EC 21-22
Computer Architecture
Fall 2021

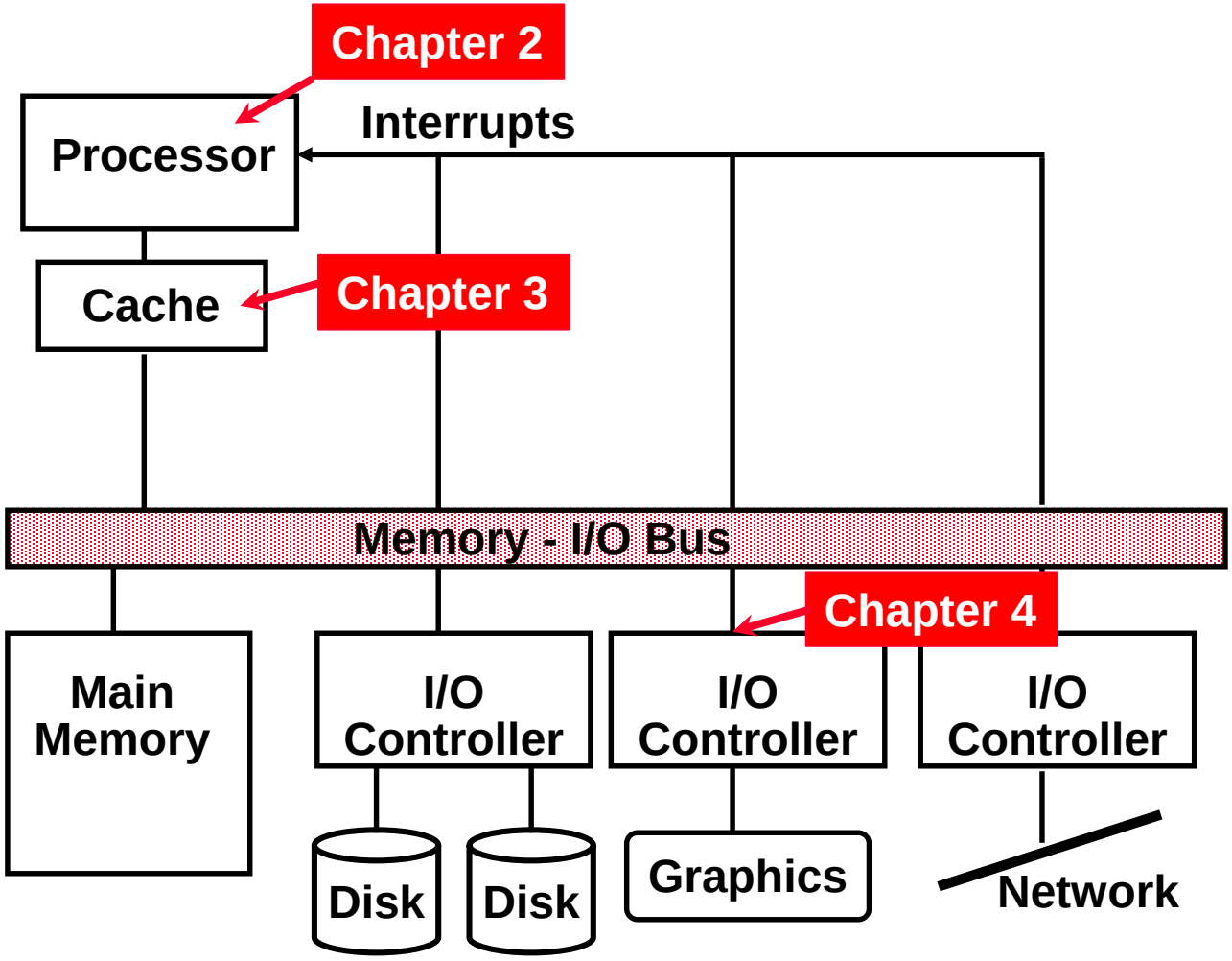
Chapter 4: Input/Output System

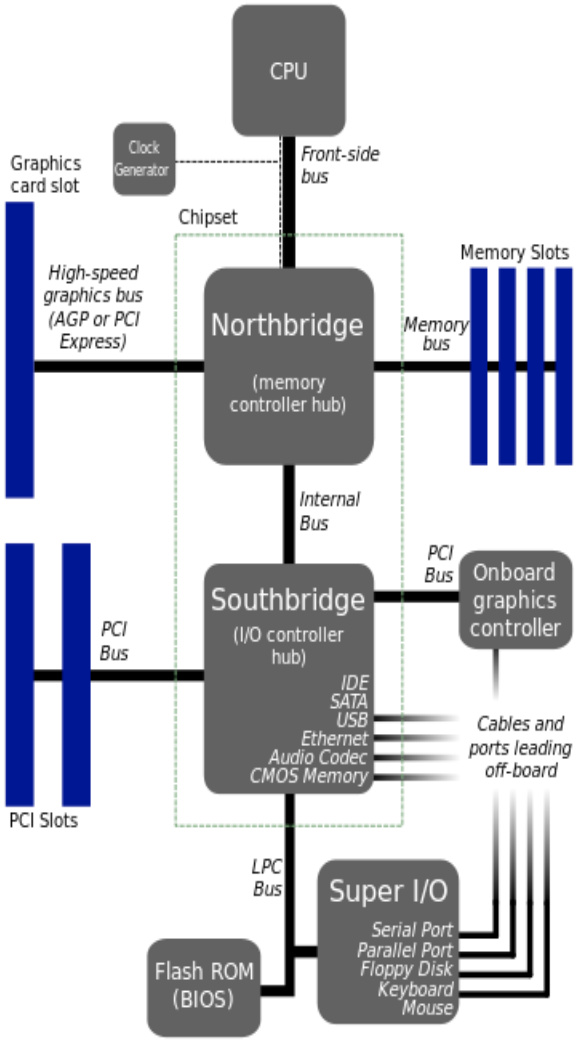
Mª Julio Villalba Moreno

www.ac.uma.es/~julio

julio@ac.uma.es

A Typical I/O System

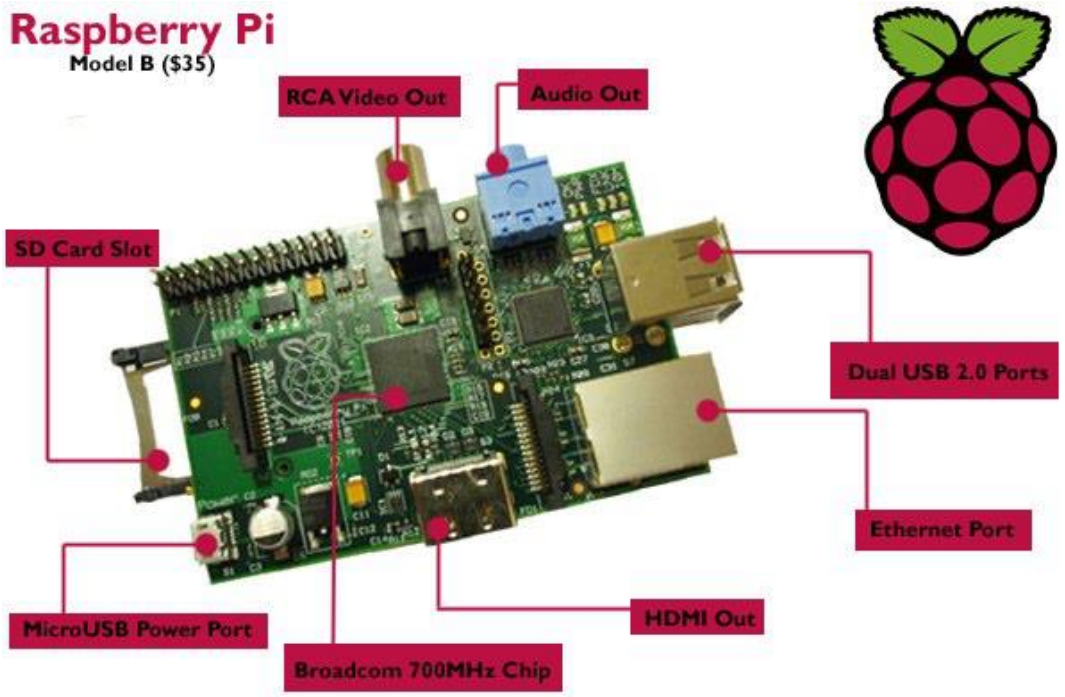




Raspberry Pi

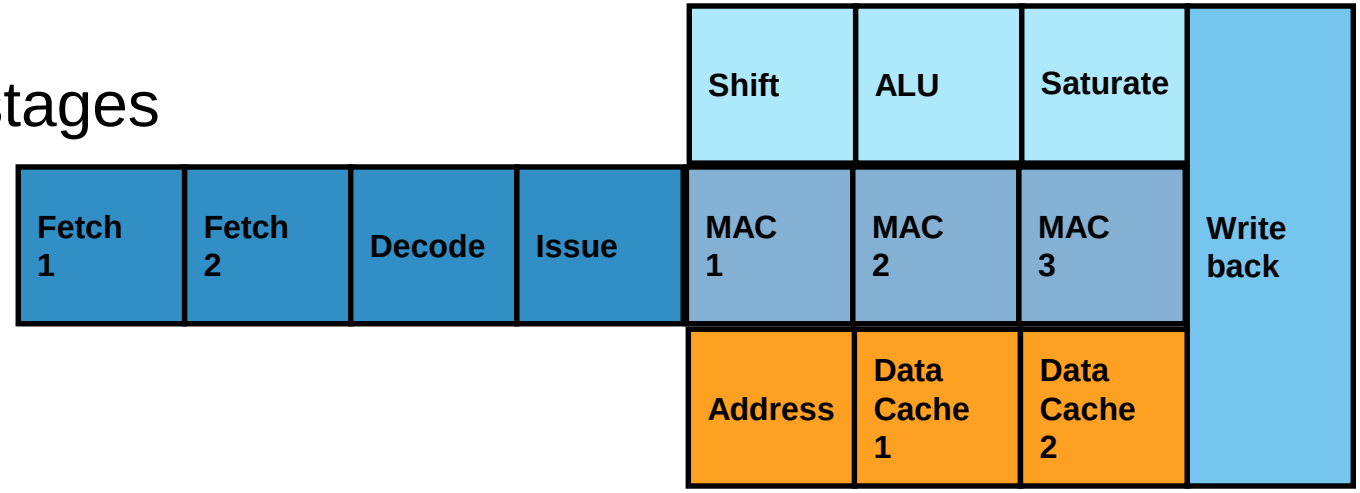
Raspberry Pi

- Based on ARM11 (ARMv6) Broadcom BCM2835/BCM2836.
- Low Power ARM1176JZ-F Applications Processor
- ARM11 is the processor integrated in the first iPhone (also in iPhone 3G).



ARM11

- 8 stages



Data Sizes and Instruction Sets

- The ARM is a 32-bit, Load-Store RISC architecture.

- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

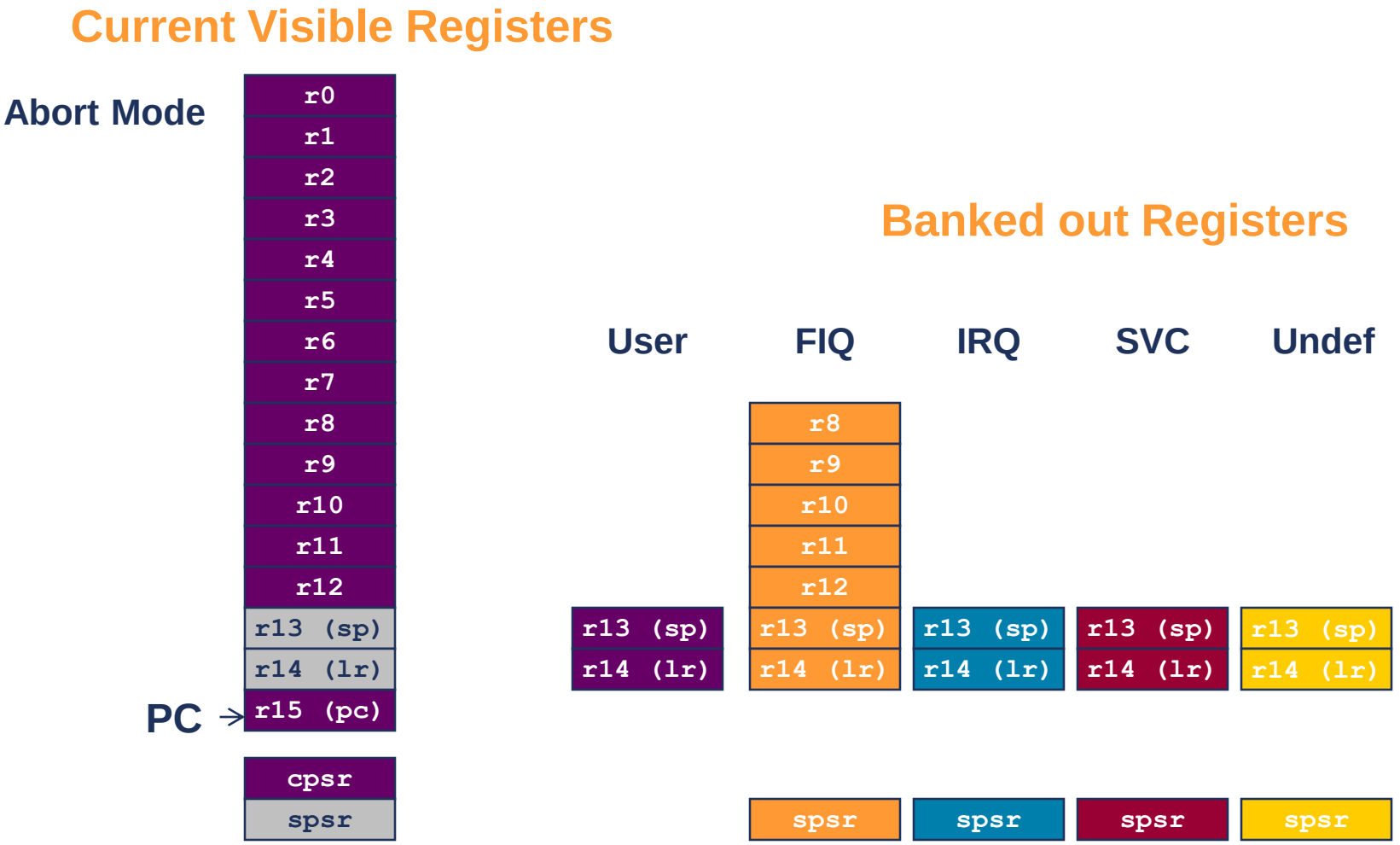
- Most ARM's implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- Jazelle cores can also execute Java bytecode (8-bit)

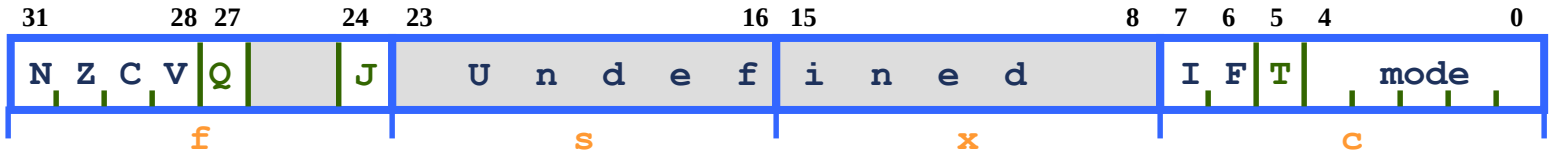
Processor Modes

- The ARM has seven basic operating modes, used to run user tasks, an operating system, and to efficiently handle exceptions such as interrupts:
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on start up or reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations as a result of fetching instructions or accessing data.
  - **Undef** : used to handle unknown or illegal instructions
  - **System** : privileged mode using the same registers as user mode

The ARM Register Set



Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed (the result is not representable in 32 bits (C2))

- **Sticky Overflow flag - Q flag**
  - Architecture 5TE/J only
  - Indicates if saturation has occurred

- **J bit**
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state

- **Interrupt Disable bits.**
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.

- **T Bit**
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state

- **Mode bits**
  - Specify the processor mode

Program Counter (r15)

- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)

- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned)

- **When the processor is executing in Jazelle state:**
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Conditional Execution (predicated inst.)

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions. Ej: **if r3≠0 then r8=r1+r2**

```
   CMP   r3,#0
   BEQ   skip
   ADD   r8,r1,r2
skip
```

```
CMP   r3,#0
ADDNE r8,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using ***"S"***. CMP does not need "S".

```
loop
   …
   SUBS r1,r1,#1
   BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

  - This loop never ends is we use sub r1,r1,#1

Condition Codes

Mnemonic {cond}	Meaning
EQ	(equal) When **Z** is enabled (**Z is 1**)
NE	(not equal). When **Z** is disabled. (**Z is 0**)
GE	(greater or equal than, in two's complement). When both **V** and **N** are enabled or disabled (**V is N**)
LT	(lower than, in two's complement). This is the opposite of GE, so when **V** and **N** are not both enabled or disabled (**V is not N**)
GT	(greather than, in two's complement). When Z is disabled and **N** and **V** are both enabled or disabled (**Z is 0, N is V**)
LE	(lower or equal than, in two's complement). When **Z** is enabled or if not that, **N** and **V** are both enabled or disabled (**Z is 1. If Z is not 1 then N is V**)
MI	(minus/negative) When **N** is enabled (**N is 1**)
PL	(plus/positive or zero) When **N** is disabled (**N is 0**)
VS	(overflow set) When **V** is enabled (**V is 1**)
VC	(overflow clear) When **V** is disabled (**V is 0**)
HI	(higher) When **C** is enabled and Z is disabled (**C is 1 and Z is 0**)
LS	(lower or same) When **C** is disabled or **Z** is enabled (**C is 0 or Z is 1**) CS/HS (carry set/higher or same) When **C** is enabled (**C is 1**)
CS/HS	(carry set/higher or same) When **C** is enabled (**C is 1**)
CC/LO	(carry clear/lower) When **C** is disabled (**C is 0**)

Update status register	cmp inst{s}: adds, subs, ands, ...
Brach (b) cond.	b{cond}: b**eq**, b**ne**, b**gt**, b**le** ...
Instruction cond.	inst{cond}: add**eq**, sub**ne**, ldr**gt**, ...

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 3 instructions
- 3 words
- 3 cycles

Data Processing Instructions

- Consist of :
  - Arithmetic: `ADD` `ADC` `SUB` `SBC` `RSB` `RSC`
  - Logical: `AND` `ORR` `EOR` `BIC`
  - Comparisons: `CMP` `CMN` `TST` `TEQ`
  - Data movement: `MOV` `MVN`
    - *TST is used to test a bit inside a register*, tst R1,#0b00100

      *Operation: R1**.AND.**0b00100 (no register afected, Z affected)*

- These instructions only work on registers, NOT memory.

- Syntax:

  `<Operation>{<cond>}{S} Rd, Rn, Operand2`

  - Comparison instructions set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.

