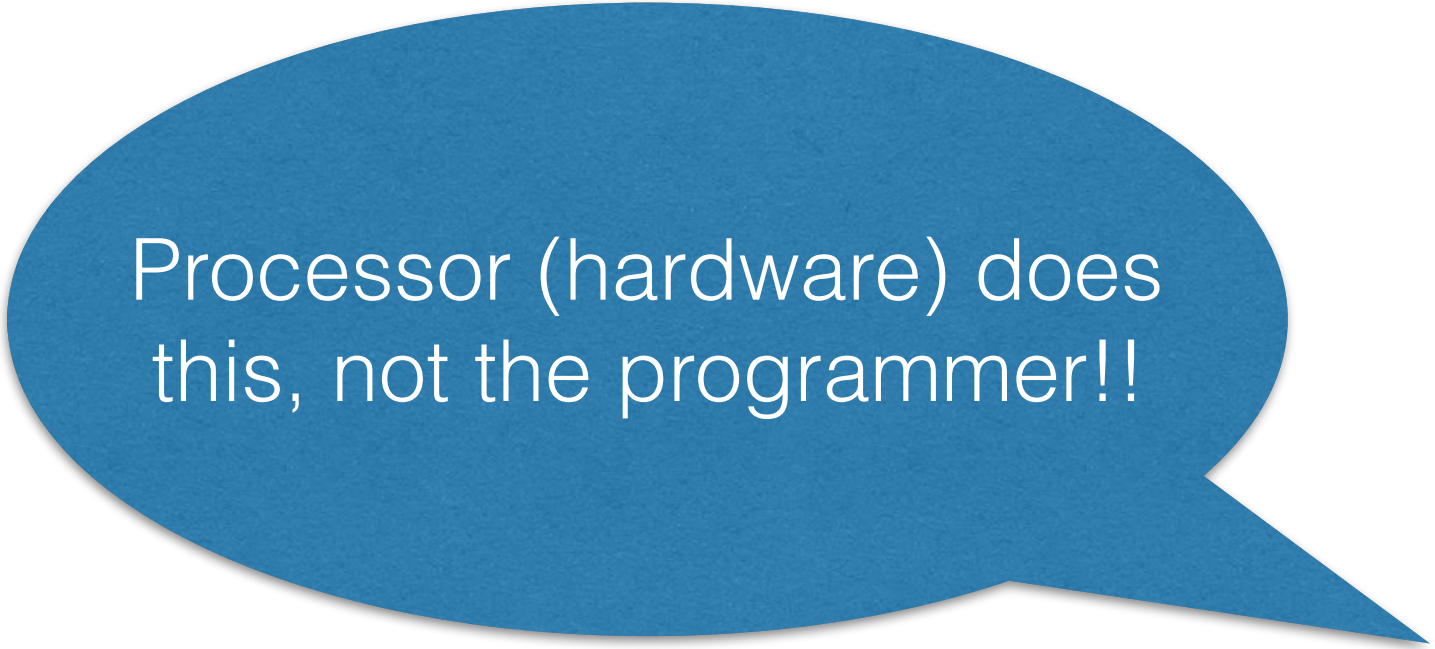
ARM Architecture Reference  
Manual (Issue I), Figure A2-1

Modes						
<div><div>Privileged modes</div><div>Exception modes</div></div>						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	<div>R8_fiq</div>
R9	R9	R9	R9	R9	R9	<div>R9_fiq</div>
R10	R10	R10	R10	R10	R10	<div>R10_fiq</div>
R11	R11	R11	R11	R11	R11	<div>R11_fiq</div>
R12	R12	R12	R12	R12	R12	<div>R12_fiq</div>
R13	R13	<div>R13_svc</div>	<div>R13_abt</div>	<div>R13_und</div>	<div>R13_irq</div>	<div>R13_fiq</div>
R14	R14	<div>R14_svc</div>	<div>R14_abt</div>	<div>R14_und</div>	<div>R14_irq</div>	<div>R14_fiq</div>
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		<div>SPSR_svc</div>	<div>SPSR_abt</div>	<div>SPSR_und</div>	<div>SPSR_irq</div>	<div>SPSR_fiq</div>

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

1. Complete execution of current instruction
2. Save current state and update CPSR
  - i. Save return address in LR\_<mode>  
(<mode> will be the new processor mode while handling the exception)
  - ii. CPSR saved to SPSR\_<mode>
  - iii. Switch to ARM state (clear T bit)  
(Exceptions are always handled in ARM state)
  - iv. Set mode (M[4...0] bits)  
(Mode appropriate to exception type)
  - v. Disable IRQs (set I bit) for all exceptions
  - vi. Disable FIQs (set F bit) when handling FIQs and Reset
3. Change PC to address of the exception handler subroutine corresponding to the exception type



