19 Interrupts
CSC 230

Department of Computer Science
University of Victoria

M&H: 8.3.1, 8.3.2, 5.3.3, 4.2.2

Stallings: 3.2, 7.3, 7.4, 14.2, 14.6 (also check index)

What is an Interrupt?

An interrupt is an ***unscheduled*** event that requires the CPU to stop normal program execution and perform some service related to the event

External (to CPU)

external circuitry generates an event (e.g. a peripheral)

➔ Called an INTERRUPT

Internal (to CPU)

software errors or from internal circuitry (e.g. division by zero)

➔Called an EXCEPTION

Applications:

- ✓coordination of I/O activities leaving CPU free
- ✓graceful way to exit from software error
- ✓multitasking (signal end of time slot)
- ✓timer interrupt to indicate delays
- ✓page faults or paging

Typical sources for interrupts or exceptions

Typical interrupts:

- a timer event
- a character from an input device
- a button push
- a sensor signal
- an external device completes an operation

externals

Typical exceptions:

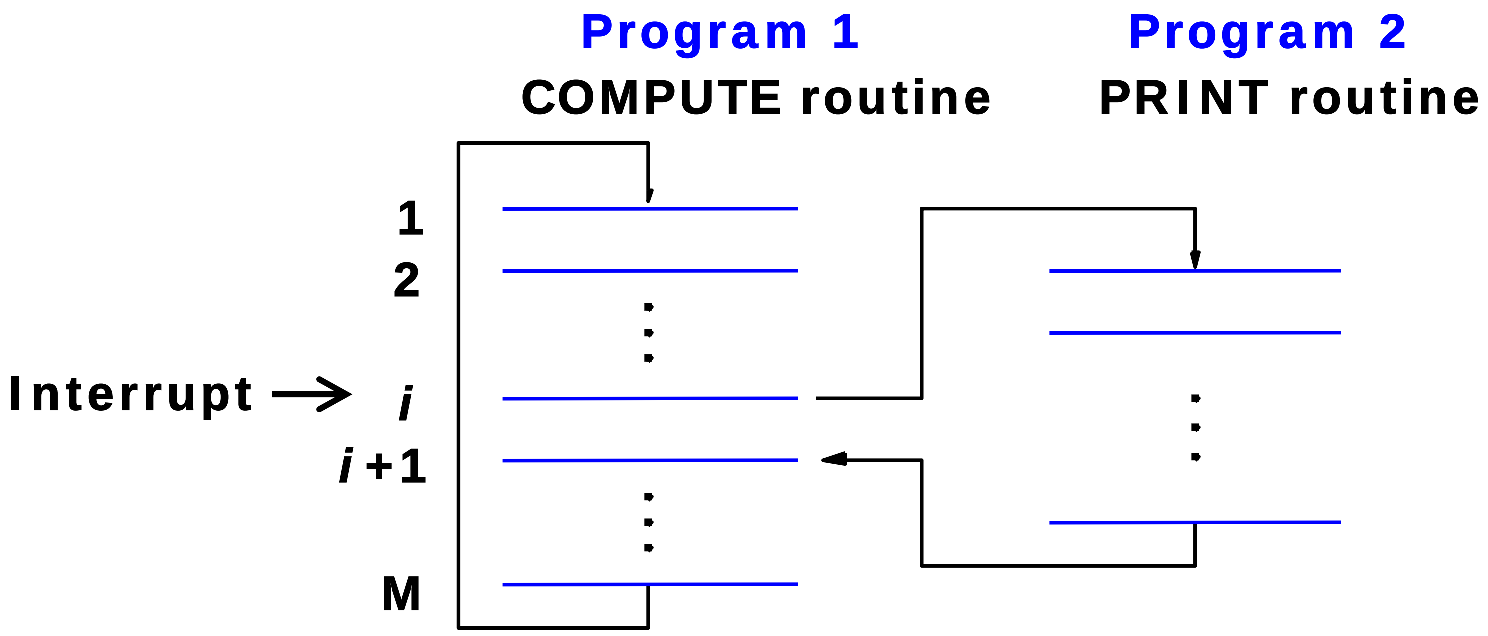
- an ALU exception e.g. divide by zero, overflow
- an addressing fault

internals

An interrupt is an event *always outside the program* which requires the program to halt what it is doing, take appropriate action, and then *(usually)* return to the point where it was interrupted.

Seems similar to a subroutine call!

Conceptually, an interrupt forces
a *special type of jump*
to a *special kind of subroutine*
at an *indeterminate point* in the program



Transfer of control through the use of interrupts.

Some Differences between a Subroutine Call and an Interrupt

A1. Subroutine performs tasks required by control flow of program

B1. Subroutine always returns to where it was called from and program resumes execution

A2. Interrupt is not scheduled by the program, may even be caused by or related to other processes/users *(outside the program)*

B2. After an interrupt, execution may:

i. Return where it left off (as in a subroutine call);

i. Return to a different place in the application program (e.g. like a ‘catch’ block in Java)

ii. Exit from the application and return to the OS;

iii. Exit from the OS and shut down (the ‘blue screen of death’?).

Some basic concepts → read more in textbook

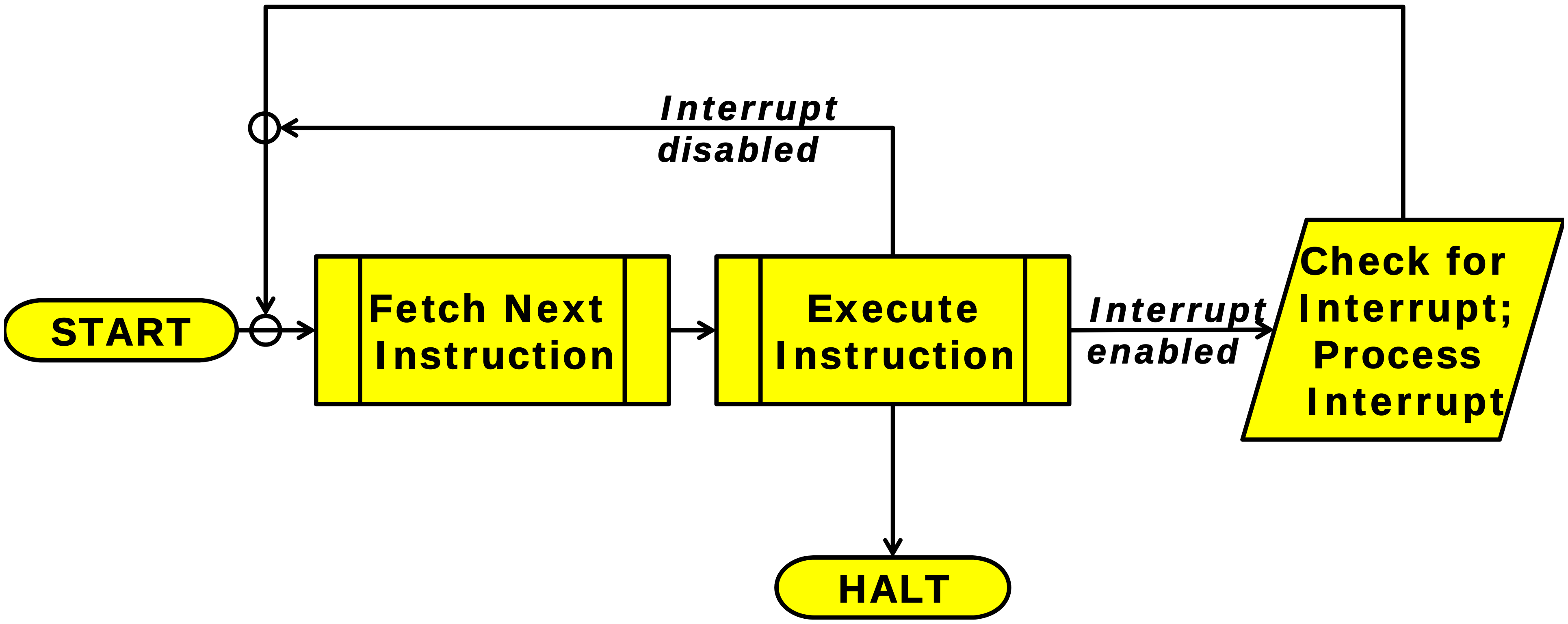
- ❑ Interrupt line
- ❑ Interrupt service routine (ISR)
  - a) General
  - b) for each possibility / device
- ❑ Interrupt acknowledge signal (IACK)
- ❑ Return from interrupt
- ❑ Context switching
- ❑ Interrupt latency
- ❑ Saving states
- ❑ OS intervention