Loading 32 bit constants

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR rd, =const`
- This will either:
  - Produce a **`MOV`** or **`MVN`** instruction to generate the value (if possible).

  or
  - Generate a **`LDR`** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- For example
  - `LDR r0,=0xFF` => `MOV r0,#0xFF`
  - `LDR r0,=0x55555555` => `LDR r0,[PC,#Imm12]`

```
...
...
DCD 0x55555555
```

- This is the recommended way of loading constants into a register

Single register data transfer

`LDR`	`STR`	Word
`LDRB`	`STRB`	Byte
`LDRH`	`STRH`	Halfword
`LDRSB`		Signed byte load
`LDRSH`		Signed halfword load

- Memory system must support all access sizes

- Syntax:
  - `LDR`{<cond>}{<size>} Rd, <address>
  - `STR`{<cond>}{<size>} Rd, <address>

e.g. `LDREQB`

Address accessed

- Address accessed by LDR/STR is specified by a base register with an offset
- For word and unsigned byte accesses, offset can be:
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)

    `LDR r0, [r1, #8]` → ***r0=m(r1+8)***
  - A register, optionally shifted by an immediate value

    `LDR r0, [r1, r2]` → ***r0=m(r1+r2)***

    `LDR r0, [r1, r2, LSL#2]` → ***r0=m(r1+(r2<<2))***
- This can be either added or subtracted from the base register:

  `LDR r0, [r1, #-8]` → ***r0=m(r1-8)***

  `LDR r0, [r1, -r2, LSL#2]` → ***r0=m(r1-(r2<<2))***

Load/Store Exercise

Assume an array of 25 words. A compiler associates y with r1. Assume that the base address for the array is located in r2. Translate this C statement/assignment using just three instructions:

```
array[10] = array[5] + y;
```

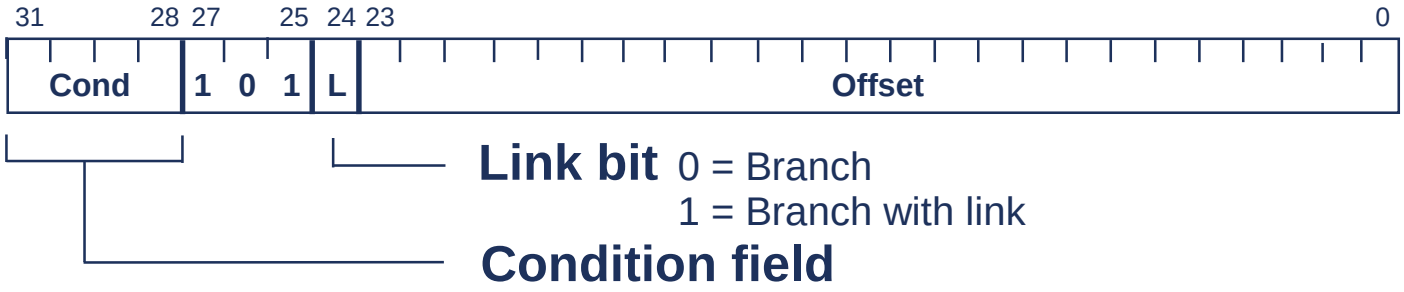
Load/Store Exercise Solution

```
array[10] = array[5] + y;
```

```
LDR     r3, [r2, #5]   ; r3 = array[5]
ADD     r3, r3, r1      ; r3 = array[5] + y
STR     r3, [r2, #10]  ; array[5] + y =
array[10]
```

Branch instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`

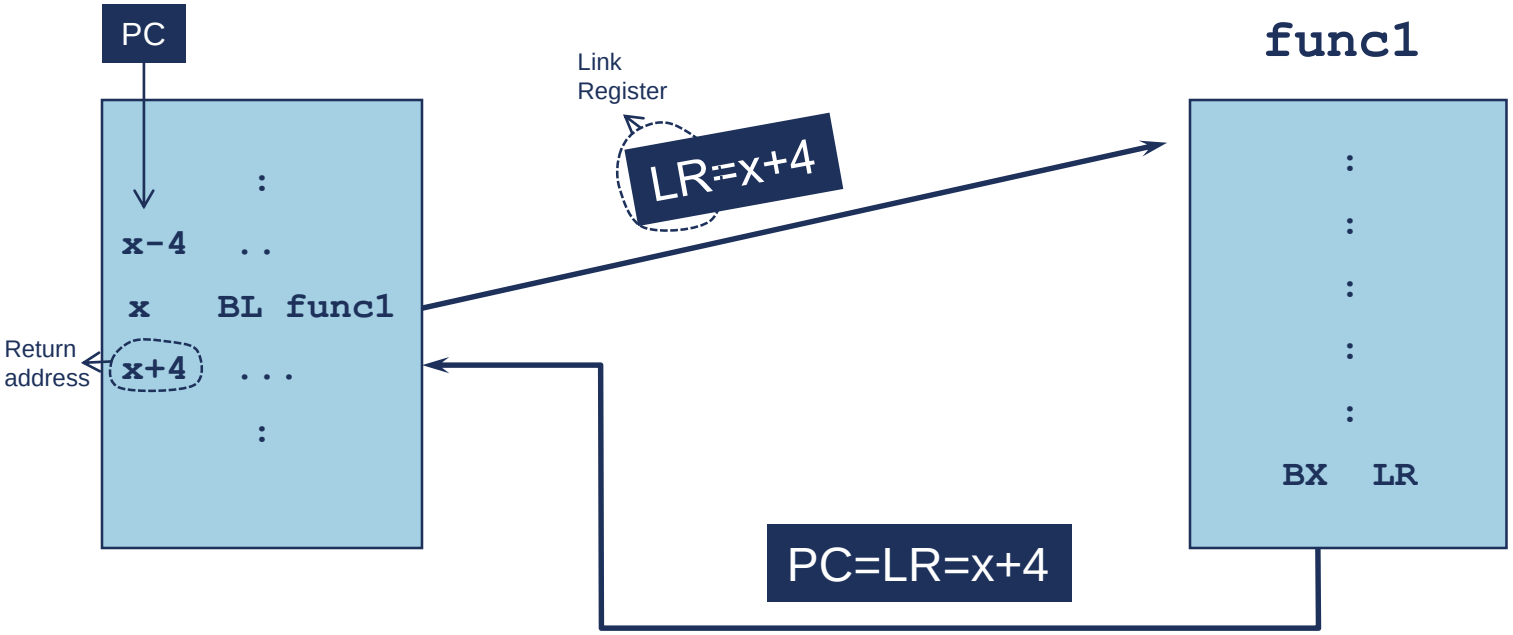


- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range

ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR *(instruction BX LR)*
  - For nonleaf functions, LR will have to be stacked *(instructions PUSH & POP)*

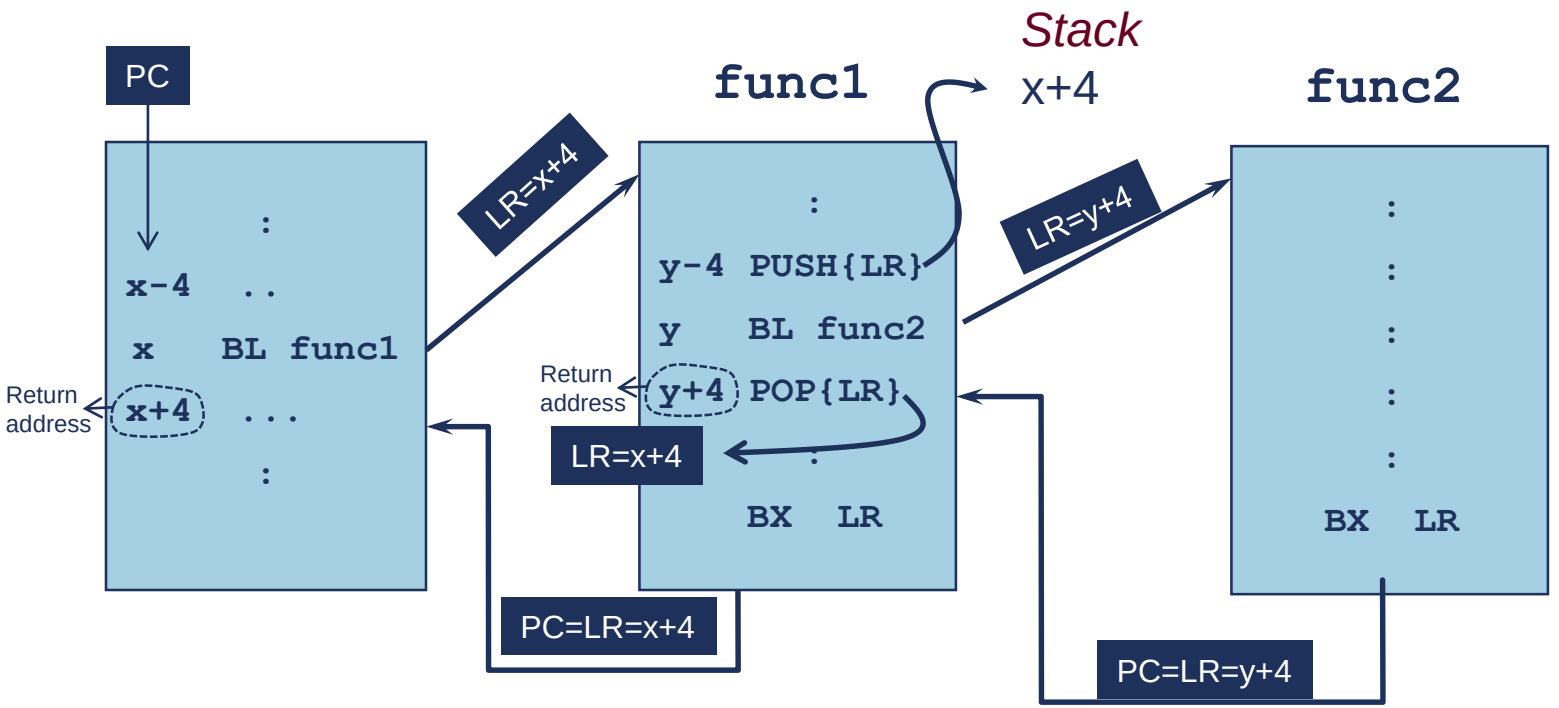
Leaf funcion:
not call other



ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR *(instruction BX LR)*
  - For nonleaf functions, LR will have to be stacked *(instructions PUSH & POP)*

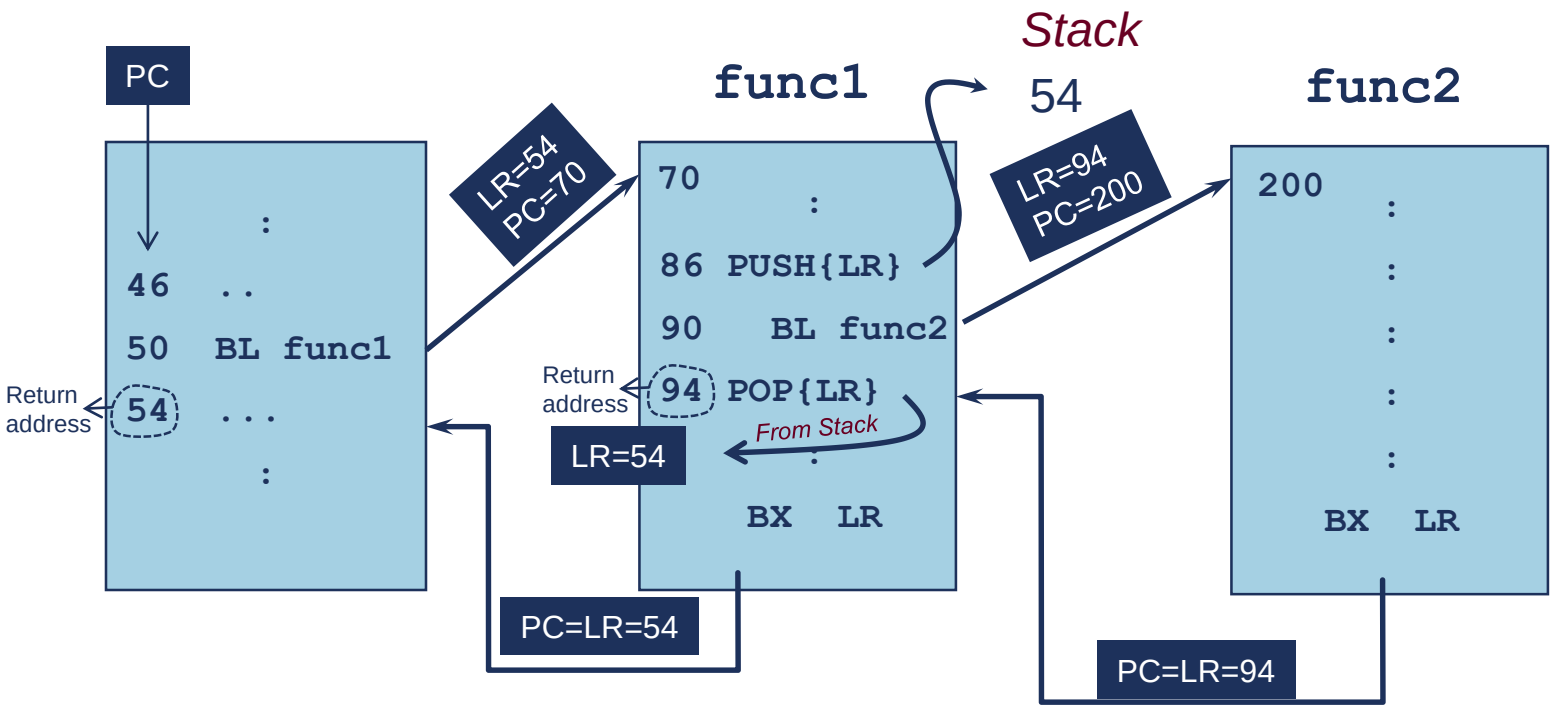
Leaf funcion:
not call other



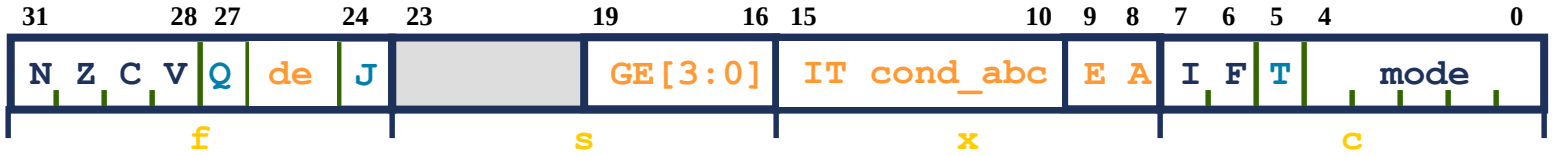
ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR *(instruction BX LR)*
  - For nested functions, LR will have to be stacked *(instructions PUSH & POP)*

nonleaf funcion:
call others



PSR access



- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register or take an immediate value
  - MSR allows the whole status register, or just parts of it to be updated
- Interrupts can be enable/disabled and modes changed, by writing to the CPSR
  - Typically a read/modify/write strategy should be used:

```
MRS r0,CPSR        ; read CPSR into r0
BIC r0,r0,#0x80    ; clear bit 7 to enable IRQ
MSR CPSR_c,r0      ; write modified value to 'c' byte only
```

- In User Mode, all bits can be read but only the condition flags (_f) can be modified

ARM assembly language

ARM assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD r1,r2,r3	r1 = r2 + r3	3 register operands
	subtract	SUB r1,r2,r3	r1 = r2 – r3	3 register operands
Data transfer	load register	LDR r1, [r2,#20]	r1 = Memory[r2 + 20]	Word from memory to register
	store register	STR r1, [r2,#20]	Memory[r2 + 20] = r1	Word from memory to register
	load register halfword	LDRH r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	load register halfword signed	LDRHS r1, [r2,#20]	r1 = Memory[r2 + 20]	Halfword memory to register
	store register halfword	STRH r1, [r2,#20]	Memory[r2 + 20] = r1	Halfword register to memory
	load register byte	LDRB r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	load register byte signed	LDRBS r1, [r2,#20]	r1 = Memory[r2 + 20]	Byte from memory to register
	store register byte	STRB r1, [r2,#20]	Memory[r2 + 20] = r1	Byte from register to memory
	swap	SWP r1, [r2,#20]	r1 = Memory[r2 + 20], Memory[r2 + 20] = r1	Atomic swap register and memory
	mov	MOV r1, r2	r1 = r2	Copy value into register
Logical	and	AND r1, r2, r3	r1 = r2 & r3	Three reg. operands; bit-by-bit AND
	or	ORR r1, r2, r3	r1 = r2 | r3	Three reg. operands; bit-by-bit OR
	not	MVN r1, r2	r1 = ~ r2	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	LSL r1, r2, #10	r1 = r2 << 10	Shift left by constant
	logical shift right (optional operation)	LSR r1, r2, #10	r1 = r2 >> 10	Shift right by constant
Conditional Branch	compare	CMP r1, r2	cond. flag = r1 – r2	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if (r1 == r2) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	B 2500	go to PC + 8 + 10000	Branch
	branch and link	BL 2500	r14 = PC + 4; go to PC + 8 + 10000	For procedure call

ARM vs MIPS

	Instruction name	ARM	MIPS
Register-register	Add	ADD	addu, addiu
	Add (trap if overflow)	ADDS; SWIVS	add
	Subtract	SUB	subu
	Subtract (trap if overflow)	SUBS; SWIVS	sub
	Multiply	MUL	mult, multu
	Divide	—	div, divu
	And	AND	and
	Or	ORR	or
	Xor	EOR	xor
	Load high part register	MOVT	lui
	Shift left logical	LSL[1]	sllv, sll
	Shift right logical	LSR[1]	srlv, srl
	Shift right arithmetic	ASR[1]	srav, sra
	Compare	CMP, CMN, TST, TEQ	slt/i, slt/iu
Data transfer	Load byte signed	LDRSB	lb
	Load byte unsigned	LDRB	lbu
	Load halfword signed	LDRSH	lh
	Load halfword unsigned	LDRH	lhu
	Load word	LDR	lw
	Store byte	STRB	sb
	Store halfword	STRH	sh
	Store word	STR	sw
	Read, write special registers	MRS, MSR	move
	Atomic Exchange	SWP, SWPB	ll;sc

Examples: ARM vs MIPS

lw $1, dato($0)	**ldr r2, =dato** **ldr r1, [r2]**	// load r2 with a constant (which is in memory) // load r1 with m(r2)
sw $1, dato($0)	**str r2, =dato** **str r1, [r2]**	// store r2 in the address of the data // store r1 in m(r2)
add $1, $2, $3 **(sub …)**	**add r1, r2, r3** **(sub …)**	
addi $1, $2, 1	**add r1, r2, #1**	
sll $1, $2, 4	**mov r1, r2, LSL #4**	// logic left shift 4 bits from r2 to r1

sra $1, $2, 2
srl $1, $2, 2
	mov r1, r2, ASR #2
	mov r1, r2, LSR #2
	add r1, r1, r1,LSL #1
		// arithmetic right shift 2 bits from r2 to r1
		// logic right shift 2 bits from r2 to r1
		// r1 ← r1 + (r1 << 1) = **3*r1** (multiplication times 3)

j dir	**b dir**	// jump dir
jal fun	**bl fun**	// jump and link (return address in Lr)
jr $ra	**bx lr**	// return from a routine (lr has the return address)

beq $1, $2, dir
(bne)
	cmp r1, r2
	beq dir
	(bne)
		// compare r1 & r2 (r1-r2), Z flag affected.
		// jump if Z=1
		// jump if Z=0