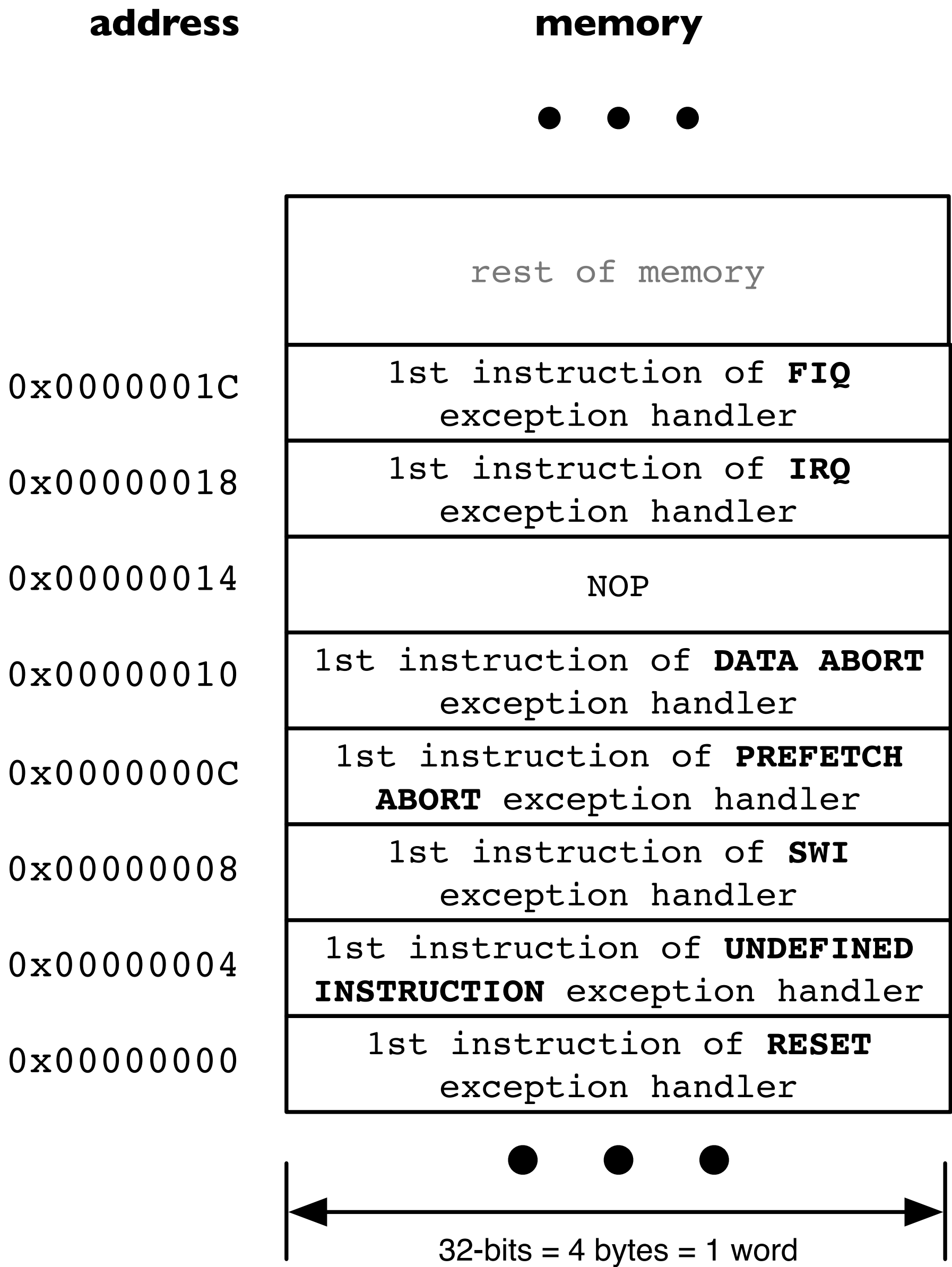
Processor (hardware) does this, not the programmer!!