  → more than just a mechanism for I/O transfers

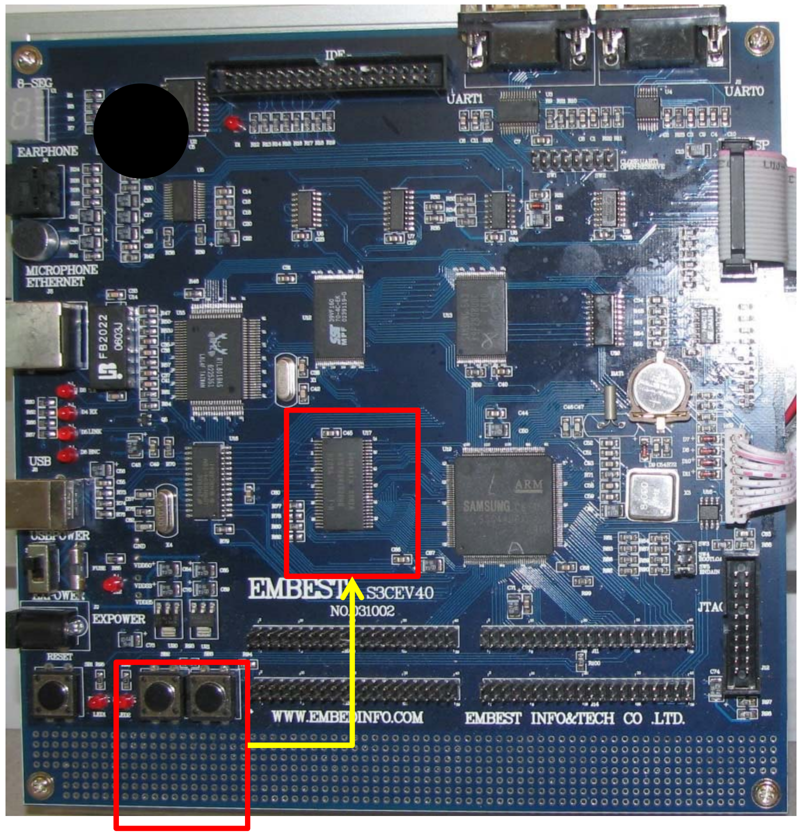
  → real-time control and processing

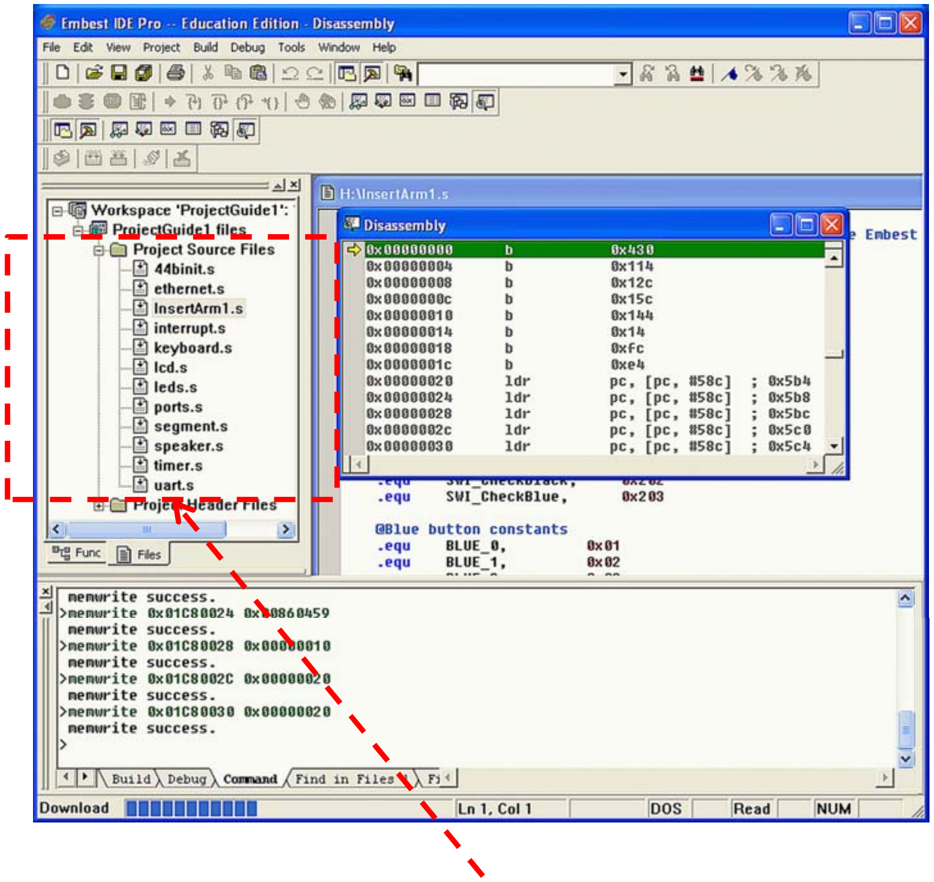
Putting it all together: Instruction Cycle with Interrupts

FETCH CYCLE EXECUTE CYCLE INTERRUPT CYCLE



What happens in the project with SWI, the buttons, etc?