Other examples of ARM vs. MIPS

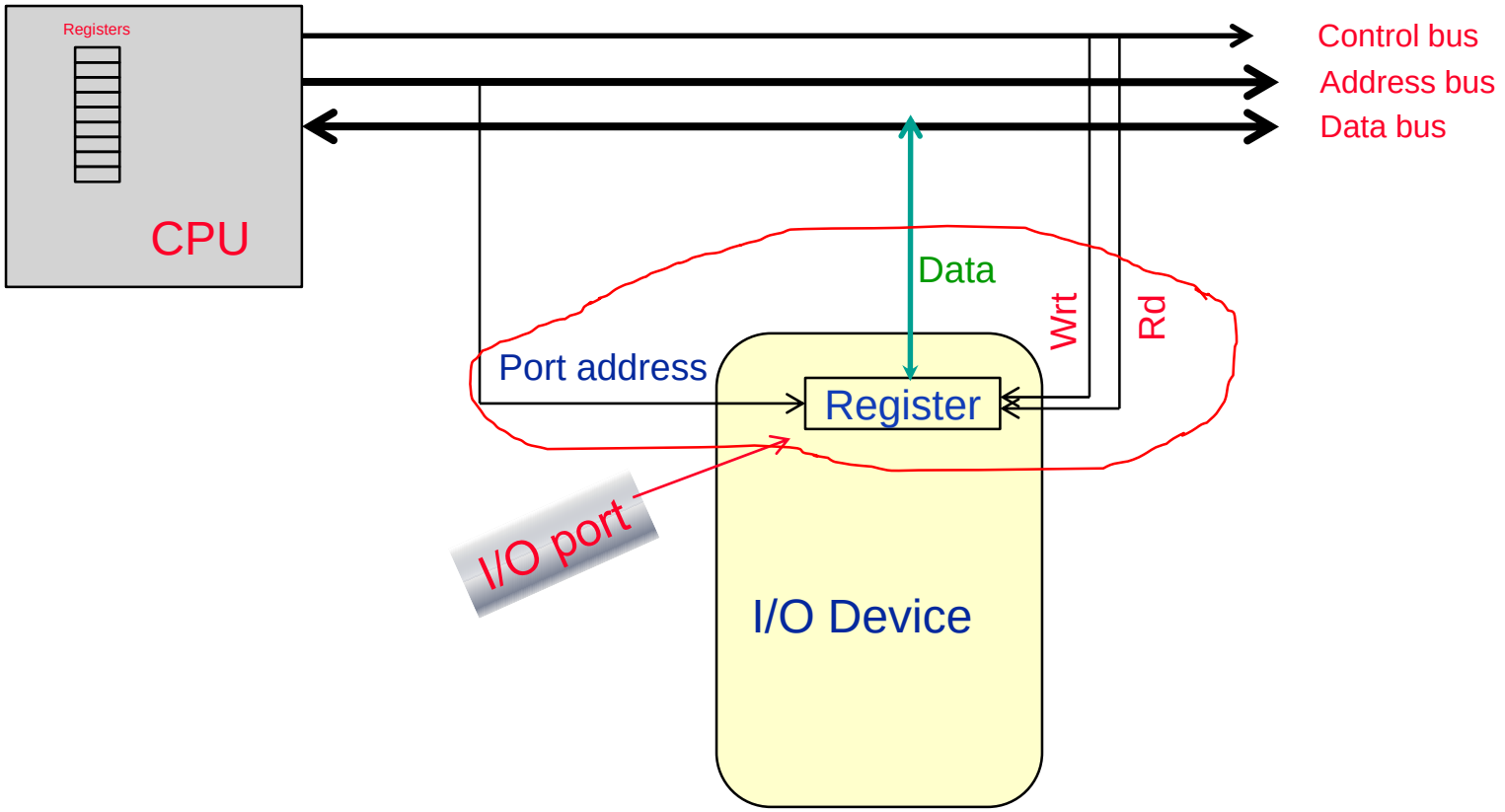
Stack management		
addi $sp, $sp, -4 sw $ra, 0($sp)	**push {lr}**	// save the content of a register (lr) in the stack. Several registers can be saved at a time {r1, r2, r3, r4}
lw $ra, 0($sp) addi $sp, $sp, 4	**pop {lr}**	// Take the content from the stack and load in a register (lr). Several register can be taken at a time {r1, r2, r3, r4}
Loop control when a counter reaches 0		
addi $8, $8, -1 beq $8, $0, exit	adds r8, r8, #-1 beq exit	// decrementa r8 y guarda estado en flags (Z) // mira flag Z, si Z=1 salta (Z cargado en adds)
Auto-increment operation to traverse data structures		
lw $8, 0($9) add $9, $9, 4	ldr r8, [r9], #4	// post-self-increment

Input/output Ports

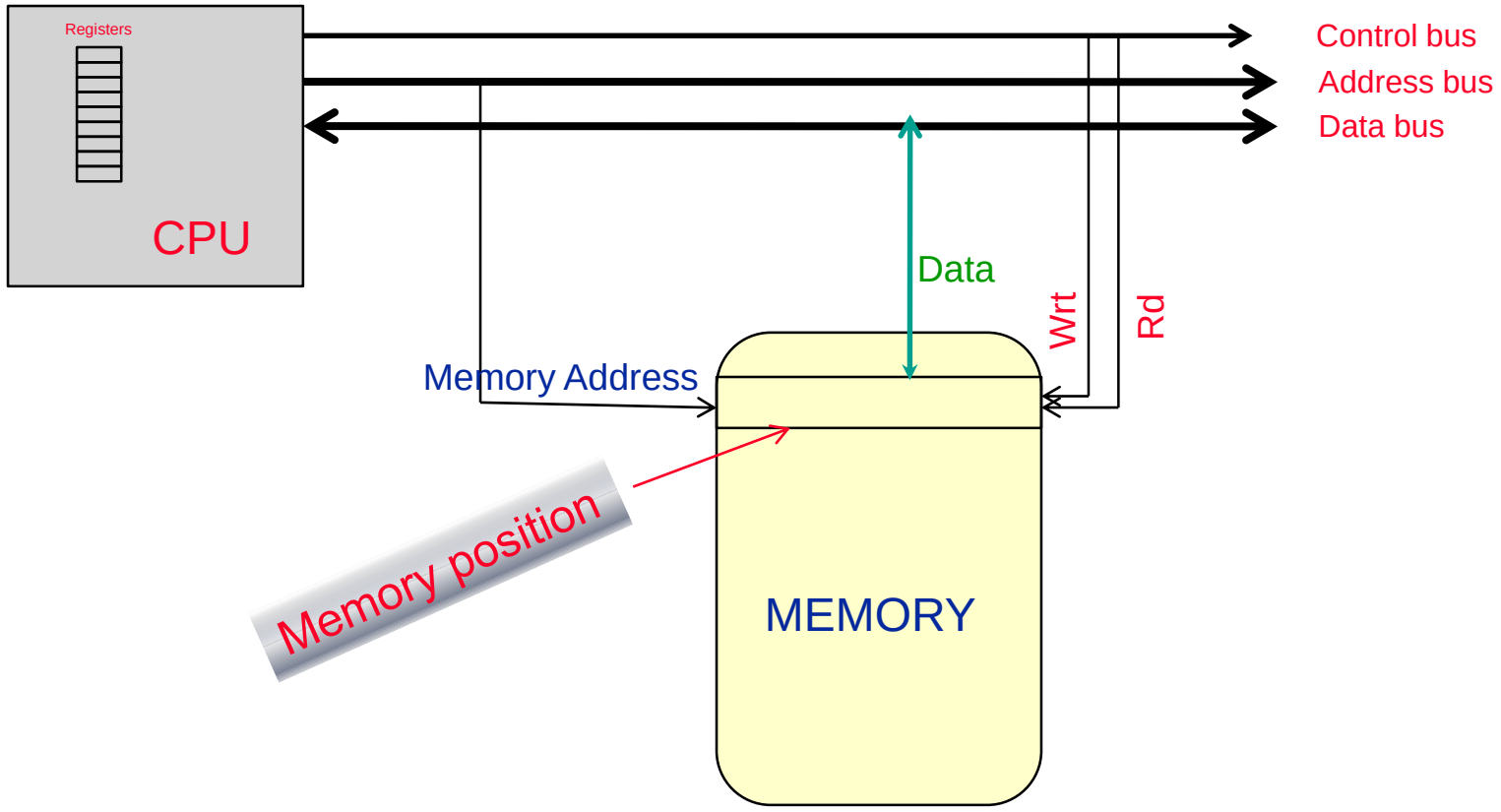
I/O ports

- Communication between CPU and I/O devices
  - How does the processor communicate with devices other than main memory?
    - By using Input/output ports
  - I/O ports
    - Input port: transfers from external device to CPU
    - Output port: transfers from CPU to external
    - Input/Output ports: transfers in both directions

I/O port access



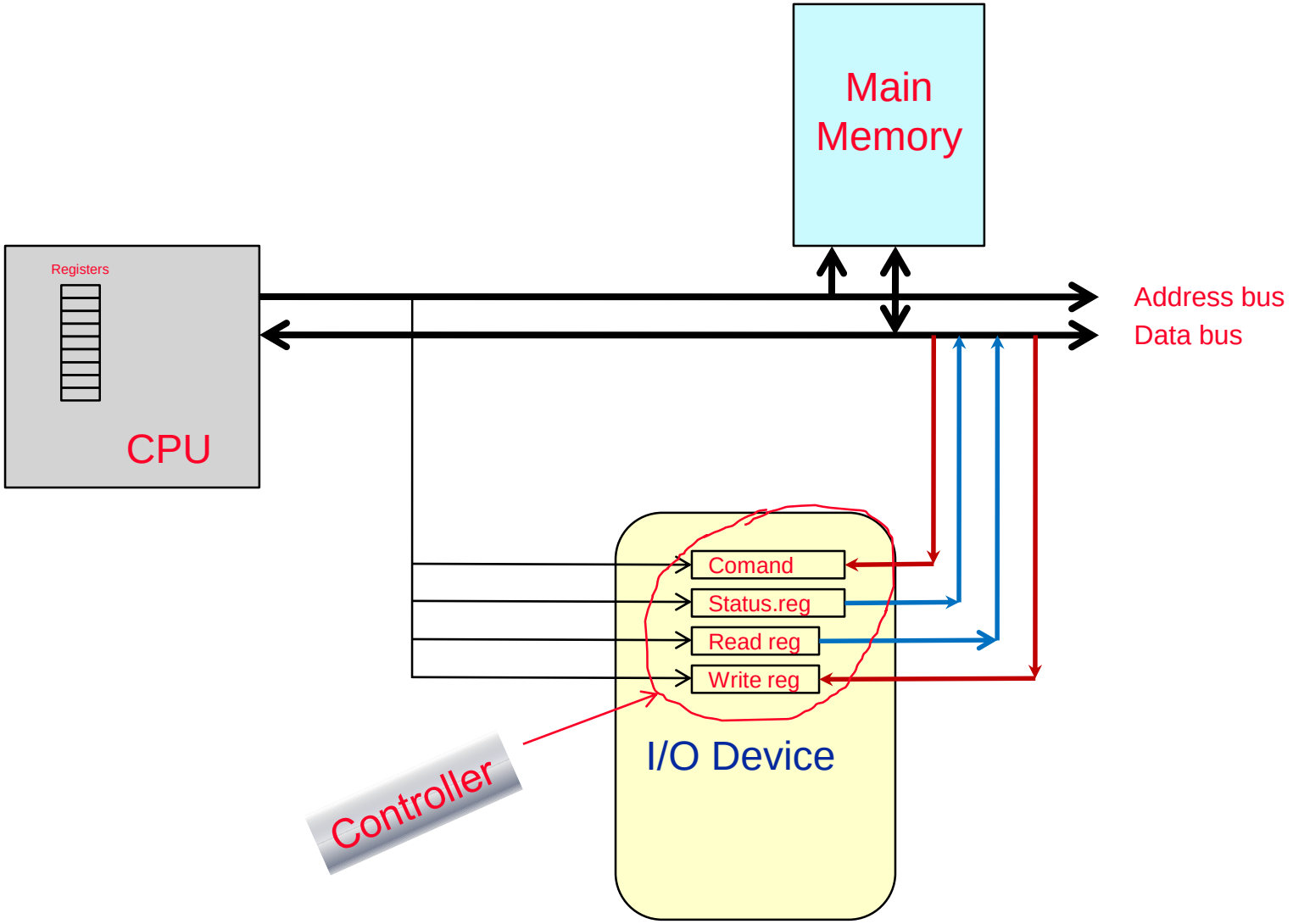
Memory access



I/O Commands

- I/O devices are managed by I/O *controller hardware*
  - Transfers data to/from device
  - Synchronizes operations with software
- Ports in a I/O controller:
  - Command registers
    - Cause device to do something
  - Status registers
    - Indicate what the device is doing and occurrence of errors
  - Data registers
    - Write: transfer data to a device
    - Read: transfer data from a device

I/O Commands


Main Memory
Registers
CPU
Address bus
Data bus
Comand
Status.reg
Read reg
Write reg
I/O Device
Controller

Communication of I/O Devices and Processor

- User programs (processor in protected mode) are prevented from issuing I/O operations directly because the OS does not provide access to the I/O ports
- Only when processor is in kernel (supervisor) mode, then the I/O ports can be accessed
- How the processor directs the I/O devices: through the address space

1. Memory-mapped I/O

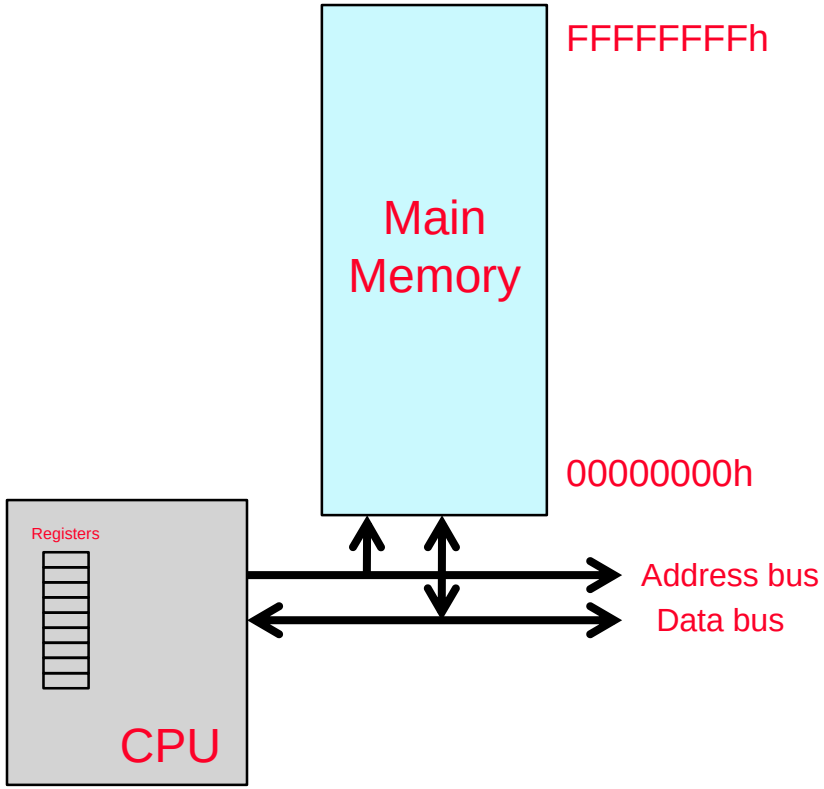
2. Specific I/O instructions

Communication of I/O Devices and Processor

- How the processor directs the I/O devices
  1. Memory-mapped I/O
     - Portions of the high-order memory address space are assigned to each I/O device
     - Read and writes to those memory addresses are interpreted as commands to the I/O devices
     - Load/stores to the I/O address space can *only* be done by the OS
     - MIPS processor:
       - Load instruction to read from I/O device
         - i.e. lw $4, 100($5)
       - Store instruction to write to I/O device
         - i.e. sw $4, 100($5)

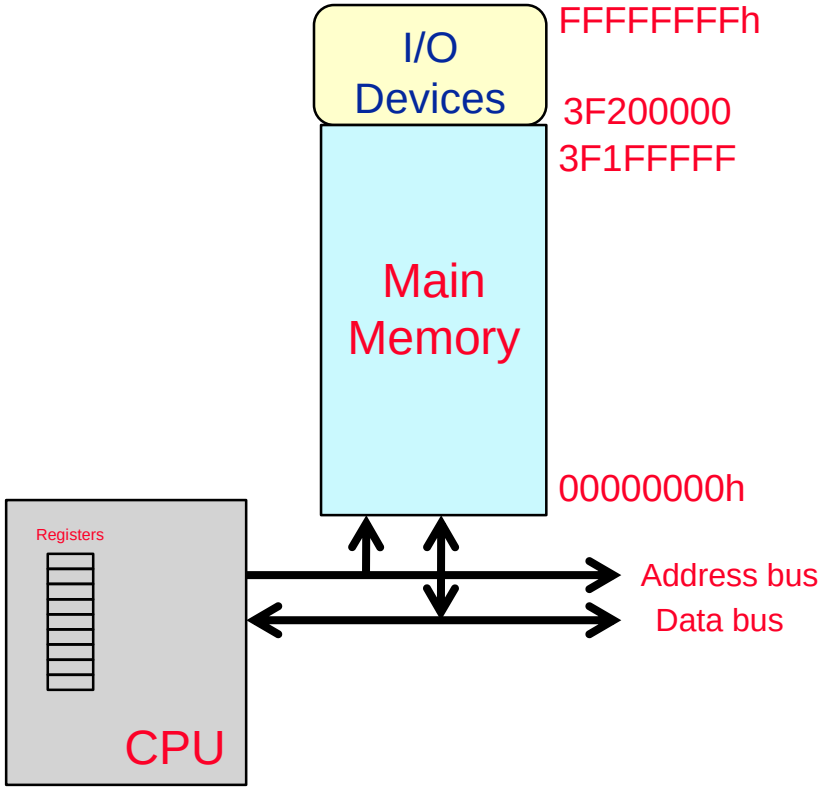
Communication of I/O Devices and Processor

- How the processor directs the I/O devices
  - Memory-mapped I/O (MIPS)



System without I/O device

System with I/O device

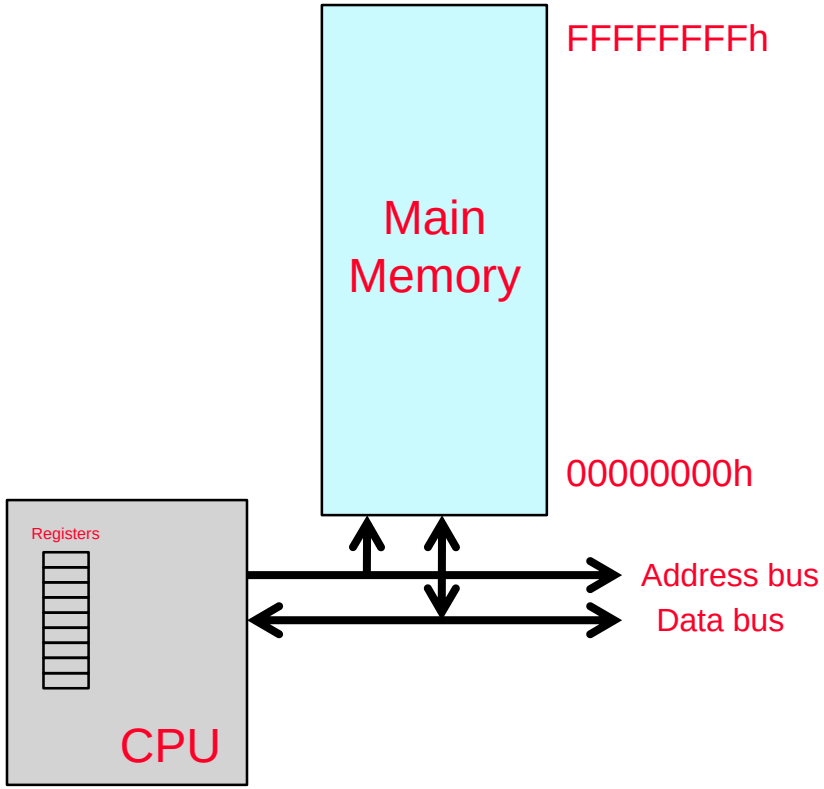


Communication of I/O Devices and Processor

- How the processor directs the I/O devices
  2. I/O instructions
     - Separate instructions to access I/O registers
     - Can only be executed in kernel mode
     - Example: x86:
       - Control signal: $\overline{IO}$/M (1= memory, 0 = IO)
       - Specific Input instruction to read from I/O device
         - i.e. in al, 37h
           - P(37) → AL register
       - Specific Output instruction to write to I/O device
         - i.e. out 37h, al
           - AL register → P(37)