Handler for each exception type starts at a **fixed, pre-defined address**

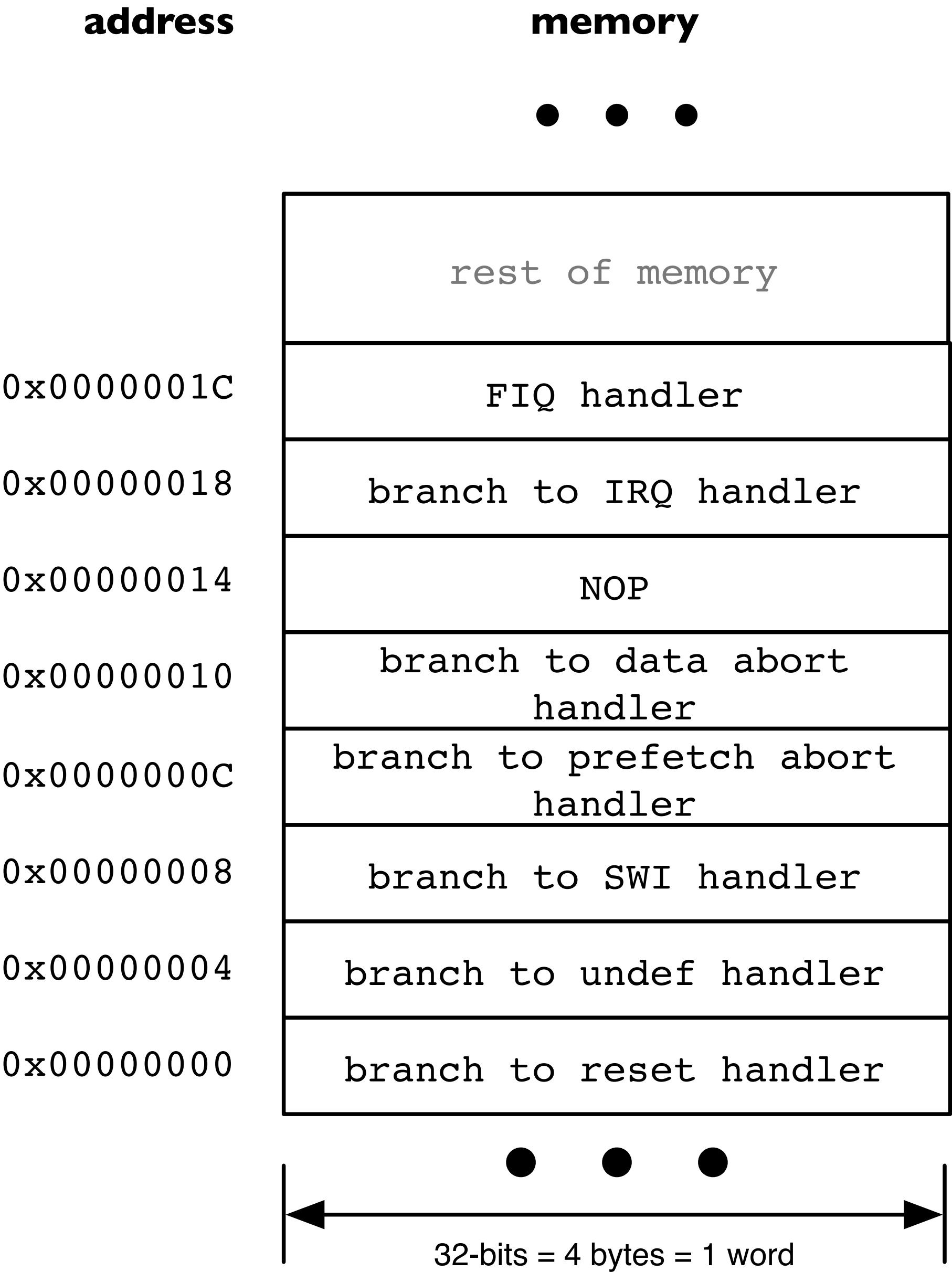
Processor loads PC with this address

How many instructions are there in each exception handler?



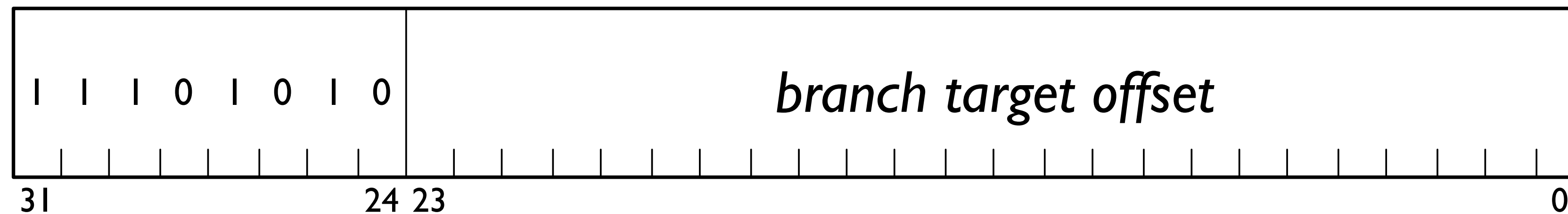
Eight words at the start of memory contain instructions that cause the processor to branch to the real exception handler for each exception type

This is the **Exception Vector Table**



A number of ways we can branch to the start of the real exception handler ...

A branch (B) instruction – range limited to  $\pm 32\text{MB}$



A MOV instruction (MOV PC,<addr>) – limited to jumping to an address that can be represented as a byte shifted by an even number of bits

An LDR (LDR PC,[offset]) instruction – loads start address of the real exception from PC+offset into the PC (but we need to store the real exception handler start addresses in memory)

	rest of memory
EVT + 0x18	branch to IRQ handler
EVT + 0x14	NOP
EVT + 0x10	branch to data abort handler
EVT + 0x0C	branch to prefetch abort handler
EVT + 0x08	branch to SWI handler
EVT + 0x04	branch to undef handler
EVT + 0x00	branch to reset handler
	. . .
0x0000001C	FIQ handler
0x00000018	branch to IRQ handler
0x00000014	NOP
0x00000010	branch to data abort handler
0x0000000C	branch to prefetch abort handler
0x00000008	branch to SWI handler
0x00000004	branch to undef handler
0x00000000	branch to reset handler

```
; e.g. branch to SWI handler
      LDR    PC, [EVT + 0x08]
```

```
; e.g. branch to reset handler
      LDR    PC, [EVT + 0x00]
```



## 13

```
; Exception Vectors
; Mapped to Address 0.
; Absolute addressing mode must be used.
; Dummy Handlers are implemented as infinite loops which can be modified.
```

```
; Address 0x00000000
```

# Labels

DCD	Reset_Handler
DCD	Undef_Handler
DCD	SWI_Handler
DCD	PAbt_Handler
DCD	DAbt_Handler
DCD	
DCD	
DCD	FIQ_Handler
...	...