Embest IDE Pro -- Education Edition - Disassembly
File Edit View Project Build Debug Tools Window Help
Func
Files
H:\InsertArm1.s
Disassembly
Build
Debug
Command
Find in Files
Download
Ln 1, Col 1
DOS
Read
NUM

```
Workspace 'ProjectGuide1':
  ProjectGuide1 files
    Project Source Files
      44binit.s
      ethernet.s
      InsertArm1.s
      interrupt.s
      keyboard.s
      lcd.s
      leds.s
      ports.s
      segment.s
      speaker.s
      timer.s
      uart.s
    Project Header Files
```

```
0x00000000      b       0x430
0x00000004      b       0x114
0x00000008      b       0x12c
0x0000000c      b       0x15c
0x00000010      b       0x144
0x00000014      b       0x14
0x00000018      b       0xfc
0x0000001c      b       0xe4
0x00000020      ldr     pc, [pc, #58c]  ; 0x5b4
0x00000024      ldr     pc, [pc, #58c]  ; 0x5b8
0x00000028      ldr     pc, [pc, #58c]  ; 0x5bc
0x0000002c      ldr     pc, [pc, #58c]  ; 0x5c0
0x00000030      ldr     pc, [pc, #58c]  ; 0x5c4
```

```
.equ    SWI_CheckBlue,      0x203

@Blue button constants
.equ    BLUE_0,         0x01
.equ    BLUE_1,         0x02
```

```
 memwrite success.
>memwrite 0x01C80024 0x00860459
 memwrite success.
>memwrite 0x01C80028 0x00000010
 memwrite success.
>memwrite 0x01C8002C 0x00000020
 memwrite success.
>memwrite 0x01C80030 0x00000020
 memwrite success.
>
```

(1) Button press ➔ bit stored in board SDRAM

➔ Aynchronously and independently of any user code

➔ Done by built-in project code provided

➔ In between instructions as capture of true interrupt signal

What happens in the project with SWI, the buttons, etc?

- ➔ Bit remains there until an appropriate SWI instruction issued by user code “polls” it
- ➔ Really fast polling basically is similar to capturing an interrupt
- ➔ In assignment 3, you are doing polling

A basic sample sequence: 1 interrupt from 1 device

(1) Device asserts “Interrupt” with a signal on an interrupt line

(2) CPU stops computing and enters a *general* Interrupt Service Routine (ISR)

(3) CPU disables Interrupts bits (using bits in Status Register)

(4) CPU sends an acknowledgement to the device

(5) Device resets the interrupt line signal

(6) CPU services “Interrupt” with *appropriate* Interrupt Service Routine

(7) CPU re-enables Interrupts bits

(8) CPU resumes computing

More ideas / concepts / questions / issues

- Many devices can generate an Interrupt
  - ➔ how to detect which one?
  - Some interrupt sources can be enabled or disabled (e.g. I/O)
  - Other interrupt sources are always enabled (e.g. power)

- An interrupt handler must be written for each type of interrupt
  - Code must be short and to the point

- A program must do the necessary initialization for handling each class of interrupts before enabling the corresponding interrupt source

More ideas / concepts / questions / issues

- ❑ **Where does one find the address of the correct ISR (Interrupt Service Routine)?**

- ❑ **What about the situation of an interrupt occurring while another interrupt handler is executing i.e. nested interrupts?**
  - ➔ **Interrupt priority - fixed or programmable**

How does the CPU know which Interrupt it is?

When an Interrupt line (communicating to the processor) is set because there is an interrupt request, “polling” to see who needs servicing is not very efficient

1. **When an Interrupt signal arrives on an Interrupt line it implies:**

   ➔ a service is requested

2. **To implement the Identification of who requested the service there are choices:**

   a) **special code has been sent on data bus by the device, often together with the interrupt signal**

      ➔ this is usually for small systems only or embedded system

   b) **Vectored interrupts**

What are Interrupt Vectors?

Interrupt vector = starting address of appropriate ISR for every device allowed to interrupt (or class of devices)

Interrupt table → it contains all interrupt vectors

- Interrupt table is located at a dedicated position in the address space, often in ROM

Sample process:

CPU knows the address of the Interrupt Table

CPU receives some “number”

→number is an index into the table

→ index indicates which entry in the table

→ entry in table contains the address of appropriate ISR

→ PC is loaded with that address

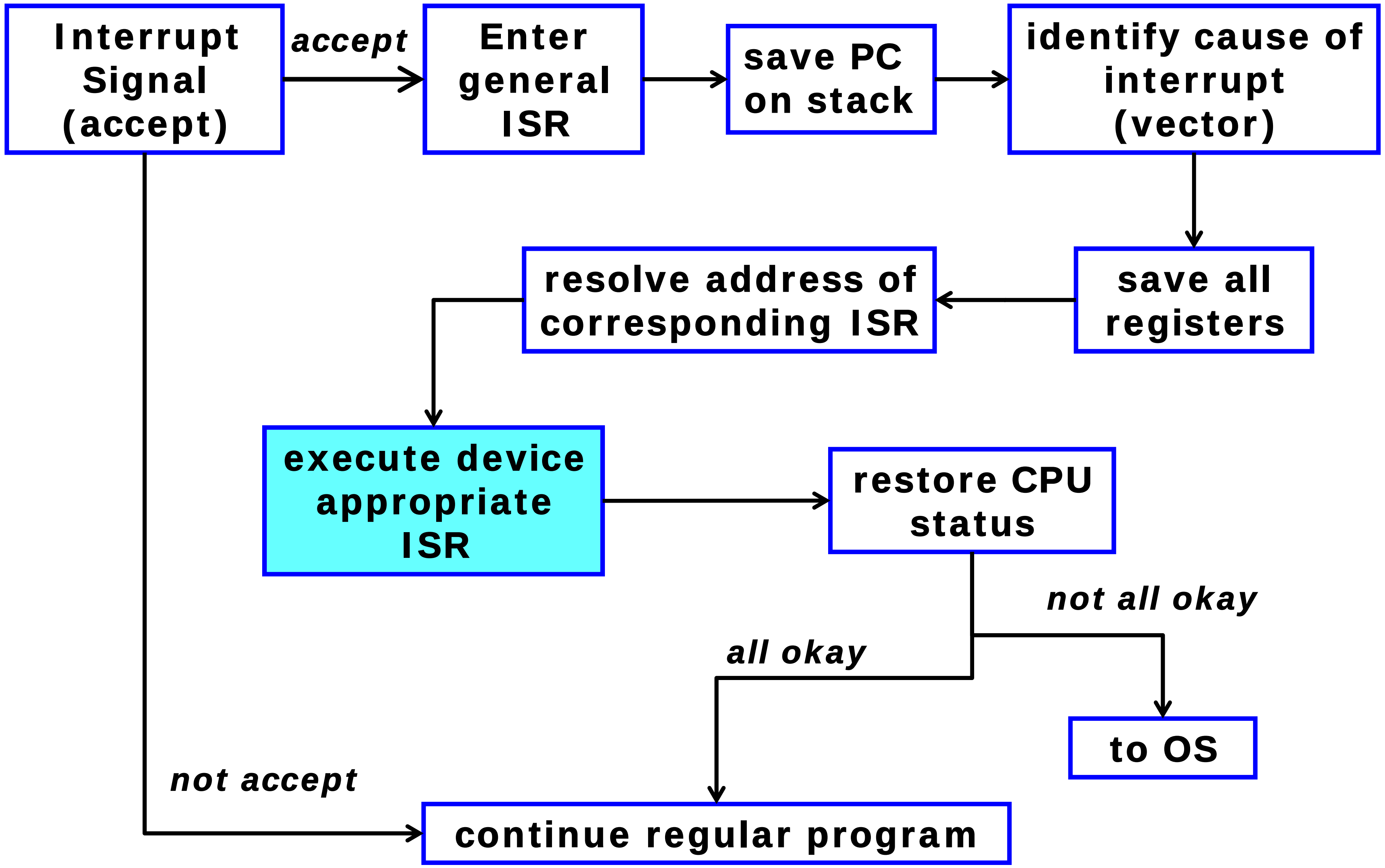
→ISR starts execution

ARM Interrupt table (subset)