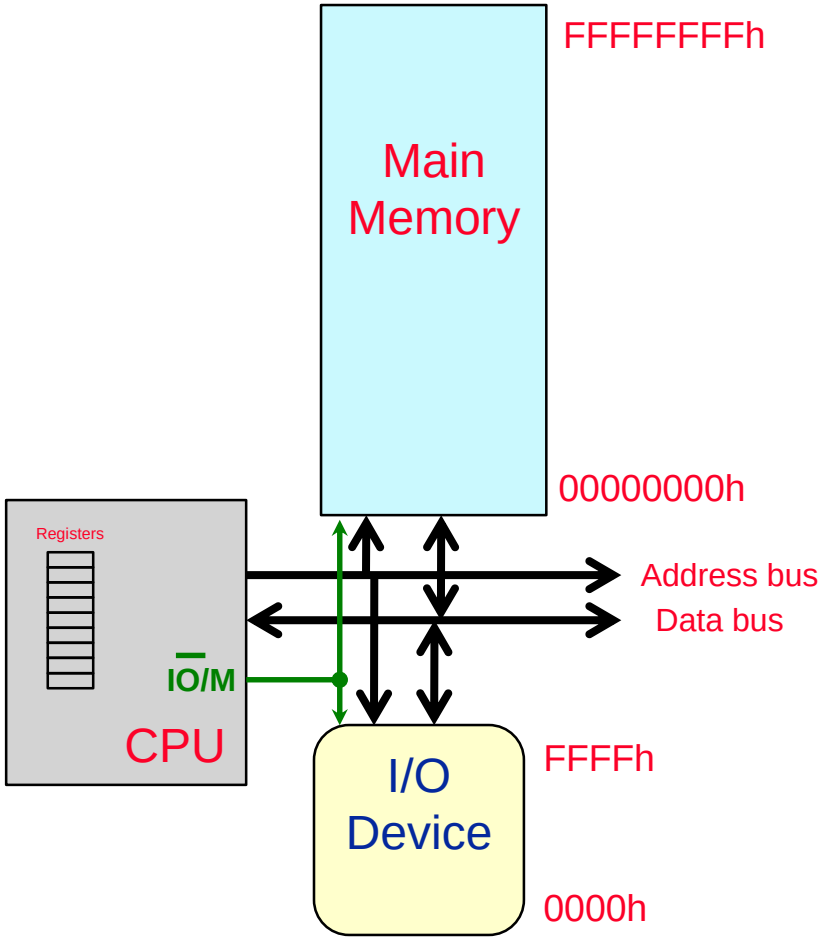
Communication of I/O Devices and Processor

- How the processor directs the I/O devices
  - I/O instructions (x86)

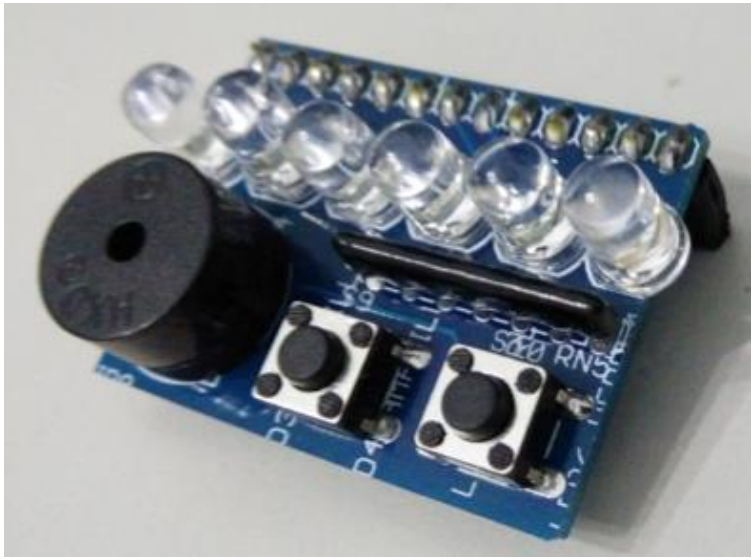


System without I/O device

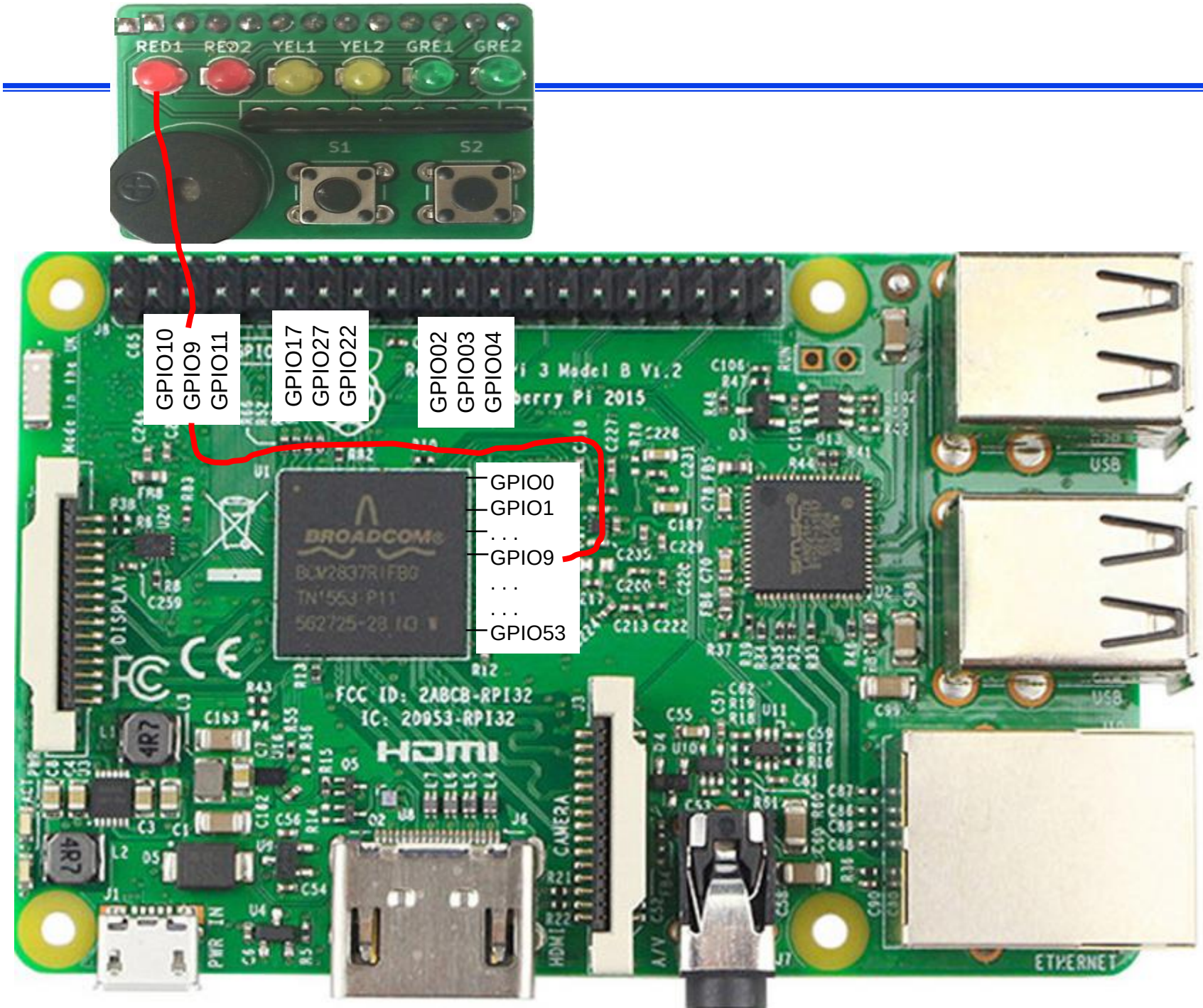
System with I/O device



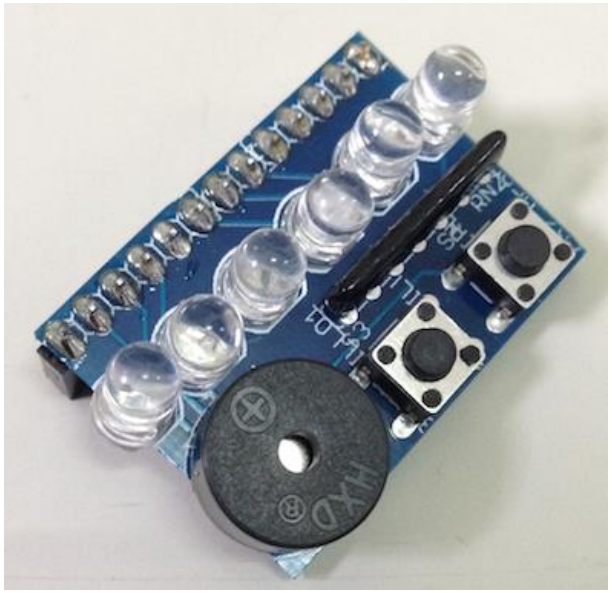
Raspberry Pi: GPIO port

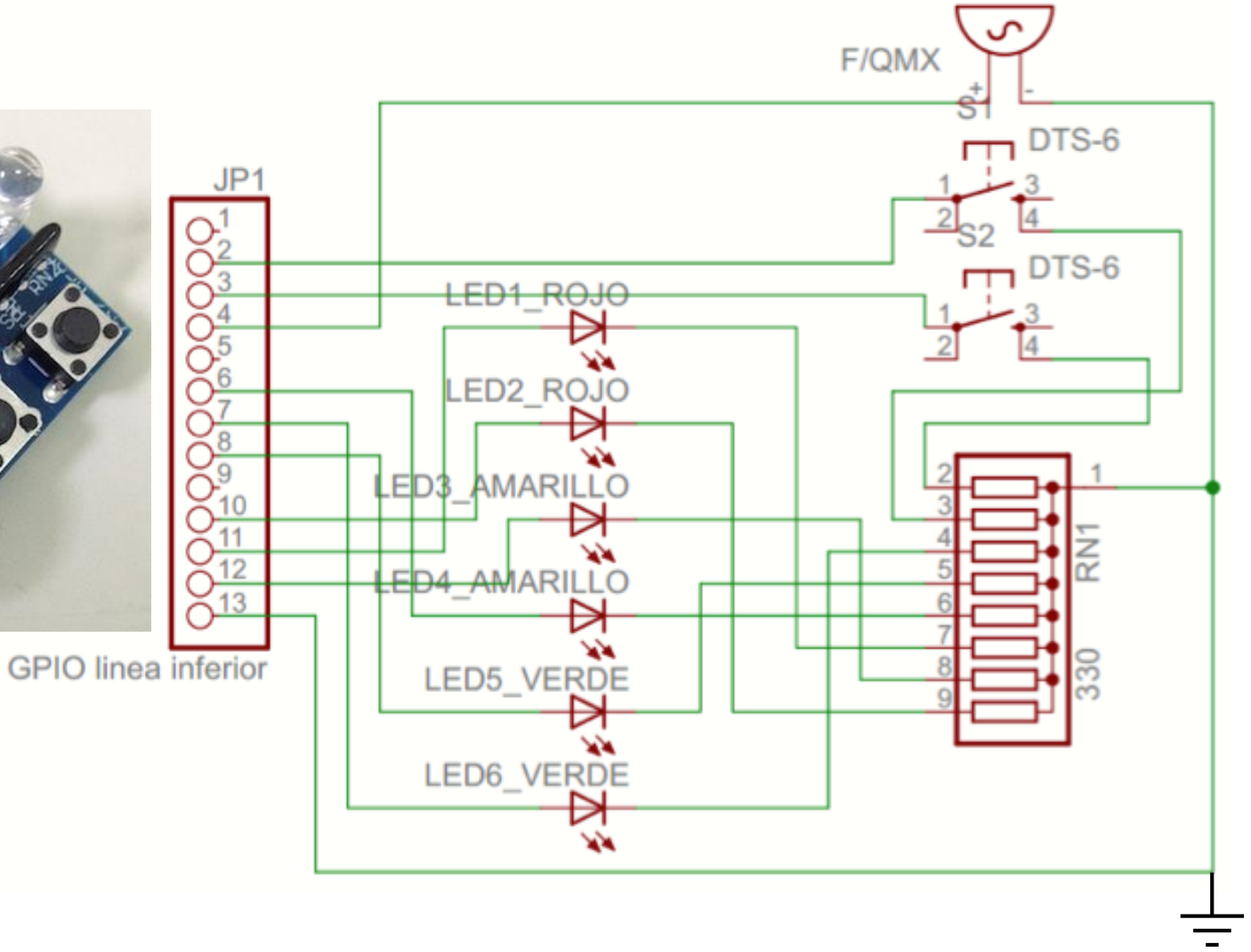


- GPIO: two rows with 13 pins (out of 54)
- We will use the internal row for connecting an external board


RED1
RED2
YEL1
YEL2
GRE1
GRE2
S1
S2
GPIO10
GPIO9
GPIO11
GPIO17
GPIO27
GPIO22
GPIO02
GPIO03
GPIO04
GPIO0
GPIO1
. . .
GPIO9
. . .
. . .
GPIO53
BROADCOM
FCC ID: 2ABCB-RPI32
IC: 20953-RPI32
HDMI
USB
USB
ETHERNET
PWR IN
DISPLAY
CAMERA

External board connected to GPIO





GPIO pins

- Raspberry Pi manages up to 54 pins
- Only the showed ones are accessible (9 GPIO pins)
- GPIO ports are mapped in memory, starting at 0x3F200000

GPIO9

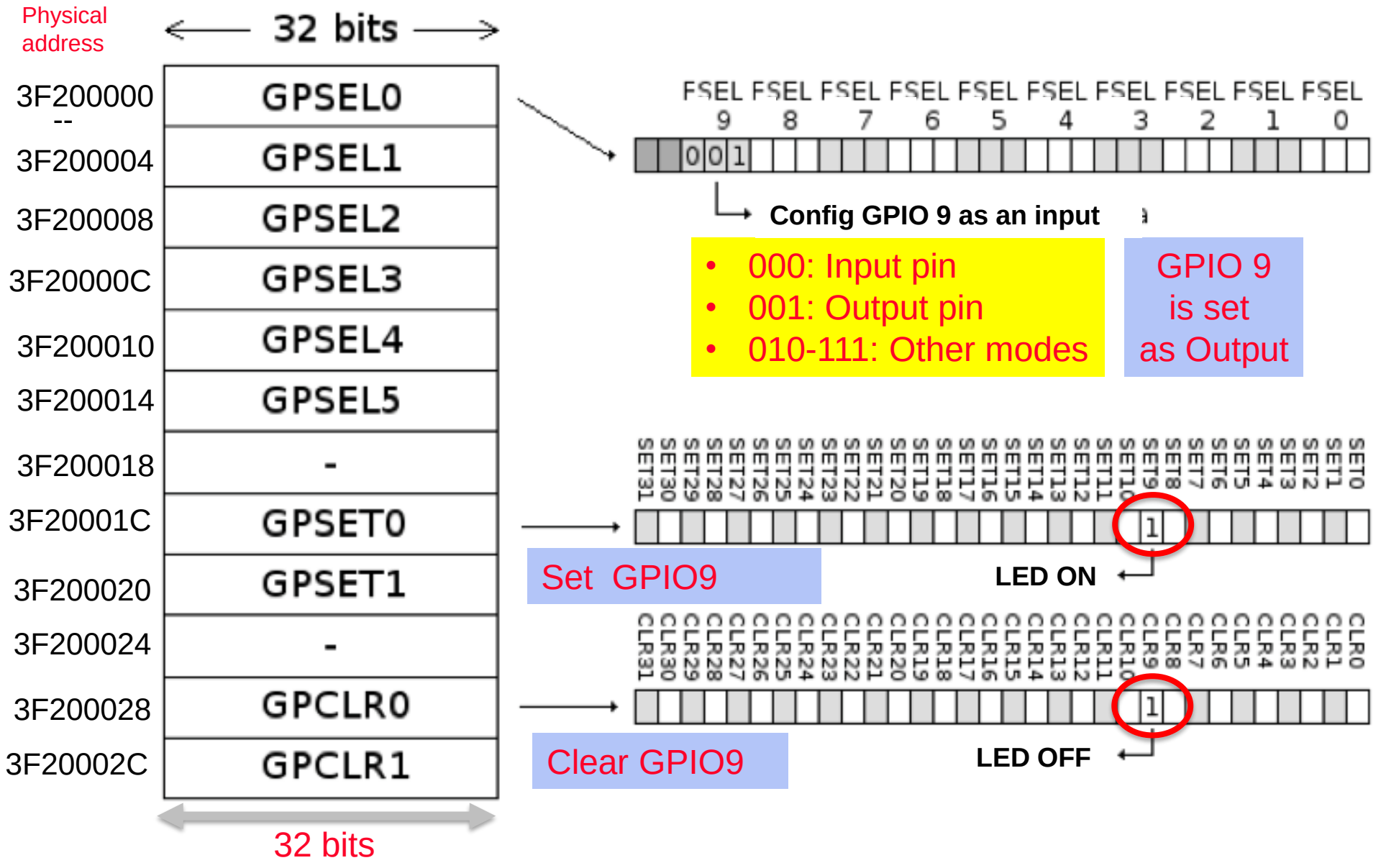
		Pin No.		
	3.3V	1	2	5V
	GPIO2	3	4	5V
	GPIO3	5	6	GND
	GPIO4	7	8	GPIO14
	GND	9	10	GPIO15
	GPIO17	11	12	GPIO18
	GPIO27	13	14	GND
	GPIO22	15	16	GPIO23
	3.3V	17	18	GPIO24
	GPIO10	19	20	GND
	GPIO9	21	22	GPIO25
	GPIO11	23	24	GPIO8
	GND	25	26	GPIO7

Tx
Rx

GPIO memory mapping

- GPIO ports
  - GPFSELn: GPIO Function Select Registers
    - The 54 pins are configured through 6 memory ports, GPSEL0 to GPSEL5
    - Each port defines 10 groups named FSELx, FSEL0 to FSEL9
    - A group consists of 3 bits
    - GPFSEL0 controls GPIO0 to GPIO9, GPFSEL1 controls GPIO10 to GPIO19, …
  - GPSETn: GPIO Pin Output Set Registers
    - GPSET0 sets pins 0 to 31, and GPSET1 sets pins 32 to 53
  - GPCLRn: GPIO Pin Output Clear Registers
    - GPCLR0 clears pins 0 to 31, and GPCLR1 clears pins 32 to 53
  - A SET or CLR operation in any pin just needs 1 in the corresponding position and only affects that pin (0 means that the pin is not modified)