## FIQ Handler

**Trinity College Dublin, The University of Dublin**



Must always be a handler for each exception, even if it does nothing ...

Undef_Handler	B	Undef_Handler
SWI_Handler	B	SWI_Handler
PAbt_Handler	B	PAbt_Handler
DAbt_Handler	B	DAbt_Handler
IRQ_Handler	B	IRQ_Handler
FIQ_Handler	B	FIQ_Handler

; Reset Handler

Reset\_Handler

< reset handler code goes here >

IRQ handler address is specified differently (more later)

FIQ is a special case and is often implemented differently (designed to reduce overheads and allow faster exception handling)

Always have a RESET handler to perform system initialisation

ARM instructions are 32-bits long

$2^{32}$  possible instruction words

Not all instruction words are valid

Invalid instructions raise undefined instruction exceptions

Take advantage of this to extend the instruction set with our own instructions (***instruction emulation***)

Instruction operation must be implemented in software

When the processor attempts to execute it, an undefined instruction exception is raised

Undefined instruction exception handler is executed

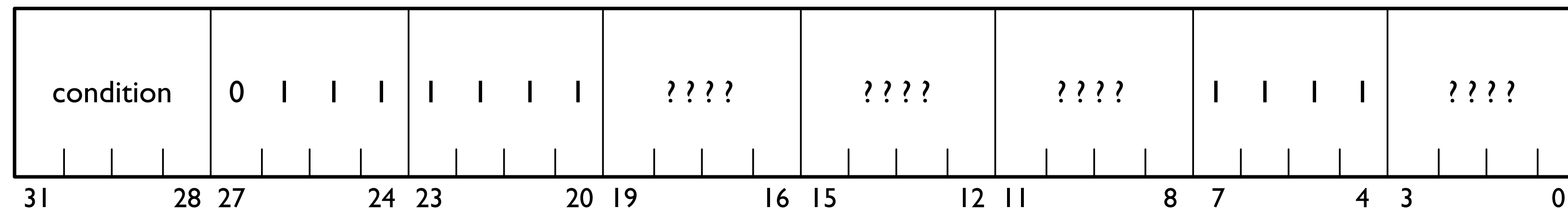
We provide our own undefined instruction exception handler to decode the instruction and implement the desired operation

# Example – Undefined POWER instruction

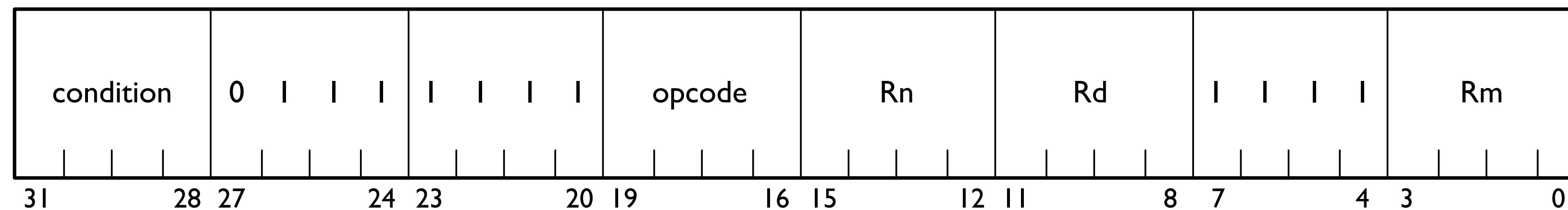
16

Provide a POWER instruction to compute  $xy$

Define our instruction template



undefined  
instruction  
space



$Rd = Rm^{Rn}$

1. Write our Undefined exception handler
2. Set up the vector table
3. Test the instruction

# Example – Undefined POWER instruction

17

UndefHandler

```
        STMFD    sp!, {r0-r12, LR}        ; save registers

        LDR      r4, [lr, #-4]             ; load undefined instruction
        BIC      r5, r4, #0xFFF0FFFF      ; clear all but opcode bits
        TEQ      r5, #0x00010000          ; check for undefined opcode 0x1
        BNE      endif1                   ; if (power instruction) {

        BIC      r5, r4, #0xFFFFFFFF0     ; isolate Rm register number
        BIC      r6, r4, #0xFFFF0FFF      ; isolate Rn register number
        MOV      r6, r6, LSR #12           ;
        BIC      r7, r4, #0xFFFFF0FF      ; isolate Rd register number
        MOV      r7, r7, LSR #8            ;

        LDR      r1, [sp, r5, LSL #2]      ; read saved Rm from stack (don't pop)
        LDR      r2, [sp, r6, LSL #2]      ; read saved Rn from stack (don't pop)

        BL       power                     ; call pow subroutine

        STR      r0, [sp, r7, LSL #2]      ; save result over saved Rd
endif1   ; }
        LDMFD    sp!, {r0-r12, PC}^       ; restore register and CPSR
```

```
;
; Install address of Exception Handler in handler address table
;
LDR    r4, =0x40000024      ; 0x00000024 is mapped to 0x40000024!
LDR    r5, =UndefHandler    ; Address of new undefined handler
STR    r5, [r4]             ; Store new undef handler address in table

;
; Test our new instruction
;
LDR    r4, =3                ; test 3^4
LDR    r5, =4                ;

; This is our undefined instruction opcode
DCD    0x77F150F4            ; POW r0, r4, r5 (r0 = r4 ^ r5)

; R0 should be 81 now!
```