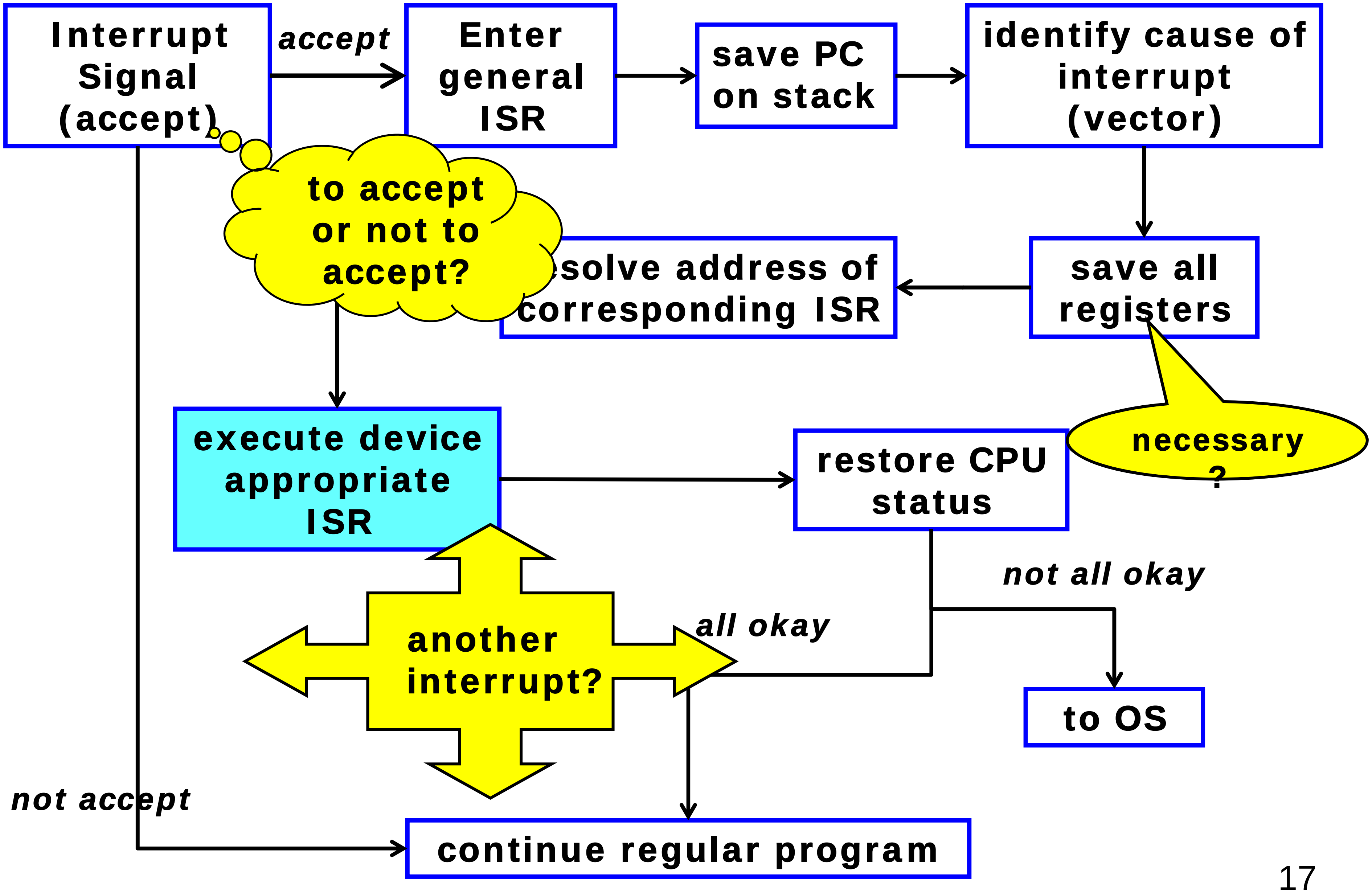
Exception / Interrupt	Mode	Vector Address
Reset	SVC	0x00000000
Undefined Instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Mode	Use
User	Normal user mode
FIQ	Processing fast interrupts
IRQ	Processing standard interrupts
SVC	Processing software interrupts (SWI)
Abort	Processing memory faults
Undef	Handling undefined instruction traps
System	Running privileged OS tasks

The Interrupt Process



The Interrupt Process



To Accept or Not: Interrupt Priority Level

Interrupt priority is used to decide

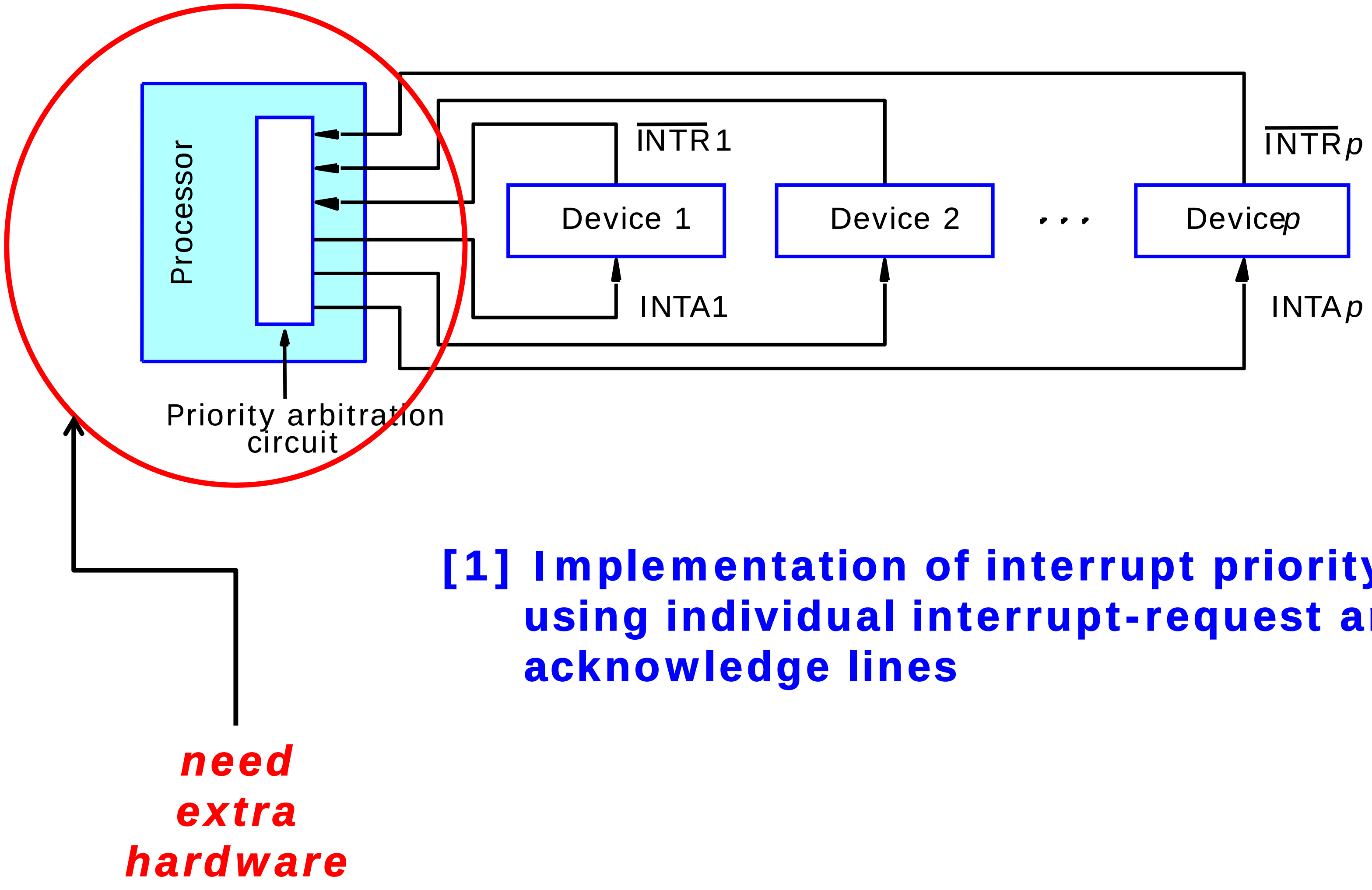
- if an interrupt should be serviced,
- or for nested interrupts