GPIO memory mapping



Example 1

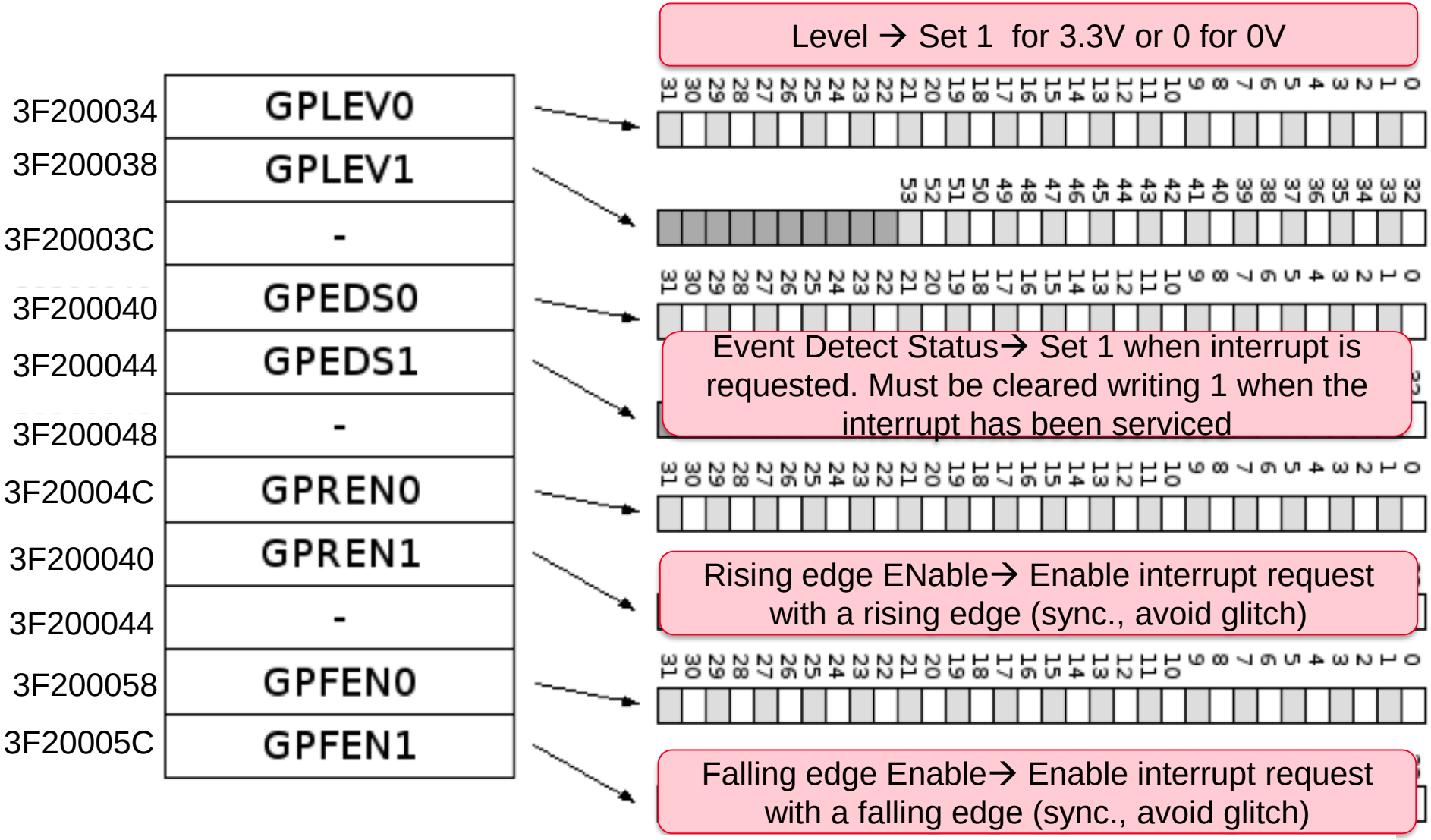
- Code for turning on a red LED (GPIO9):

```
        .set    GPBASE,  0x3F200000
        .set    GPFSEL0, 0x00
        .set    GPSET0,  0x1c
.text
        ldr     r0, =GPBASE
/* guia bits          xx9998887776665554443332221110 00*/
        mov     r1, #0b00001000000000000000000000000000
        str     r1, [r0, #GPFSEL0]  @ Configura GPIO 9
/* guia bits          10987654321098765432109876543210*/
        mov     r1, #0b00000000000000000000001000000000
        str     r1, [r0, #GPSET0]   @ Enciende GPIO 9
infi:   b       infi
```

GPIO memory mapping cont.

- Other GPIO ports
  - GPLEVn: GPIO Pin Level Registers
    - Returns 0 if level is 0V or 1 when level is 3.3V
  - GPEDSn: GPIO Event Detect Status Registers
    - Manages interruption requests
  - GPRENn / GPFENn: GPIO Rising / Falling Edge Detect Enable Registers
    - Enable interruptions fired by rising edge / falling edge
    - Synchronous edge detection (sampled by system clock). It suppresses glitches
  - GPHENn / GPLENn: GPIO High / Low Detect Enable Registers
    - Enable interruptions fired by high/ low level
  - GPAENn / GPAFENn: GPIO Asynchronous rising Edge / Falling Edge Detect Enable Registers
    - Enable interruptions fired by asynchronous rising edge / falling edge
    - For detecting edges of very short duration
  - GPPUD /GPPUDCLKn : GPIO Pull-up/down Register

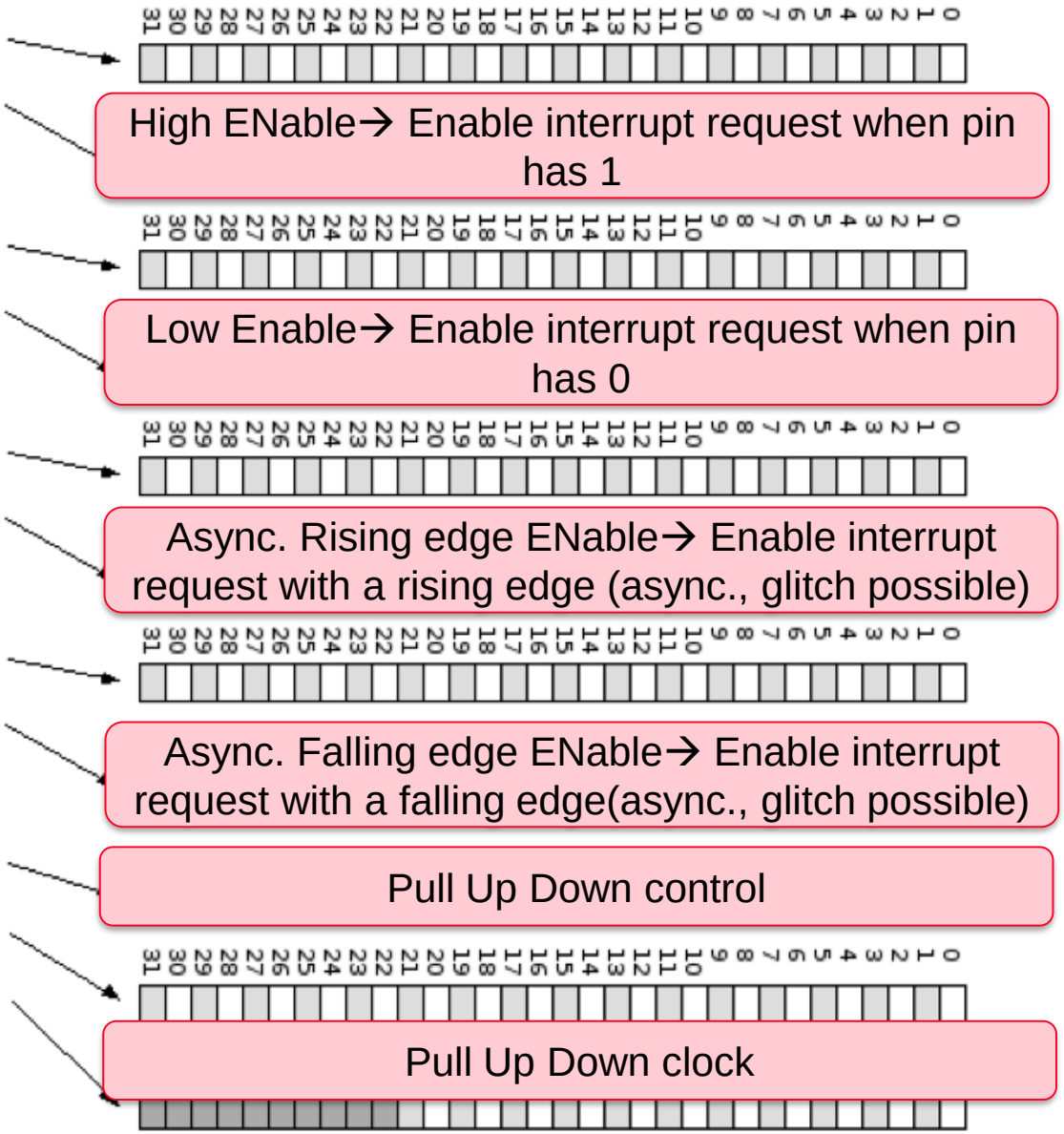
GPIO memory mapping cont.



The suppression of glitches is done by sampling the pin using the system clock and then looking for a “011” (rising) or “011” (falling edge) pattern on the sampled signal.

GPIO memory mapping cont.

3F200064	GPHEN0
3F200068	GPHEN1
3F20006C	-
3F200070	GPLEN0
3F200074	GPLEN1
3F200078	-
3F20007C	GPAREN0
3F200080	GPAREN1
3F200084	-
3F200088	GPAFEN0
3F20008C	GPAFEN1
3F200090	-
3F200094	GPPUD
3F200098	GPPUDCLK0
3F20009C	GPPUDCLK1



Communication of I/O Devices and Processor

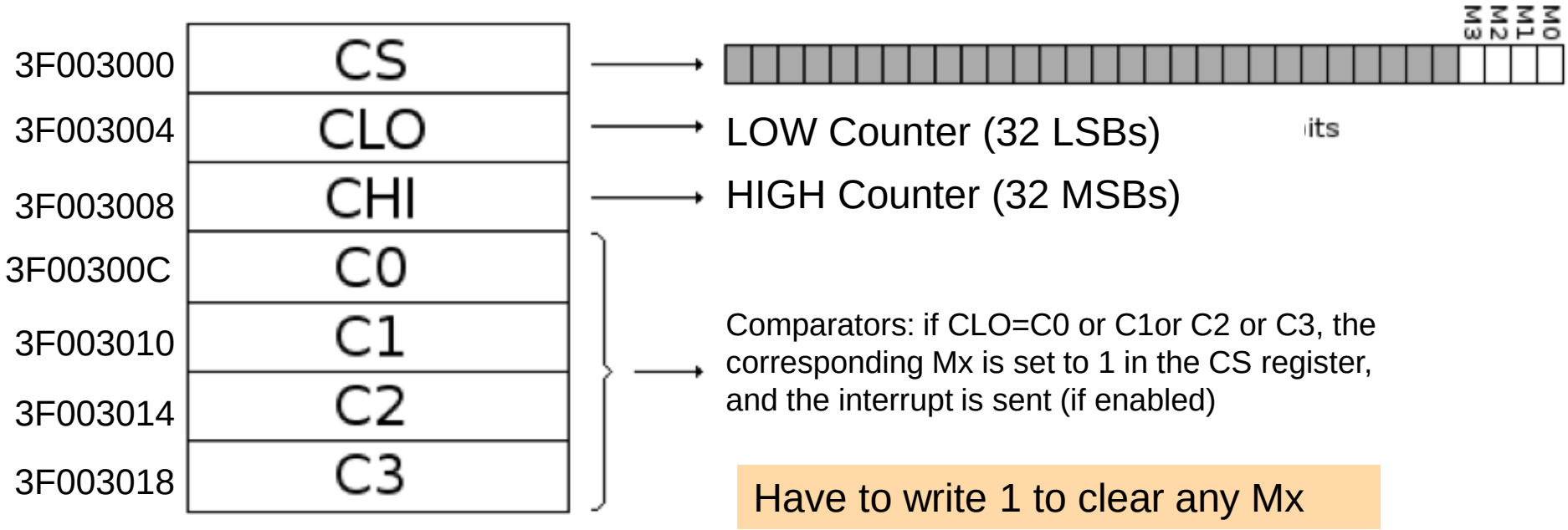
- How I/O devices communicate with the processor
  - ***Polling*** – the processor periodically checks the status of an I/O device (through the OS) to determine its need for service
    - Processor is totally in control – but does all the work
    - In real-time embedded applications:
      - I/O rates are predetermined and it makes I/O overhead predictable (helpful for real time)
    - Can waste a lot of processor time due to speed differences
  - ***Interrupt-driven I/O*** – the I/O device issues an interrupt to indicate that it needs attention
    - Advantages of using interrupts
      - Relieves the processor from having to continuously poll for an I/O event; user program progress is only suspended during the actual transfer of I/O data to/from user memory space
    - Disadvantage – special hardware is needed to
      - Indicate the I/O device causing the interrupt and to save the necessary information prior to servicing the interrupt and to resume normal processing after servicing the interrupt

Polling

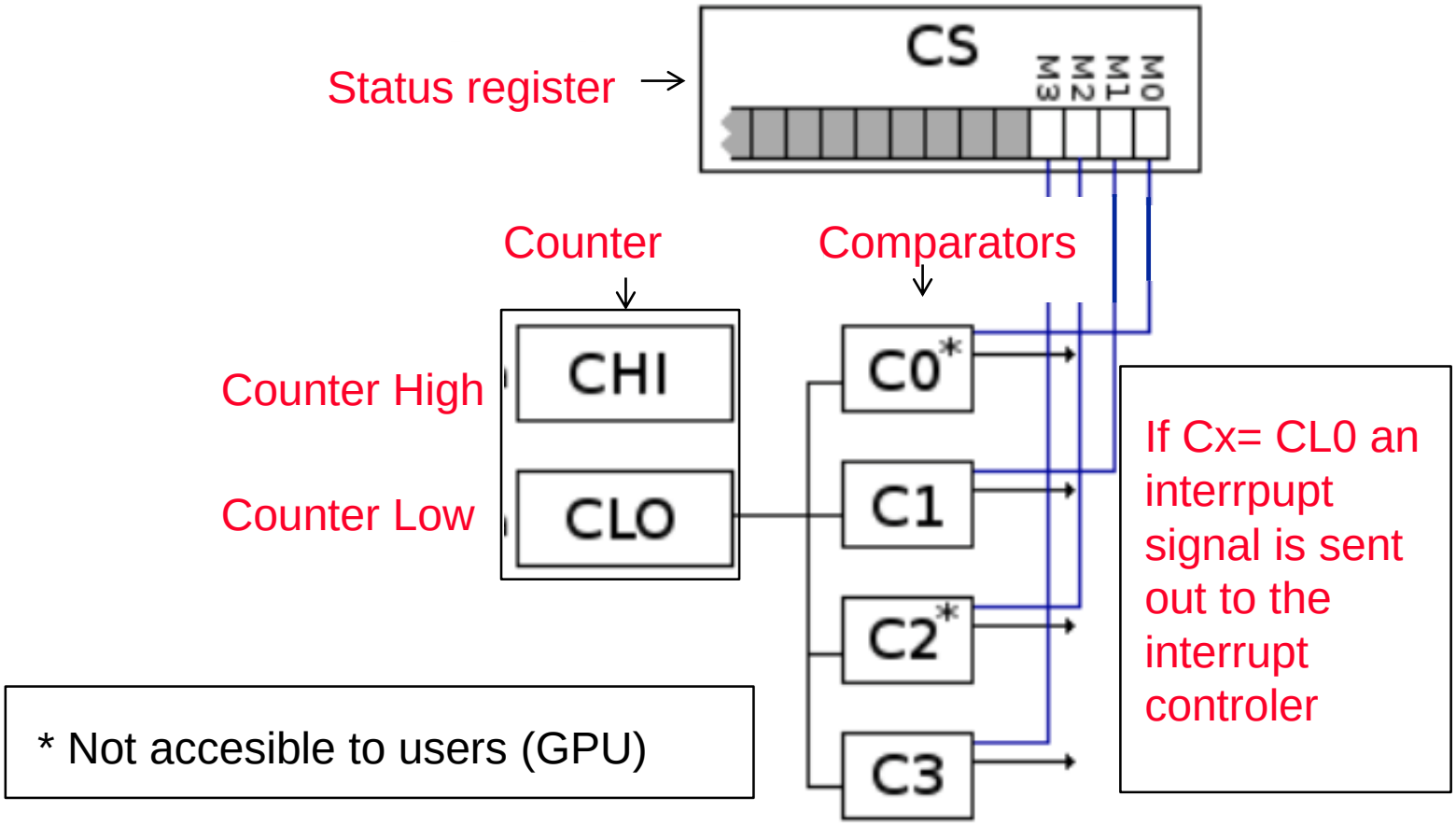
- Periodically check I/O status register
  - If device ready, do operation
  - If error, take action
- Common in small or low-performance real-time embedded systems
  - Predictable timing
  - Low hardware cost
- In other systems, wastes CPU time

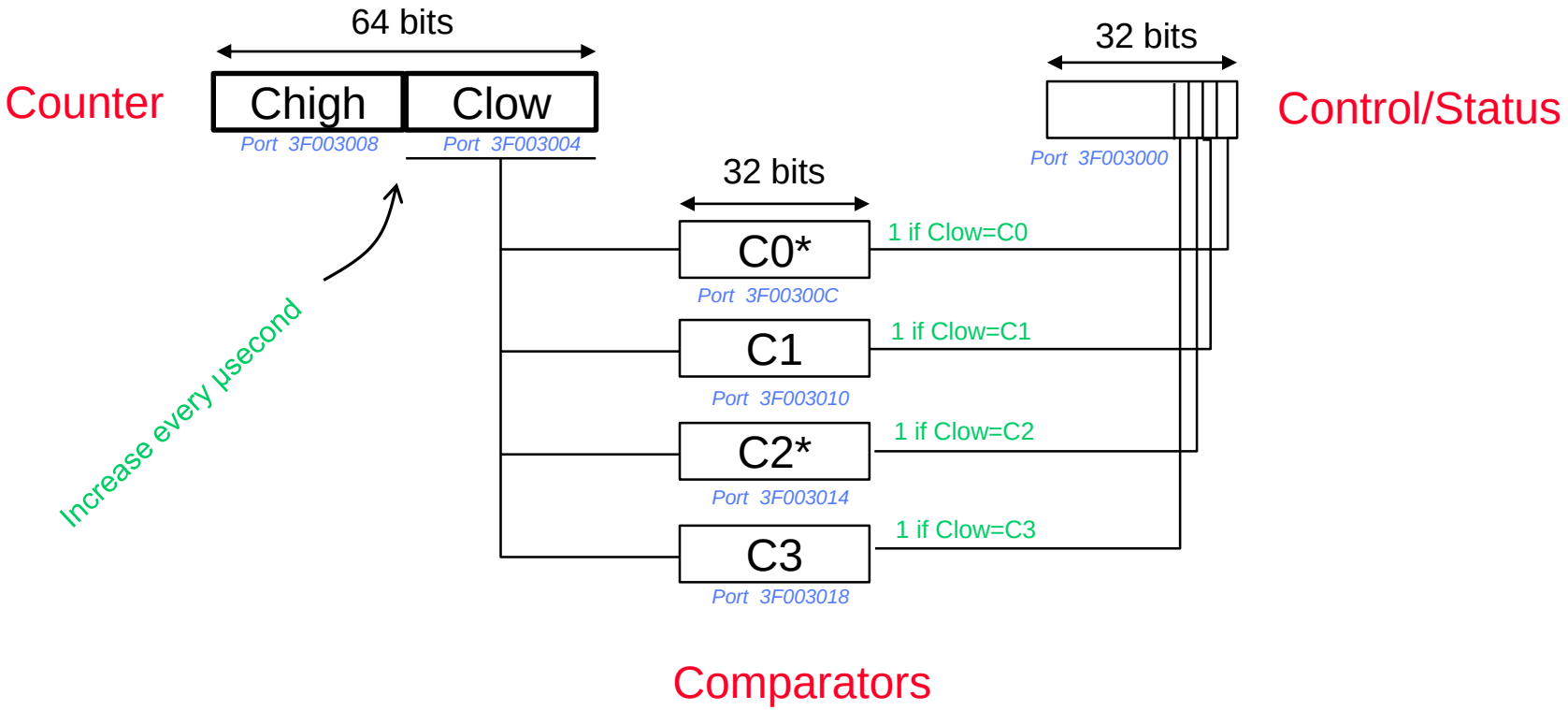
Raspberry Pi: System Timer

- 64 bits counter (CHI:CLO → high y low)
- C0 to C3: Comparison registers (4 time channels)
- CS: Control/status register
  - M0 - M3 fields are set to 1 if CLO == C0 : C3
- CLK frequency: 1MHz (each increment 1microsec.)