# Example – Undefined POWER instruction

19

```
; power subroutine
; Computes x^y
; paramaters:  r1: x (value)
;              r2: y (value)
; return:      r0: result (variable)
power
    STMFD    sp!, {r1-r2,lr}        ; save registers

    CMP      r2, #0                 ; if (y = 0)
    BNE      else2                   ; {
    MOV      r0, #1                   ; result = 1
    B        endif2                   ; }

else2    ; else {
    MOV      r0, r1                   ; result = x
    SUBS     r2, r2, #1               ; y = y - 1
    BEQ      endif3                  ; if (y != 0) {
do4      ; do {
    MUL      r0, r1, r0               ; result = result * x
    SUBS     r2, r2, #1               ; y = y - 1
    BNE      do4                      ; } while (y != 0)
endif3    ; }
endif2    ; }

    LDMFD    sp!, {r1-r2, pc} ; restore registers and return
```



**Trinity College Dublin**  
Coláiste na Tríonóide, Baile Átha Cliath  
The University of Dublin

# Part II – Interrupts

CS1022 – Introduction to Computing II

Dr Jonathan Dukes / [jdukes@tcd.ie](mailto:jdukes@tcd.ie)  
School of Computer Science and Statistics

Want to write a program to take some action when a button is pressed

## Approach 1

suppose a memory location (e.g. 0xE1000000) is set to 1 when the button is pressed

keep reading 0xE1000000 until we read 0

do nothing between tests

## Approach 2

keep reading 0xE1000000 until we read 0

do useful things between tests

***Polling***

## Approach 3

do useful things

break out of F-D-E cycle when event occurs

***Interrupts***



Interrupt ReQuest (IRQs) } Exception Classes  
Fast Interrupt reQuest (FIQs) }

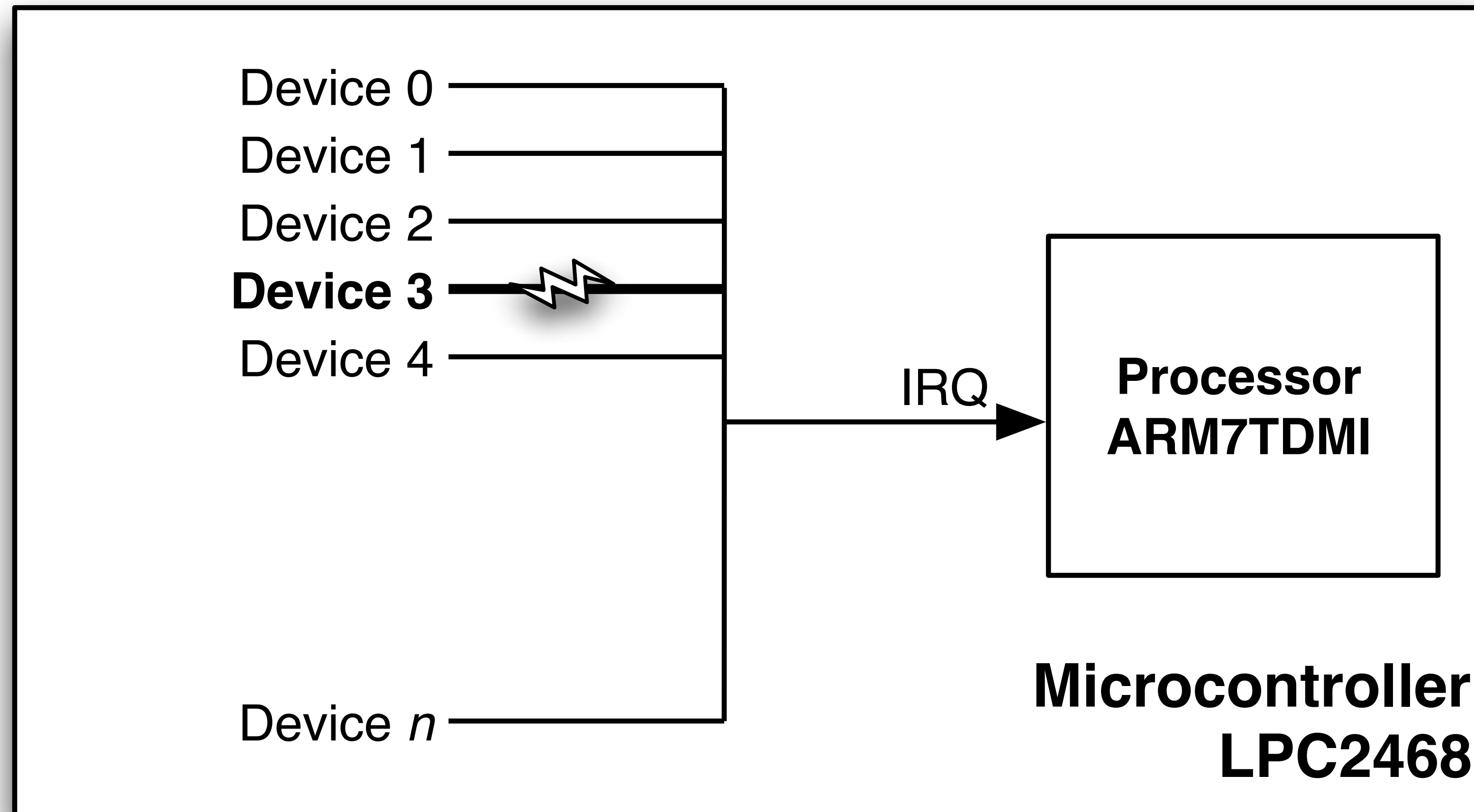
Interrupts are “raised” by devices external to the CPU when events of interest occur