IPL = Interrupt Priority Level

Maskable interrupts: those that can be ignored

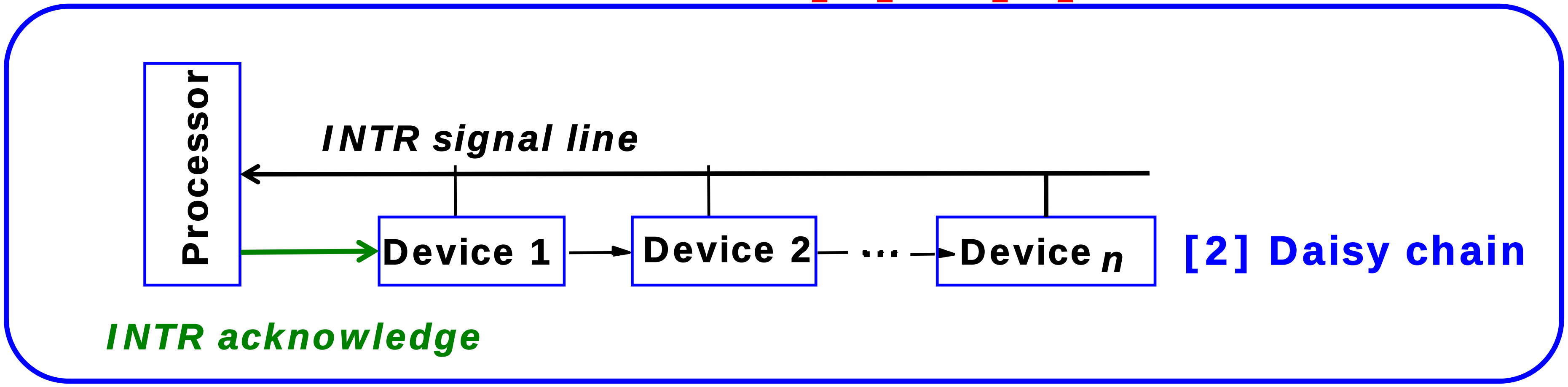
- it is possible to set a maskable bit in the status register to enable or disable interrupts
- the maskable bits state the level above which an interrupt will be considered
  - if current level = 4, only interrupts with priority 5 and above will be serviced

How to implement Interrupt Priority Levels with Hardware [1]

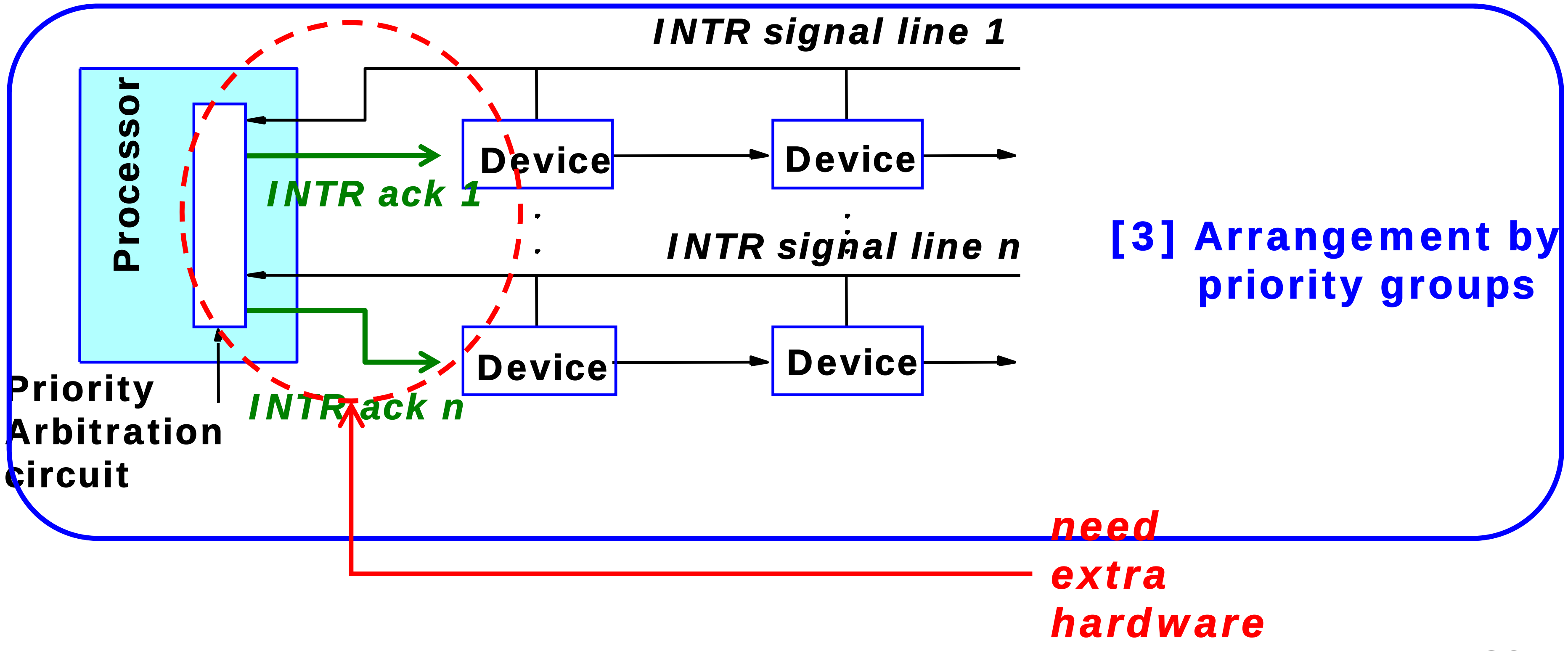

Processor
Priority arbitration circuit
$\overline{\text{INTR}}1$
$\overline{\text{INTR}}p$
Device 1
Device 2
Device p
INTA1
INTA p
need extra hardware