System Timer





* Not accesible to users (GPU)

Example 4

- Red LED blinking (controlled by the system timer)

```
        .set    GPBASE,   0x3F200000
        .set    GPFSEL0,        0x00
        .set    GPSET0,         0x1c
        .set    GPCLR0,         0x28
        .set    STBASE,   0x3F003000
        .set    STCLO,          0x04
.text
        mov     r0, #0b11010011
        msr     cpsr_c, r0
        mov     sp, #0x8000000  @ Inicializ. pila en modo SVC

        ldr     r4, =GPBASE
/* guia bits           xx99988777666555444333222111000*/
        mov     r5, #0b00001000000000000000000000000000
        str     r5, [r4, #GPFSEL0]  @ Configura GPIO 9
/* guia bits           10987654321098765432109876543210*/
        mov     r5, #0b00000000000000000000001000000000
        ldr     r0, =STBASE     @ r0 es un parametro de espera (dir. base ST)

        ldr     r1, =500000     @ r1 es un parametro de espera (microsec.)
bucle:
        bl      espera          @ Salta a rutina de espera
        str     r5, [r4, #GPSET0]       @ enciende led
        bl      espera          @ Salta a rutina de espera
        str     r5, [r4, #GPCLR0]       @apaga led
        b       bucle
```

Timer ports

Example 4 cont.

- Delay routine:
  1. Load CLO (LOW Counter) in r4
  2. Add r1 (=500000, half a second (1MHz))
  3. Load CLO in r5 and compare with r4: while it is lower then try it again. Otherwise returns

```
/* rutina que espera */
espera:
        push    {r4,r5}
        ldr     r4, [r0, #STCLO]        @ Lee contador en r4
        add     r4, r1                  @ r4= r4+medio millon
ret1:   ldr     r5, [r0, #STCLO]
        cmp     r5, r4                  @ Leemos CLO hasta alcanzar
        blo     ret1                    @ el valor de r4
        pop     {r4,r5}
        bx      lr
```

Example 4 complete

- Code for blinking a red LED (GPIO9):

```
        .set    GPBASE,   0x3F200000
        .set    GPFSEL0,        0x00
        .set    GPSET0,         0x1c
        .set    GPCLR0,         0x28
        .set    STBASE,   0x3F003000
        .set    STCLO,          0x04
.text
        mov     r0, #0b11010011
        msr     cpsr_c, r0
        mov     sp, #0x8000000  @ Inicializ. pila en modo SVC

        ldr     r4, =GPBASE
/* guia bits           xx99988877766655544433322211l000*/
        mov     r5, #0b00001000000000000000000000000000
        str     r5, [r4, #GPFSEL0]  @ Configura GPIO 9
/* guia bits           10987654321098765432109876543210*/
        mov     r5, #0b00000000000000000000001000000000
        ldr     r0, =STBASE     @ r0 es un parametro de espera (dir. base ST)

        ldr     r1, =500000     @ r1 es un parametro de espera (microsec.)
bucle:
        bl      espera          @ Salta a rutina de espera
        str     r5, [r4, #GPSET0]       @ enciende led
        bl      espera          @ Salta a rutina de espera
        str     r5, [r4, #GPCLR0]       @apaga led
        b       bucle

/* rutina que espera medio segundo */
espera:
        push    {r4,r5}
        ldr     r4, [r0, #STCLO]        @ Lee contador en r4
        add     r4, r1                  @ r4= r4+medio millon
ret1:   ldr     r5, [r0, #STCLO]
        cmp     r5, r4                  @ Leemos CLO hasta alcanzar
        blo     ret1                    @ el valor de r4
        pop     {r4,r5}
        bx      lr
```

Timer ports

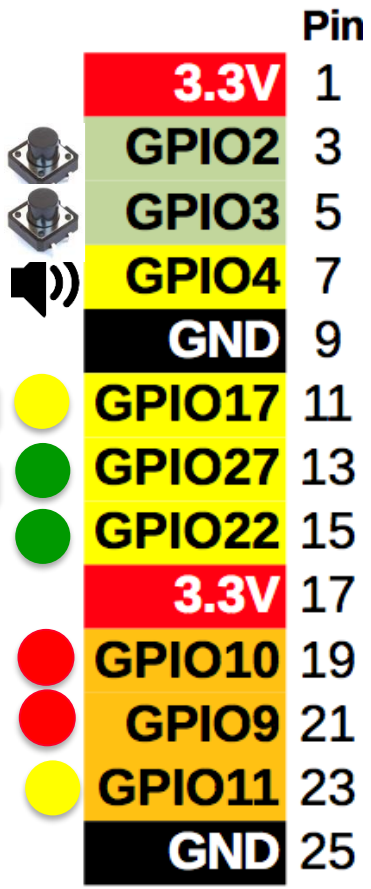
Example 5

- Sound generation
  - Similar to led blinking
  - Differences
    - GPIO4 must be configured as Output

```
        ldr     r0, =GPBASE
        ldr     r1, =0b00000000000000000001000000000000
        str     r1, [r0, #GPFSEL0]
```

    - Wave generation: send 1 and 0 to GPIO4

```
        ldr     r0, =STBASE     @ r0 es un parametro de sonido (dir base ST)
        ldr     r1, =1136       @ r1 es un parametro de sonido (periodo/2)

bucle:  bl      sonido          @ Salta a rutina de sonido
        str     r5, [r4, #GPSET0]
        bl      sonido          @ Salta a rutina de sonido
        str     r5, [r4, #GPCLR0]
        b       bucle

/* rutina que espera r1 microsegundos */
sonido:
        push    {r4,r5}
        ldr     r4, [r0, #STCLO]  @ Lee contador en r4
        add     r4, r1            @ r4= r4 + perido/2
ret1:   ldr     r5, [r0, #STCLO]
        cmp     r5, r4            @ Leemos CLO hasta alcanzar
        blo     ret1              @ el valor de r4
        pop     {r4,r5}
        bx      lr
```



Example 5 complete

- Code for generating a sound (440 Hz, GPIO4):

```
        .set    GPBASE,   0x3F200000
        .set    GPFSEL0,        0x00
        .set    GPSET0,         0x1c
        .set    GPCLR0,         0x28
        .set    STBASE,   0x3F003000
        .set    STCLO,          0x04
.text
        mov     r0, #0b11010011
        msr     cpsr_c, r0
        mov     sp, #0x8000000  @ Inicializ. pila en modo SVC

        ldr     r4, =GPBASE
/* guia bits           xx99988877766655544433322211000*/
        mov     r5, #0b00000000000000000000001000000000000
        str     r5, [r4, #GPFSEL0]  @ Configura GPIO 4
/* guia bits           10987654321098765432109876543210*/
        mov     r5, #0b00000000000000000000000000010000
        ldr     r0, =STBASE     @ r0 es un parametro de sonido (dir base ST)
        ldr     r1, =1136       @ r1 es un parametro de sonido (periodo/2)

bucle:  bl      sonido          @ Salta a rutina de sonido
        str     r5, [r4, #GPSET0]
        bl      sonido          @ Salta a rutina de sonido
        str     r5, [r4, #GPCLR0]
        b       bucle

/* rutina que espera r1 microsegundos */
sonido:
        push    {r4,r5}
        ldr     r4, [r0, #STCLO]  @ Lee contador en r4
        add     r4, r1            @ r4= r4 + perido/2
ret1:   ldr     r5, [r0, #STCLO]
        cmp     r5, r4            @ Leemos CLO hasta alcanzar
        blo     ret1              @ el valor de r4
        pop     {r4,r5}
        bx      lr
```

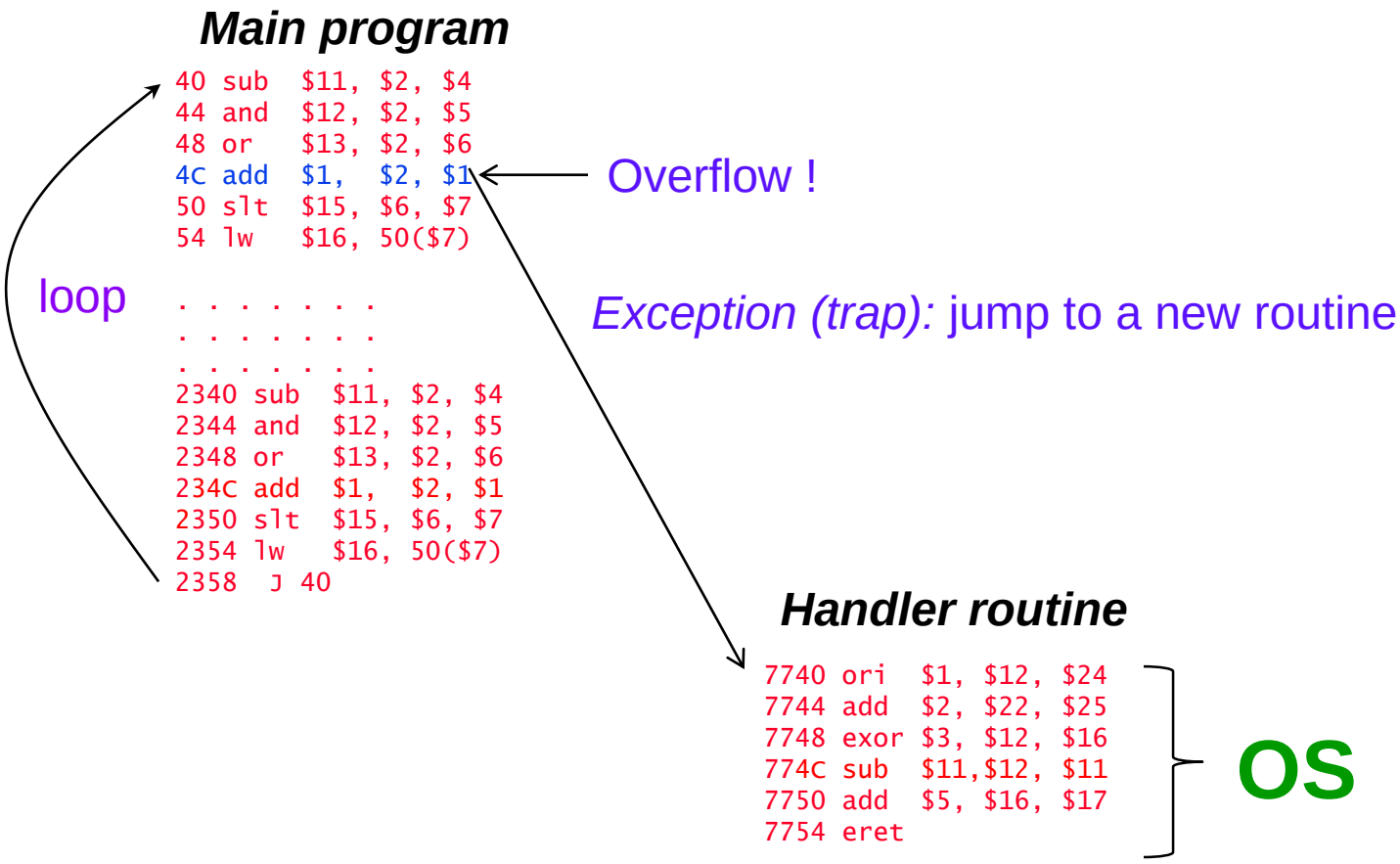
Exceptions

Exceptions

- "Unexpected" events requiring change in flow of instructions execution
  - Branch and Jumps are excluded (they are "expected changes)
- Two possible sources of exceptions
  - Internal exceptions
    - e.g., undefined opcode, overflow, syscall, …
  - External exceptions → INTERRUPTS
    - From an external device (no memory)

Exceptions (trap)

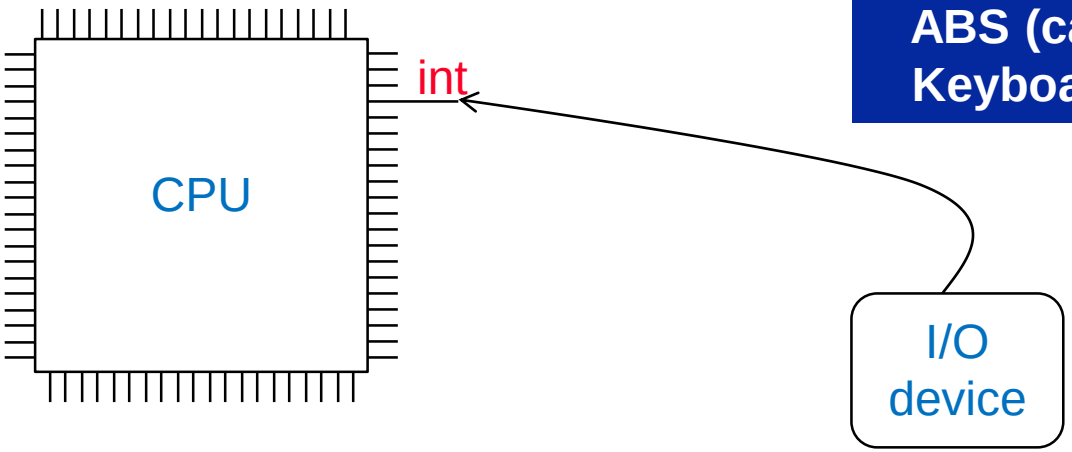
- Example of internal exception (trap)



Two Types of Exceptions

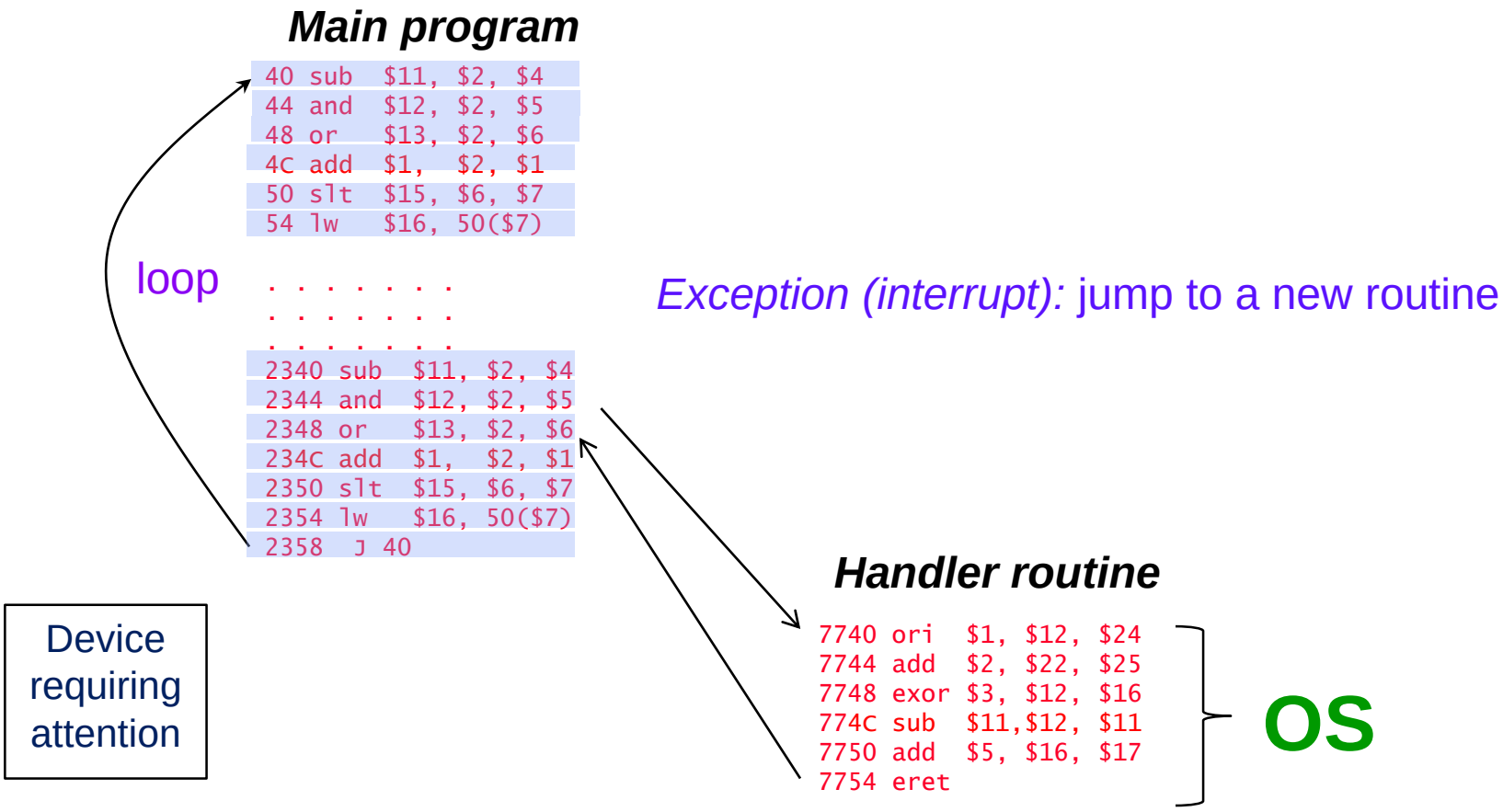
- External exceptions -> ***Interrupts*** – asynchronous to program execution
  - caused by external events
  - may be handled between instructions:
    - let the instructions currently active in the pipeline *complete*
    - pass control to the OS interrupt handler
  - simply suspend and resume user program

Examples:
ABS (cars)
Keyboard (computer)