e.g. mouse movement, key press, hardware timer expired

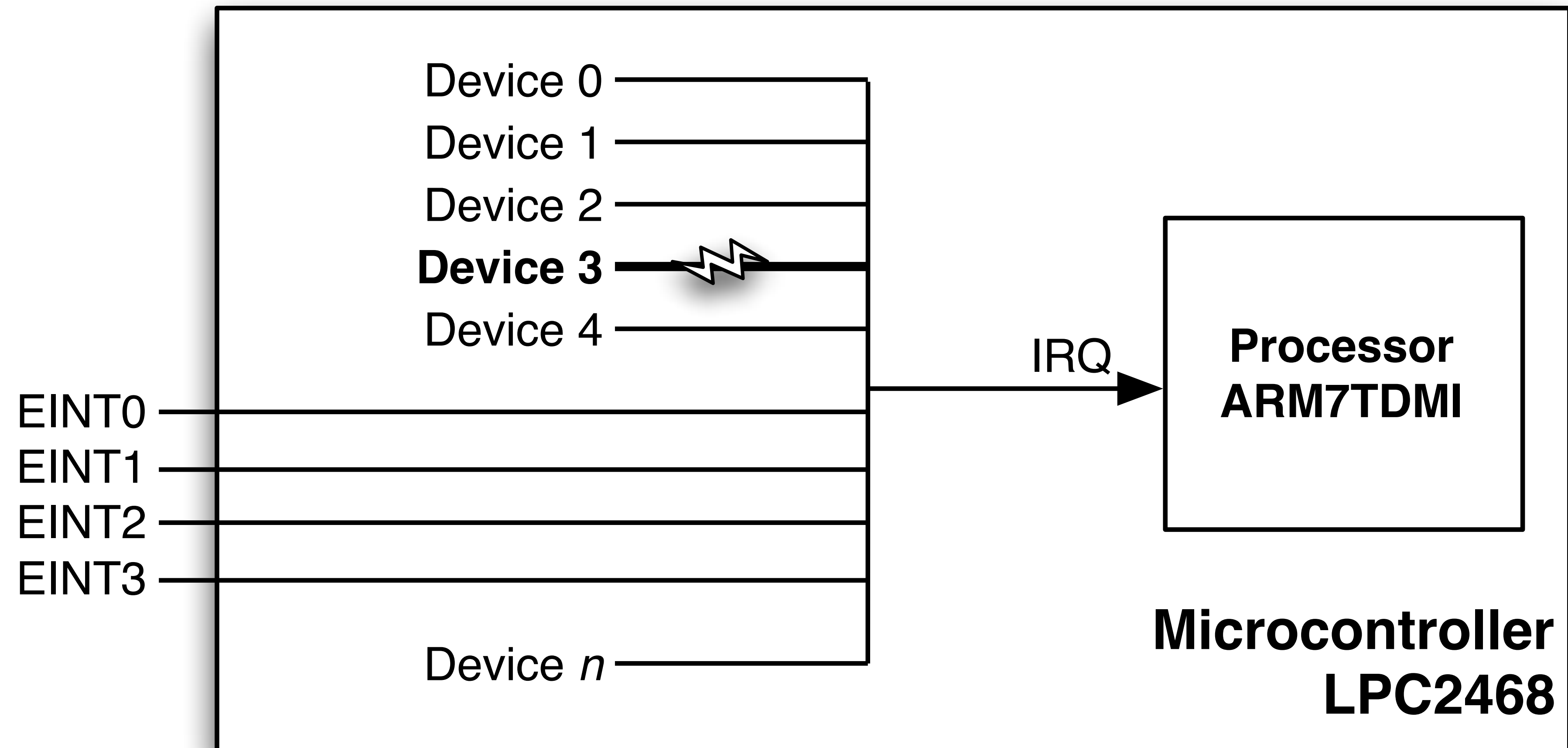
When an IRQ or FIQ occurs, the IRQ or FIQ exception handler is executed

Interrupt could have been “raised” by any one of the devices connected to the CPU

Handler must determine which device raised the interrupt and execute a further handler for that device







How can we determine which device raised the interrupt?