[1] Implementation of interrupt priority using individual interrupt-request and acknowledge lines

How to implement Interrupt Priority Levels with Hardware [2] & [3]



[2] Daisy chain

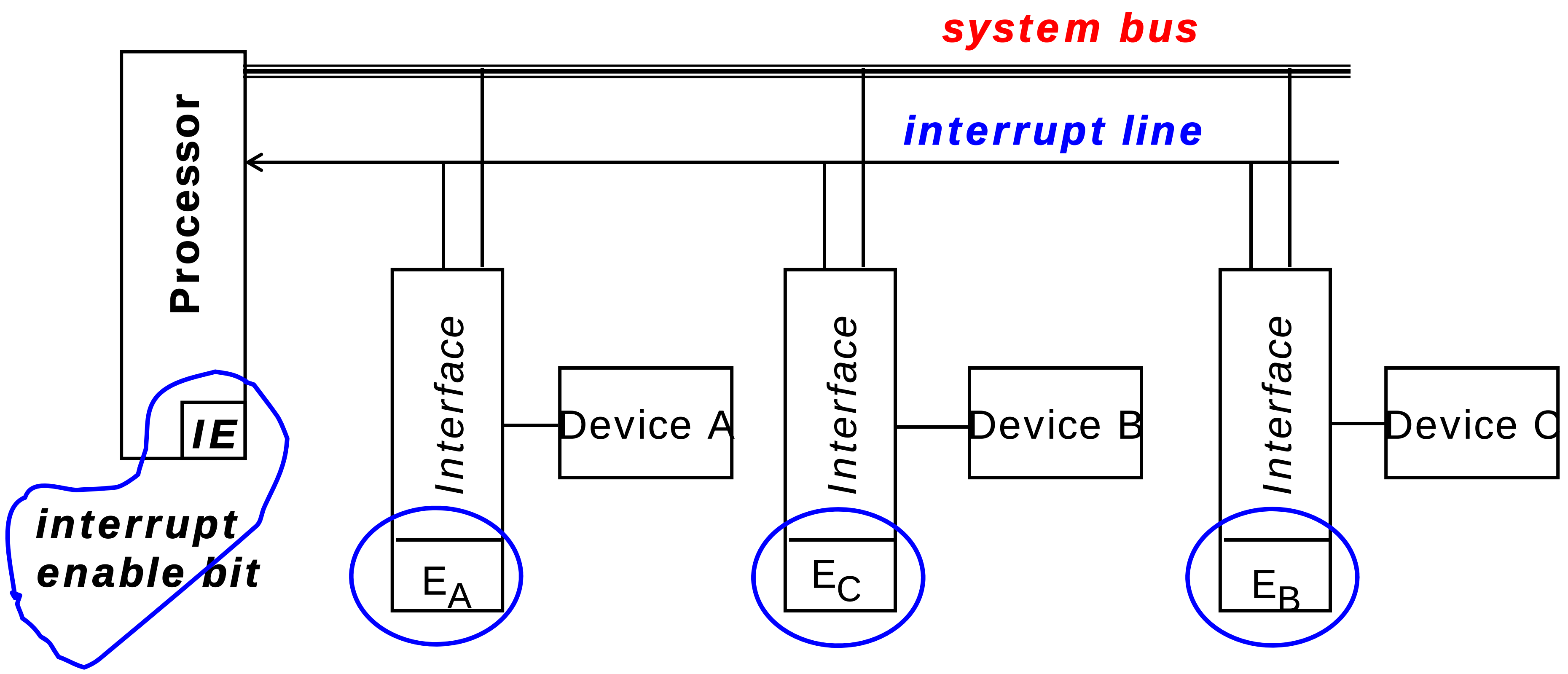


[3] Arrangement by priority groups

Interrupt Enable / Disable Bits

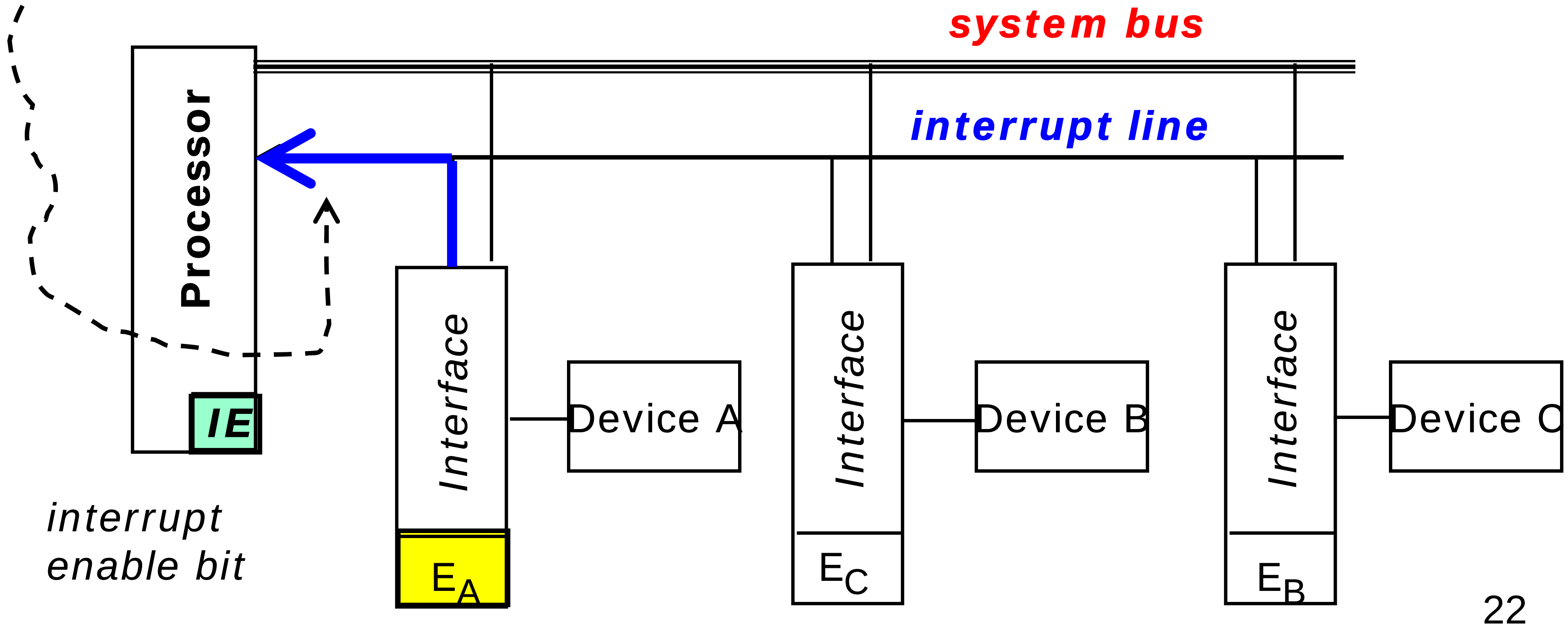
There are bits *both* in the processors and in the device interfaces to show the current "status" for interrupts