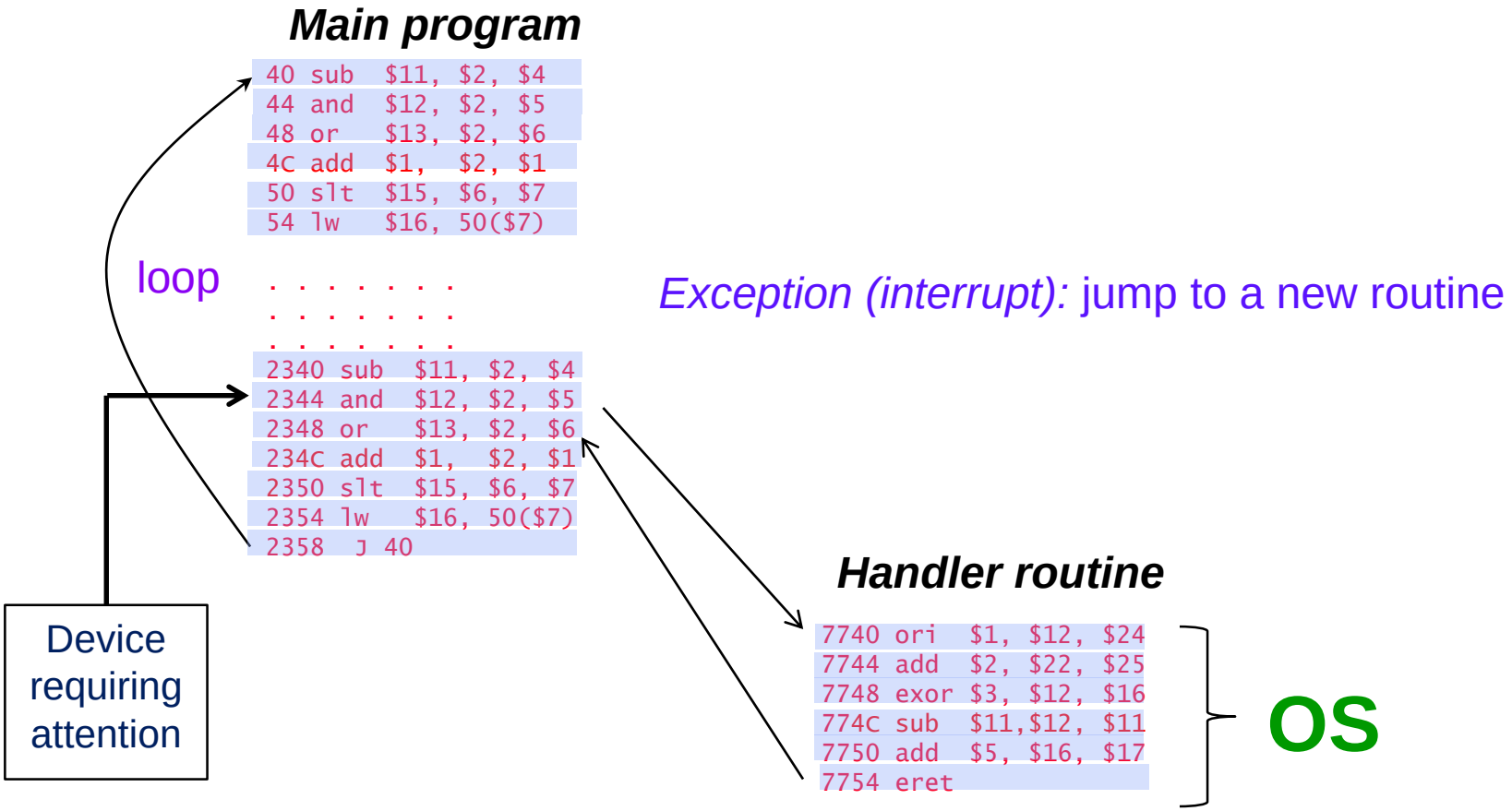
Interrupts

- Example of interrupt



Interrupts

- Example of interrupt



Dealing with Exceptions

- Exceptions are just another form of control hazard. Exceptions arise from
  - Arithmetic overflow (internal, exc.)
  - Trying to execute an undefined instruction (internal, exc.)
  - An OS service request (e.g., a page fault) (internal, exc.)
  - A hardware malfunction (internal or external)
  - An I/O device request (external, int)
- Invoke the OS from the user program (internal, software int. or system call)

- The software (OS) /HW looks at the cause of the exception and “deals” with it

Interrupt Driven I/O

- With I/O interrupts
  - Need a way to identify the device generating the interrupt
    - **Vectored interrupts**: the device can send a vector (id.) to the processor, which uses it to address the table of the interrupt vectors, from where it gets the address of the handle.
    - **Non vectored interrupts**: the device places a status field in the Cause register, jumps to a handler at a fixed direction. → **MIPS**
    - **Auto-vectored interrupts:** each exception has vector associated to it. → **~ARM**
    - When the handler gets control, it knows the identity of the device and can immediately start the I/O operation
  - Can have different urgencies (so need a way to prioritize them)
    - I/O interrupts have lower priority than internal exceptions
    - UNIX OS uses four to six levels
      - Interrupt priority levels (IPLs) assigned by the OS to each process can be raised and lowered via changes to the Status's Interrupt mask field
        - Lowest ILP: all interrupts are permitted
        - Highest ILP: all interrupts are blocked

Exceptions in ARM

Exceptions in ARM

- Autovectored
  - Vector table
- Type of exceptions:
  - Reset
  - Undefined instruction: op. code not valid
  - Software interruptions: system calls
  - Prefecth abort /data abort: memory misalignment, access privilege errors
  - IRQ: regular interruptiont
  - FIQ: fast interruptions

Exceptions in ARM

- ARM's exception system is auto-vectorized
  - There are 8 exception types, NI=0:7
  - Each NI has an exception vector associated to it
    - The exception vector is a jump to a handler
  - Operation modes: User, SVC, IRQ, FIQ, Abort, Undefined
    - An exception forces a change to a new operation mode

Vector table

Exception	Type	Offset	Mode
Reset	Interruption	0x00	SVC
Undefined Instruct.	Exception	0x04	Undefined
SW interrupt	SW Interrup.	0x08	SVC
Prefetch abort	Exception	0x0C	Abort
Data abort	Exception	0x10	Abort
Reserved	-	0x14	-
IRQ	Interruption	0x18	IRQ
FIQ	Interruption	0x1C	FIQ

Exceptions in ARM

- Contents of the vector table (memory)

Vector table

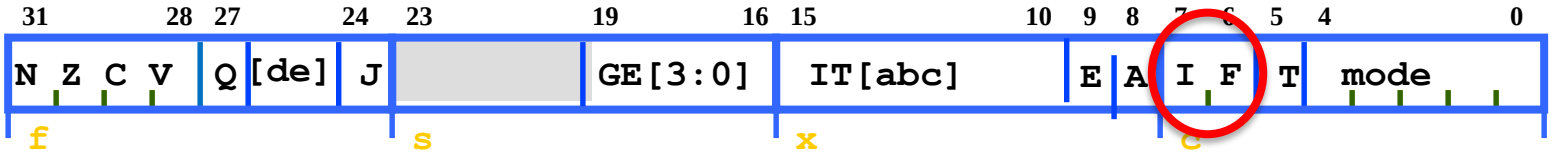
Mem. address	Contents
0x00000000	b reset_handler_routine
0x00000004	b Unexisingcode_hadeler_routine
0x00000008	b SVC_handler_routine
0x0000000C	b Abort1_handler_routine
0x00000010	b Abort2_handler_rountie
0x00000014	-
0x00000018	b IRQ_handler_routine
0x0000001C	b FIQ_hadler_routine

Exception priorities

- When multiple exceptions arise at the same time, a fixed priority system determines the order that they are handled:

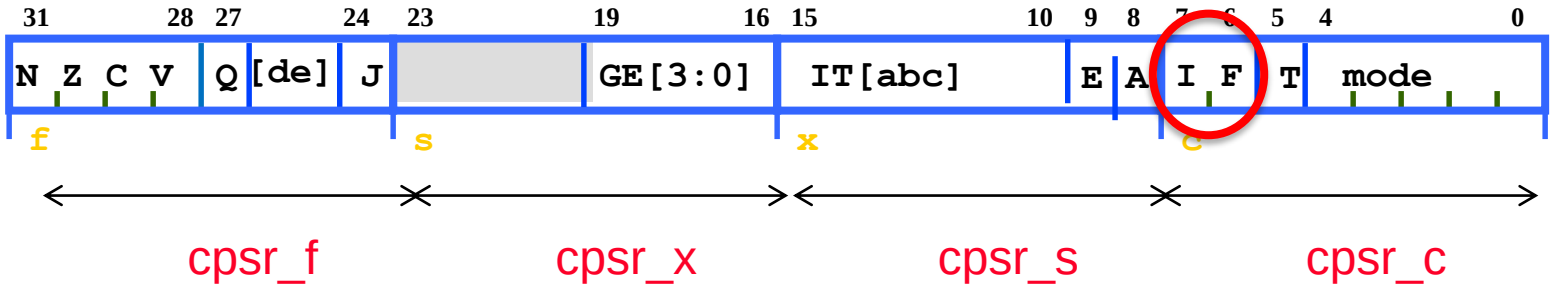
Priority		Exception
Highest	1	Reset
	2	Precise Data Abort
	3	FIQ
	4	IRQ
	5	Prefetch Abort
	6	Imprecise Data Abort
Lowest	7	Undefined Instruction SVC

Status register again: cpsr_fsxc



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Sticky Overflow flag - Q flag**
  - Indicates if saturation has occurred
- **SIMD Condition code bits – GE[3:0]**
  - Used by some SIMD instructions
- **IF THEN status bits – IT[abcde]**
  - Controls conditional execution of Thumb instructions
- **T bit**
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - J = 1: Processor in Jazelle state
- **Mode bits**
  - Specify the processor mode
- **Interrupt Disable bits**
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- **E bit**
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- **A bit**
  - A = 1: Disable imprecise data aborts

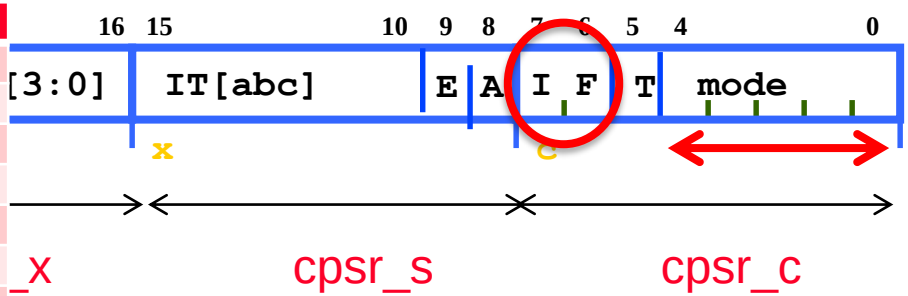
Status register again: cpsr_fsxc



- **We can modify the Current Program Status Register (CPSR) in several ways by mean of the *msr* instruction:**
  - **Every byte can be independently modified by accessing the corresponding byte (cpsr_f, cpsr_x, cpsr_s, cpsr_c):**
    - Example:
      - mov r0,#0bxxxxxxxx
      - msr cpsr_c, r0 → only the 8 LSB are modified
    - Example 2: disable interrupts IQR, FIR for Supervisor (SVC) mode and enter in supervisor mode:
      - mov r0,#0b11010011
      - msr cpsr_c, r0

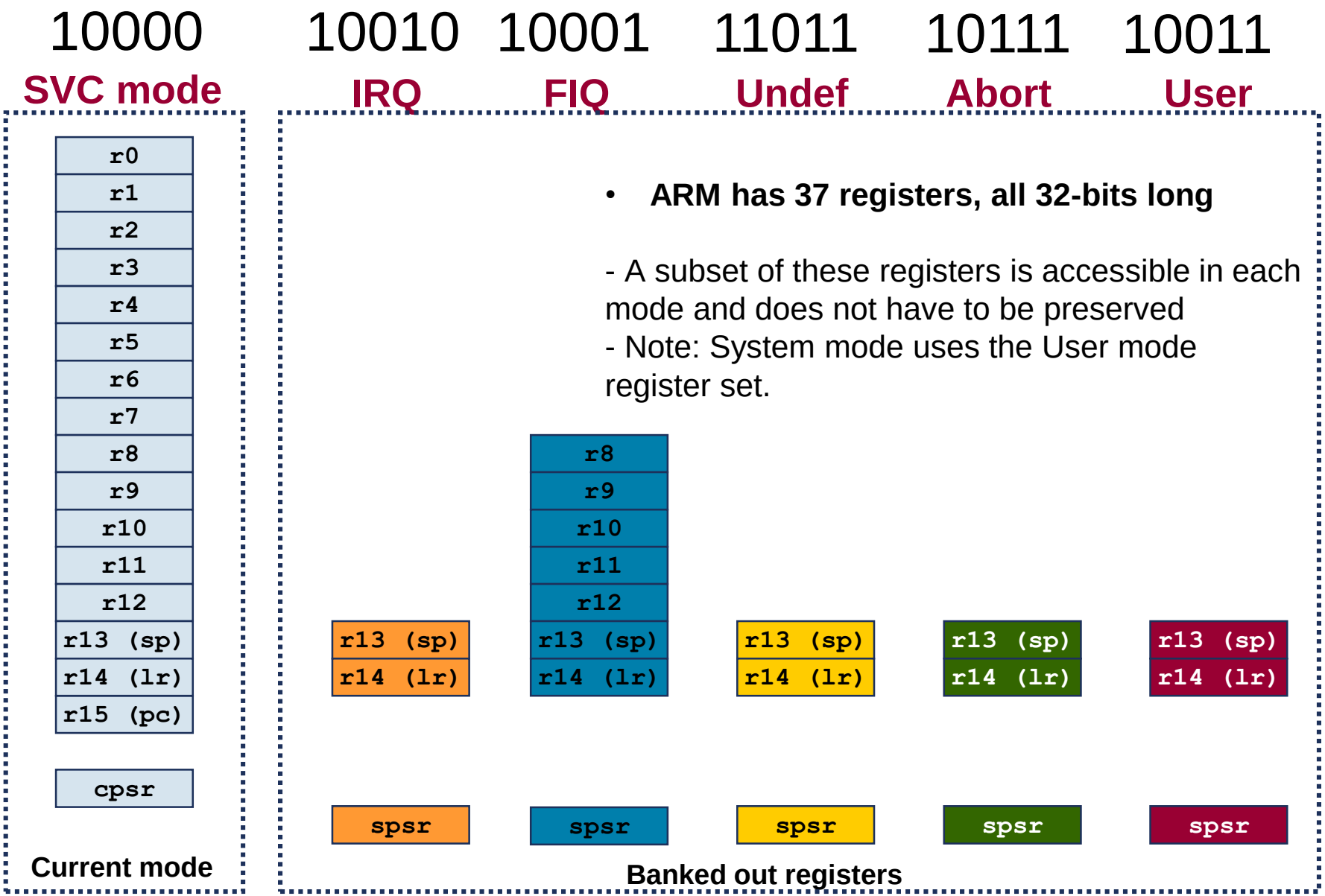
Status register again: cpsr_fsxc

Code (bits 4-0 CPSR)	Mode
10000	User
10001	Fast interrupt (FIQ)
10010	Regular interrupt (IRQ)
10011	Supervisor (SVC)
10110	Sure Monitor
10111	Abort
11011	Undefined
11111	System



- Example 2: disable interrupts IQR, FIR for Supervisor (SVC) mode and enter in supervisor mode:
  - mov r0,#0b11010011
  - msr cpsr_c, r0

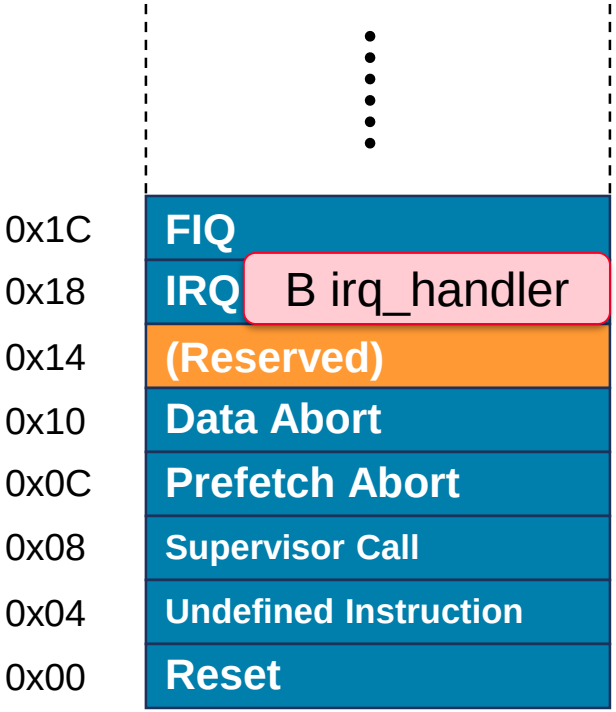
Banking of registers



Handling exceptions

CPSR = ***Current*** Status Register , ***S***PSR= ***Saved*** Status Register

- **When an exception occurs, the core…**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- **To return, exception handler needs to…**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

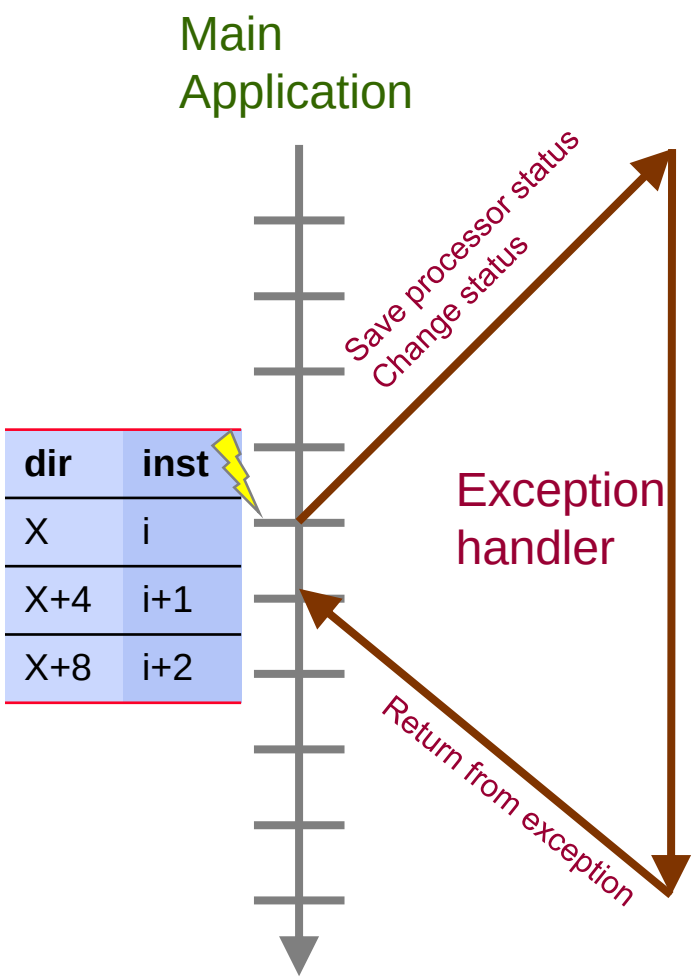


Vector Table

Vector table can also be at **`0xFFFF0000`** on most cores

This can only be done in ARM state.