“Ask” each device (e.g. read a memory-mapped status register) whether it has a pending interrupt

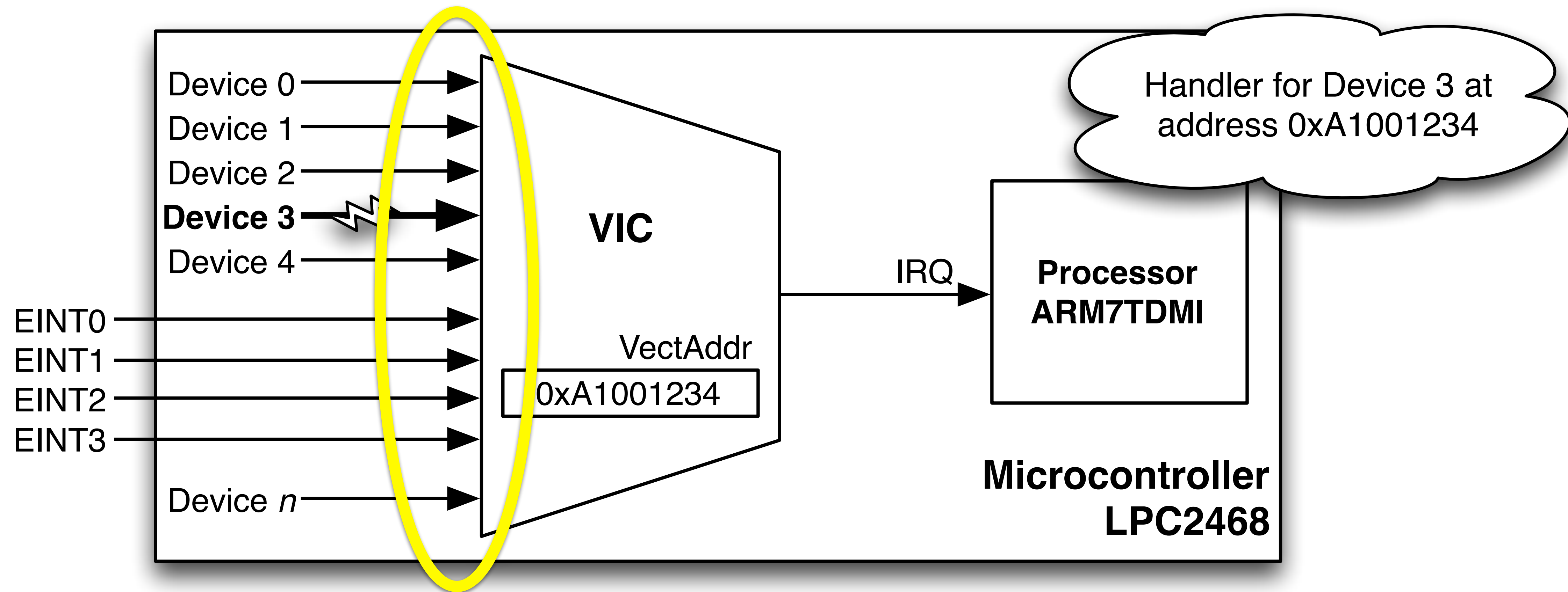
- Takes multiple cycles (multiple LDRs) to determine source of interrupt

- Too slow for IRQs

- Way too slow for FIQs

Get assistance from an external “helper” device

***Vectored Interrupt Controller (“VIC”)***



```
LDR    PC, [address of VIC VectAddr register]
```

```
LDR    PC, [PC, #-0x0120]
```

Up to 32 interrupt sources

Internal (e.g. TIMER, ADC)

External (EINT0/1/2/3, e.g. P2.10 push-button)

Interrupt sources configured independently

When one of the sources raises an interrupt, the VIC ...

loads address of a source-specific handler into VectAddr

raises either the processor's IRQ line or FIQ line

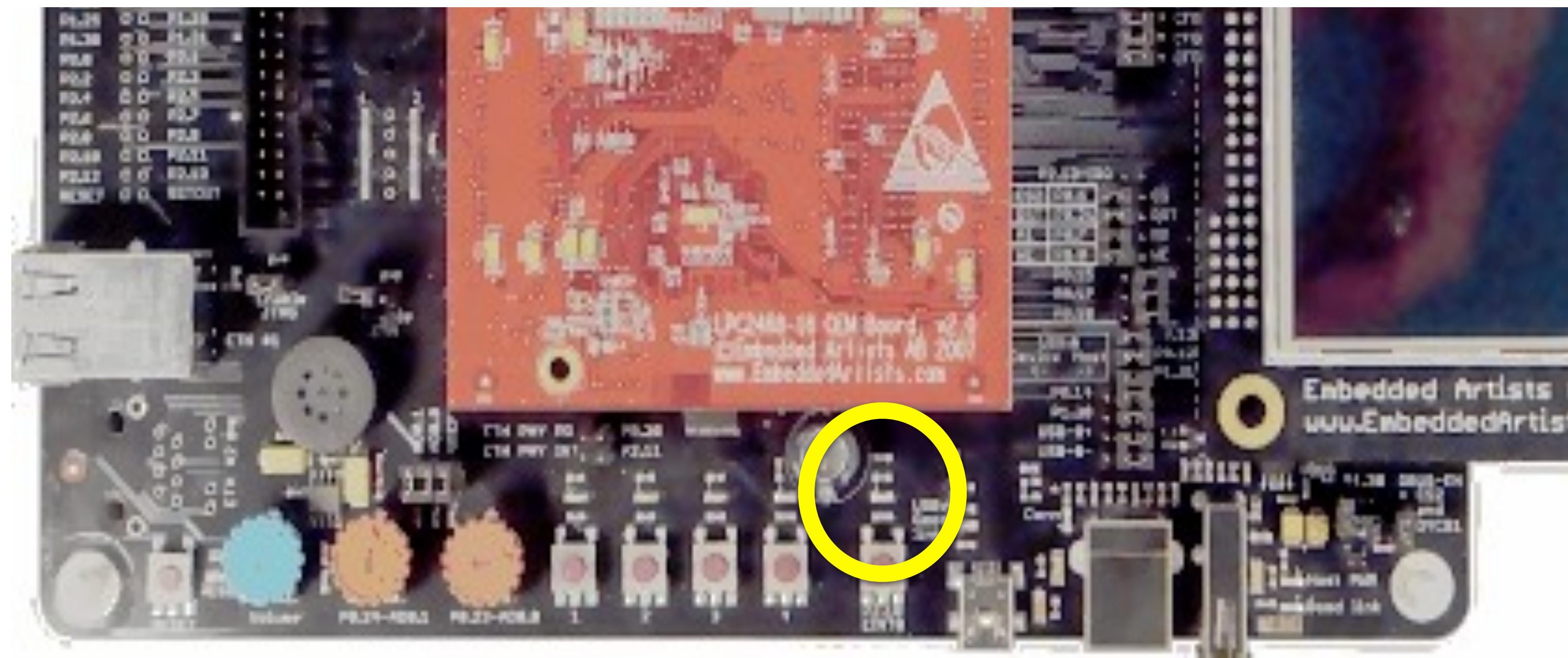
Processor executes source-specific handler loading Program Counter with address in VectAddr register