- ➔**in CPSR of processor (set / disabled by processor)**
- ➔ **in interfaces**



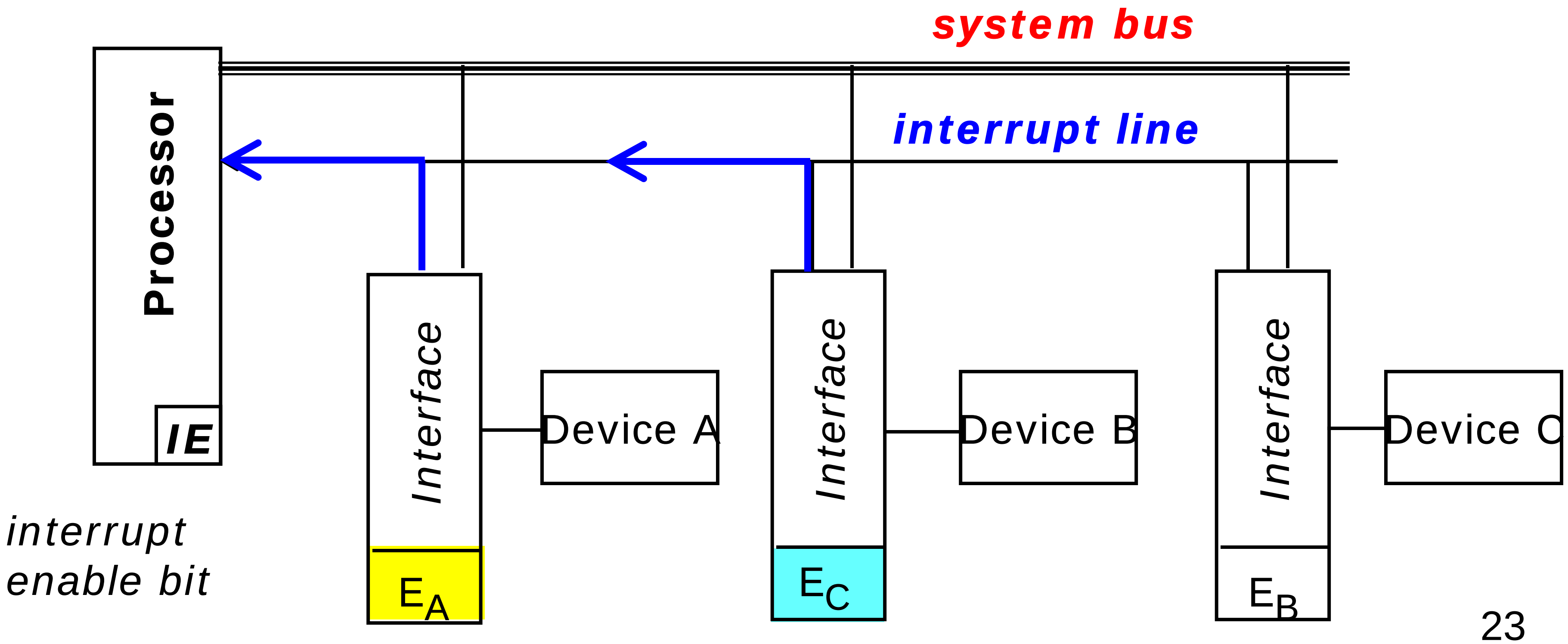
Interrupt Enable / Disable with 1 bit: example 1

1. **Device A sends interrupt signal on Interrupt line and sets EA to 1 to show that it is busy requesting interrupt**
2. **Processor accepts interrupt, sets IE to 0 (no more interrupts)**
3. **Device A sets EA to 0**
4. **If another device now sends a signal, it will be ignored**



Interrupt Enable: example 2

1. **Device A sends interrupt signal on Int. line and sets EA to 1**
2. **Processor accepts interrupt and leaves IE to 1 – others can still interrupt (perhaps with higher priority)**
3. **Device A sets EA to 0**
4. **Device C: sends interrupt signal on Int. line and sets EC to 1 – this could interrupt the servicing of A since IE is still 1**
5. **This could continue until IE is set to 0 and more interrupts are ignored**



Priority Levels in ARM

1. **Reset**
2. **Data abort**
3. **FIQ (with 1 possible source)**
4. **IRQ (with 32 possible sources)**
5. **Prefetch abort**
6. **SWI**

If more than one arrives at the same time, ...

➔ ARM Reference Peripheral Specification

Exceptions

Type of event	Where from?	Terminology
I/O device request	External	Interrupt
Invoke the OS from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunction	Either	Exception or Interrupt

In ARM, 3 major groups:

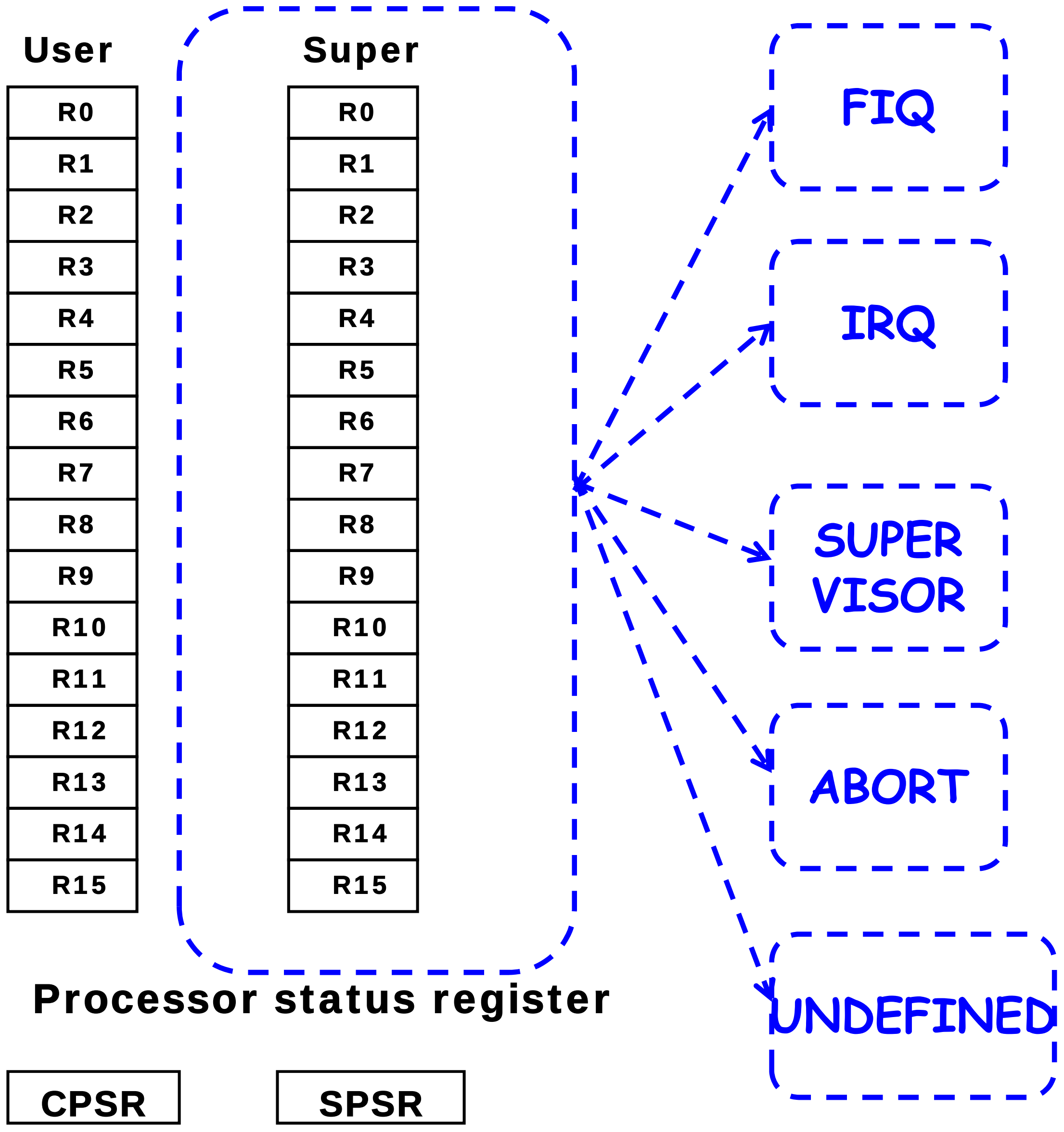
1. Exceptions generated as the direct effect of executing an instruction

   e.g. SWI, undefined instructions

2. Exceptions generated as a side-effect of an instruction

   e.g. data aborts

3. Exceptions generated externally, unrelated to the instruction flow (interrupts proper)



General-purpose registers and banked registers

Different registers are accessible in different modes of the ARM processor.

User	FIQ	IRQ	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13-svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15	R15	R15	R15	R15	R15

General-purpose registers and program counter

FIQ has more banked registers to save time on context switching

Processor status register

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

Accessible registers in different modes of the ARM processor.

Switching modes

➔ switching some registers ➔ to banked registers

e.g. R13 ➔ R13_irq

Exception-handling routines start at fixed memory addresses

see interrupt vector table

in this slot, there is a Branch to appropriate service routine (except for FIQ) – usually in ROM

1. Save return address in “banked R14”
2. Save CPSR in SPSR
3. Change bits in CPSR (for mode, for disabling)
4. Branch to correct service routine - no stack use for efficiency

Interrupt Latency (ARM example)

1. Time for request signal to pass through the FIQ synchronization → 3 clock cycles

2. Time for the longest instruction to complete (LDM 16 registers) → 20 clock cycles

3. Time for data abort entry sequence → 3 clock cycles

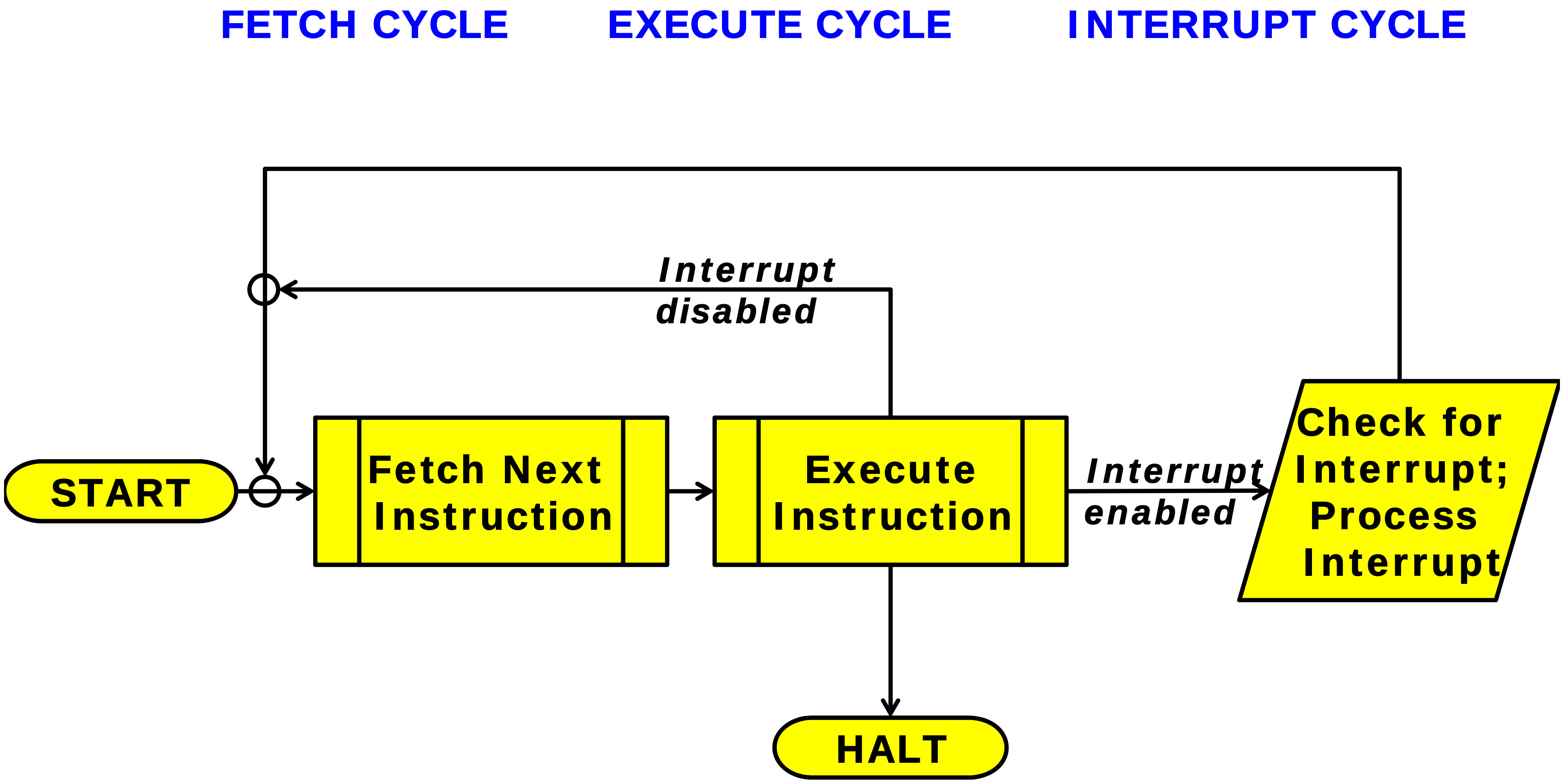
4. Time for FIQ entry sequence → 2 clock cycles

With cache and MMU involvement, it can get worse → 87 clock cycles!

Interrupts and exceptions

- Debuggers
  - traces, breakpoints, stepping through
- Privileged Instructions
- OS uses ISR for all I/O access by user programs
- OS uses interrupts to change control from one program/user to another
- OS is mostly still in user mode, goes into supervisor mode when needed
- Multitasking with time slicing

Putting it all together: Instruction Cycle with Interrupts



Why, on ARM, is user-level code prohibited from disabling interrupts?

It would make a protected Operating System impossible!

Look at this possible (malicious) user code:

```
        MSR    CPSR_f,#0xC0        ;disable IRQ and FIQ
LOOP:   BAL    LOOP                ;loop forever
```

Once interrupts are disabled, OS cannot regain control

Program will loop forever, can only be stopped by a hard reset (which destroys all loaded programs)

Advanced