Design and write a program that will cause an LED to blink with a period of 1s

Use LED connected to pin P2.10 of the LPC2468 (see IO example)

User TIMER0 integrated into LPC2468 to generate an interrupt every 1 second.





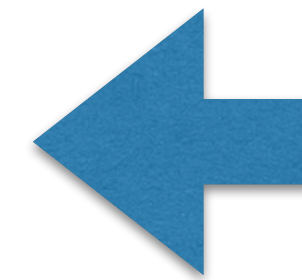
Four integrated timer/counters

Multipurpose

Count input signal transitions (examples?)

Capture time on input signal transition (examples?)

**Measure time interval (counting input clock cycles) (examples?)**



Optionally generate an interrupt when an event occurs (events?)

*See LPC2468 User Manual Chapter 24*

## When program starts ...

Configure pin P2.10 for LED output (PINSEL4 – GPIO, FIO2DIR1 – output)

Stop and Reset TIMER0

Configure TIMER0 for periodic 1 second interrupts

Configure VIC channel for TIMER0 (IRQs or FIQs, handler address, enable channel)

Clear any previous TIMER0 interrupts (e.g. from last time program ran)

Start TIMER0

## When an interrupt occurs ...

Reset TIMER0 interrupt (so the device can raise further interrupts)

Read and invert LED state (FIO2PIN1)

Clear VIC interrupt source (so VIC can raise further interrupts, perhaps from other devices)

Reset and disable TIMER0 (it might already be enabled)

TOTCR [0xE0004004] – Set bit 1 = 1 (reset) and bit 0 = 0 (stop)

Clear any previous TIMER0 interrupt

TOIR [0xE0004000] – Write 0xFF to clear interrupts

Want timer mode, generating interrupts every second

TOCTCR [0xE0004070] – Set bits 1:0 to 00 for timer mode

Want to generate an interrupt (when the count register equals the match register) every 1 seconds

TOMR0 [0xE0004018] = 12,000,000 (assuming TIMER is “fed” with a 12MHz clock)

Want to generate an interrupt, reset and restart the timer when the count register equals the match register

TOMCR [0xE0004014] – Set bits 1:0 to 11

No pre-scaling required (period is sufficiently small)

TOPR [0xE000400C] – Set to 0 for prescale factor of 1



See LPC2468 User Manual!!

Identify the VIC vector number for TIMER0 ... 4

Corresponds to a bit mask 0x10 ( 0001 0000 ) – we will use this later ...

Set vector for IRQ (not FIQ)

VICIntSelect [0xFFFFF00C] – set bit 4 to 0 – BIC using our mask (0x10)

Set priority for vector 4 (e.g. to 15)

VICVectPriority0 (Base Address of VICVectPriority array) = 0xFFFFF200

VICVectPriority4 [0xFFFFF200 + (4 x 4)] – Set to 15

Set handler address

VICVectAddr0 (Base Address of VICVectAddr array) = 0xFFFFF100

VICVectAddr4 [0xFFFFF100 + (4 x 4) ] – Set to address of our TimerHandler routine

Enable vector 4

VICIntEnable [0xFFFFF010] – Write 1 to bit 4 (using our mask (0x10))

Interrupt handler looks like a subroutine, with a few differences

Note the ^ in the LDMFD instruction to restore the SPSR as the CPSR when we return

Note the subtraction of 4 from the LINK register

Note: we save/restore all used registers, including R0-R3

No parameters passed to the subroutine

Interrupt handler must

Clear TIMER0 interrupt – Set T0IR to 0xFF to clear TIMER0 interrupt

Perform action (e.g. inverting LED state)

Clear VIC (by writing 0 to VICVectAddr)

Return