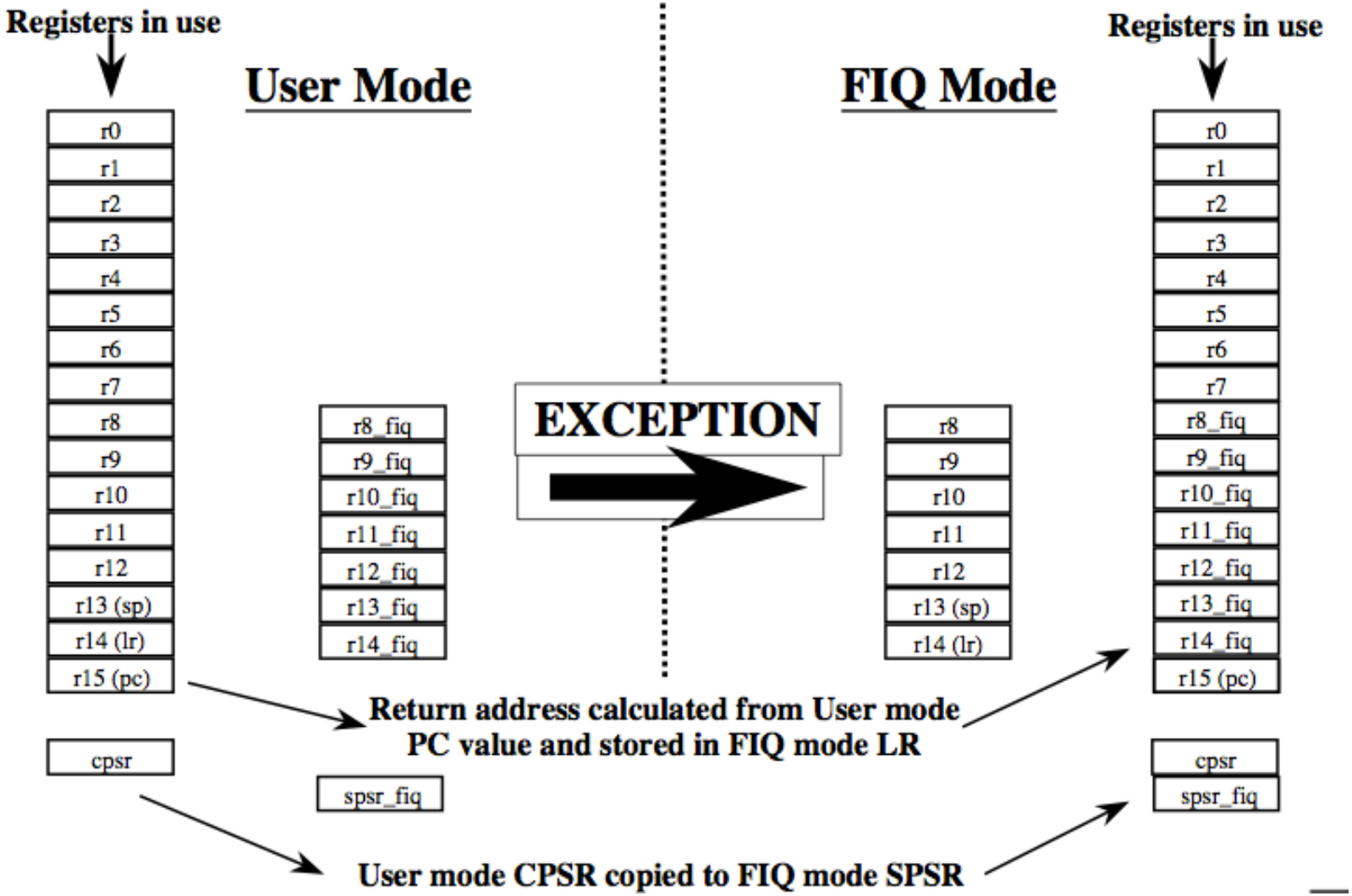
Handling exceptions (again)



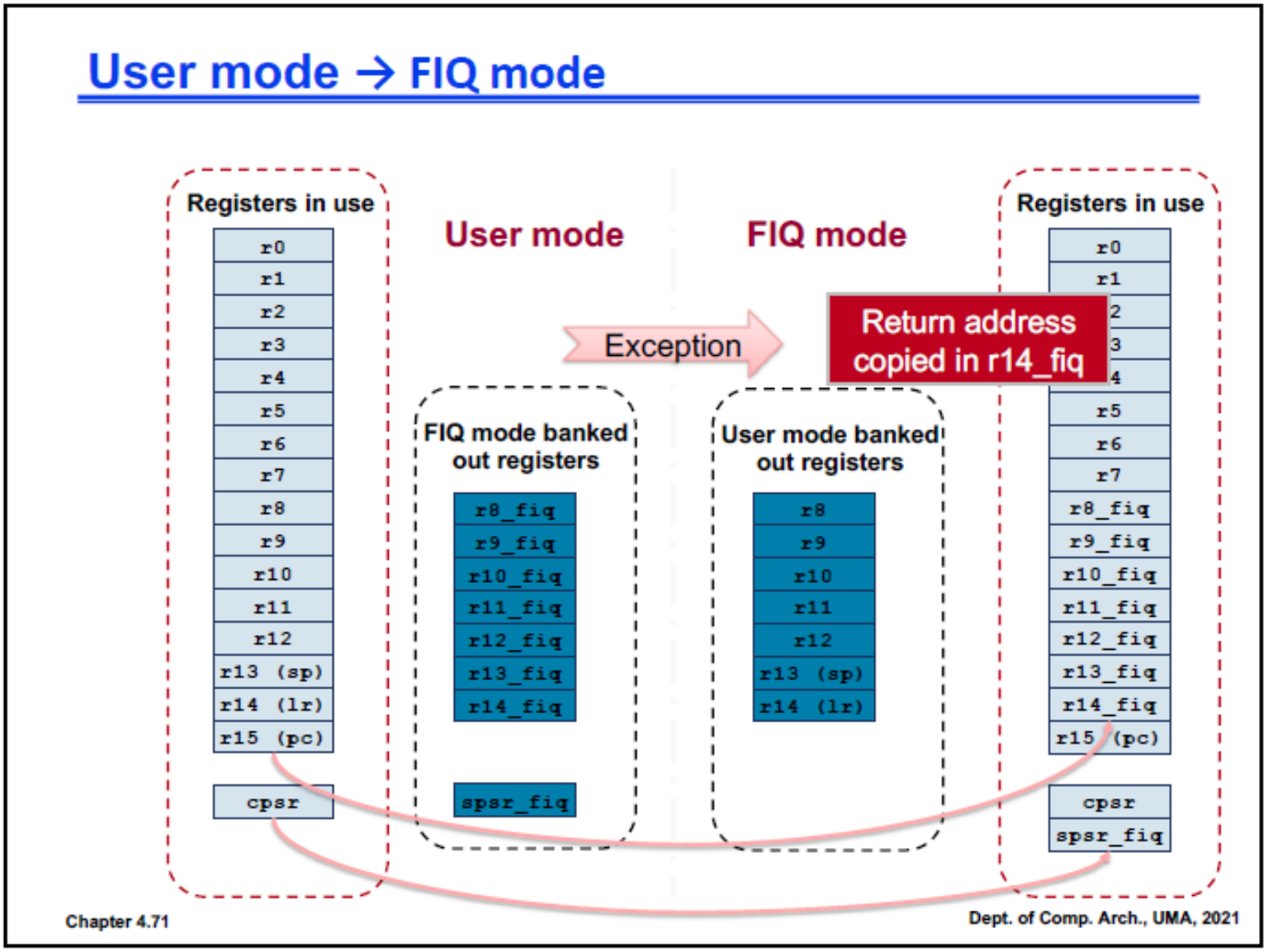
1. **Save processor status *(automatically)***
   - Stores PC in `LR_<mode>`
     - Adjusts LR based on exception type
     - Stores X+8 → `LR_<mode>`
   - Copies `CPSR` into `SPSR_<mode>`
2. **Change processor status for exception *(automatically)***
   - Forces the CPSR mode bits to a value (depends on the exception)
   - Sets PC to vector address
3. **Execute exception handler *(user software)***
   - <user code>
4. **Return to main application *(user software)***
   - Restore `CPSR` from `SPSR_<mode>`
   - Restore PC: PC ← `LR_<mode>` **`-4`**

- **1 and 2 performed automatically by the core**
- **3 and 4 responsibility of software**

Example: User Mode → FIQ Mode



Example: User Mode → FIQ Mode



Just before interrupt

```
- - -
349C   mov r4,r3,r5

34A0   add r1,r2,r3

34A4   sub r8,r6,r9
- - -
```

During the interrupt

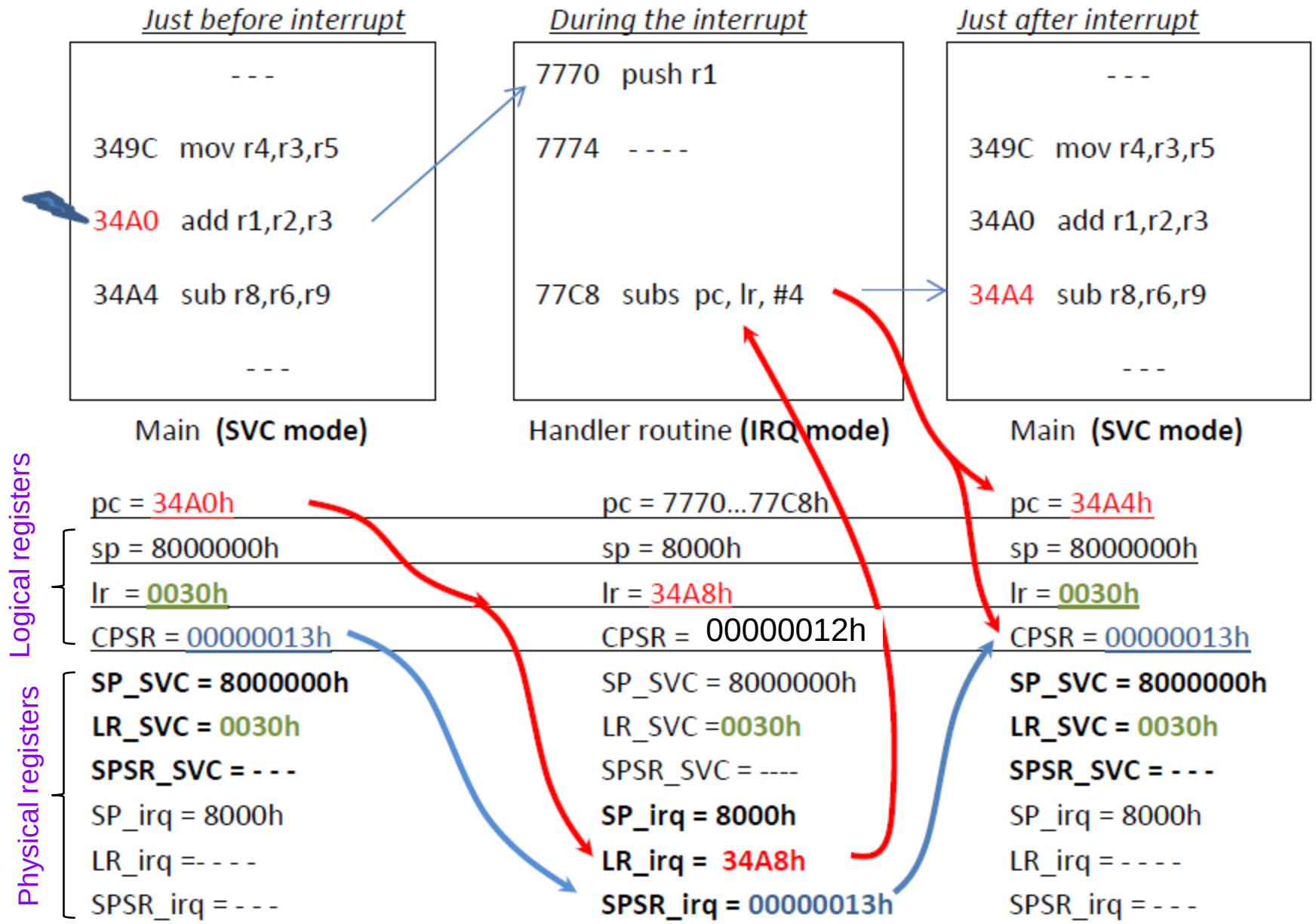
```
7770   push r1

7774   - - - -
       - - - -
       - - - -
       - - - -
77C8   subs  pc, lr, #4
```

Just after interrupt

```
- - -
349C   mov r4,r3,r5

34A0   add r1,r2,r3

34A4   sub r8,r6,r9
- - -
```

	Main **(SVC mode)**	Handler routine **(IRQ mode)**	Main **(SVC mode)**
Logical registers	pc = 34A0h	pc =	pc =
	sp = 8000000h	sp =	sp =
	lr = **0030h**	lr =	lr =
	CPSR = 00000013h	CPSR =	CPSR = 00000013h
Physical registers	**SP_SVC = 8000000h**	SP_SVC = 8000000h	**SP_SVC = 8000000h**
	LR_SVC = 0030h	LR_SVC =**0030h**	**LR_SVC = 0030h**
	SPSR_SVC = - - -	SPSR_SVC = ----	**SPSR_SVC = - - -**
	SP_irq = 8000h	**SP_irq = 8000h**	SP_irq = 8000h
	LR_irq =- - - -	**LR_irq =**	LR_irq = - - - -
	SPSR_irq = - - -	**SPSR_irq =**	SPSR_irq = - - -

Code (bits 4-0 CPSR)	Mode
10000 (10h)	User
10001 (11h)	Fast interrupt (FIQ)
10010 (12h)	Regular interrupt (IRQ)
10011 (13h)	Supervisor (SVC)
10110 (14h)	Sure Monitor


Just before interrupt
During the interrupt
Just after interrupt
- - -
349C mov r4,r3,r5
34A0 add r1,r2,r3
34A4 sub r8,r6,r9
- - -
7770 push r1
7774 - - - -
77C8 subs pc, lr, #4
- - -
349C mov r4,r3,r5
34A0 add r1,r2,r3
34A4 sub r8,r6,r9
- - -
Main (SVC mode)
Handler routine (IRQ mode)
Main (SVC mode)
Logical registers
Physical registers
pc = 34A0h
sp = 8000000h
lr = 0030h
CPSR = 00000013h
SP_SVC = 8000000h
LR_SVC = 0030h
SPSR_SVC = - - -
SP_irq = 8000h
LR_irq =- - - -
SPSR_irq = - - -
pc = 7770...77C8h
sp = 8000h
lr = 34A8h
CPSR = 00000012h
SP_SVC = 8000000h
LR_SVC =0030h
SPSR_SVC = ----
SP_irq = 8000h
LR_irq = 34A8h
SPSR_irq = 00000013h
pc = 34A4h
sp = 8000000h
lr = 0030h
CPSR = 00000013h
SP_SVC = 8000000h
LR_SVC = 0030h
SPSR_SVC = - - -
SP_irq = 8000h
LR_irq = - - - -
SPSR_irq = - - -

Exception handler

- Basic structure of a exception handler
  - Interruption: the return is done by lr-4
  - Internal exception (as data abort): the return is done by lr-8

    Handler routine:

❶ Push registers to be used: `push {…, …}`
❷ Source of interruption?
❸ Perform handler work depending on ❷
❹ Clear event (notify to device IRQ/FIQ has been served)
❺ Pop registers: `pop {…, …}`
❻ Return from handler: `subs pc, lr, #4`

  - User must manage A, I and F flags to disable/enable nesting of new exceptions and interruptions.
    - Initially the interruptions are disabled (I=F=1).

❷ Source of interruption?

- In case of interruption, the handler must identify the source reading the IRQ pending ports
  - GPIO interruption detection: use GPEDSn.

```
❷ Source of interruption?.  Check if push-button  (2) was pressed
ldr     r0, =GPBASE
ldr     r2, [r0, #GPEDS0]
ands    r2, #0b00000000000000000000000000000100
...
```

  - System Timer interrupt detection: STCS notifies interruption due to C0 : C3 counters

```
❷ Source of timer interruption?:
ldr     r0, =STBASE
ldr     r2, [r0, #STCS]
ands    r2, #0b0010  @C1?
...
ldr     r2, [r0, #STCS]
ands    r2, #0b1000 @C3?
```

Main program: steps to set up the Interruptions

0. Raspberry Pi 3 : Change from HYP mode to SVC mode

❶ Initialize Vector Table (IRQ/FIQ) in the Vector Table
❷ Init the stack/s for FIQ/IRQ modes

```
sp_ifq <- 0x00004000
sp_irq <- 0x00008000
```

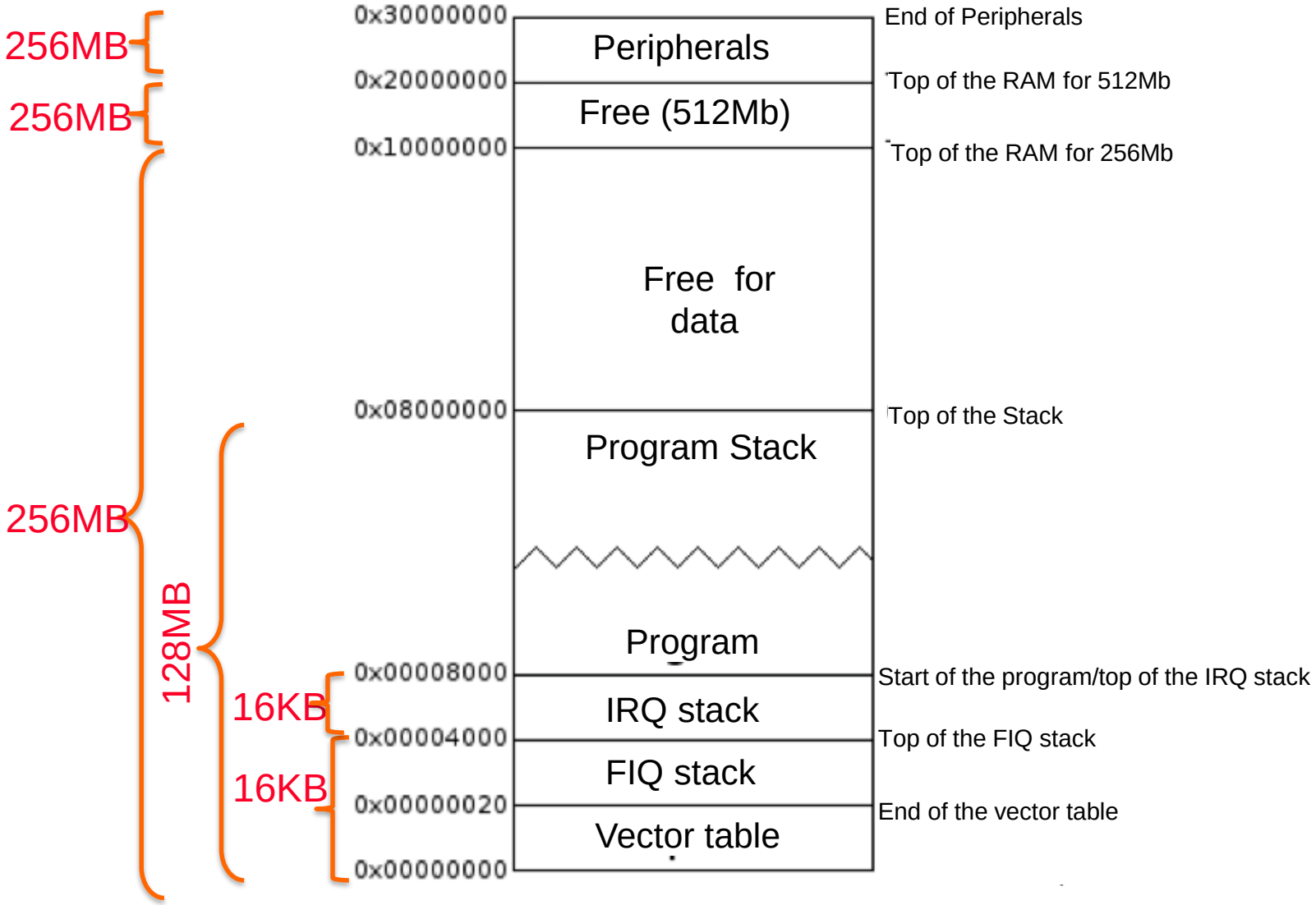
❸ Init the stack for SVC mode (SVC mode selected)

```
sp_svc <- 0x08000000
```

❹ Configure GPIOs (I&O)
❺ Configure sources of interruption: timer/push-buttons
❻ Local enabling of configured interrupts
❼ Global enabling of interrupts (SVC mode)
❽ ... (main program tasks)

Memory map

- Exception Vector Table starts at 0x00000000 (or 0xFFFF0000)



1- Initialize Vector Table

- To write in the Vector Table:
  - Example for IRQ

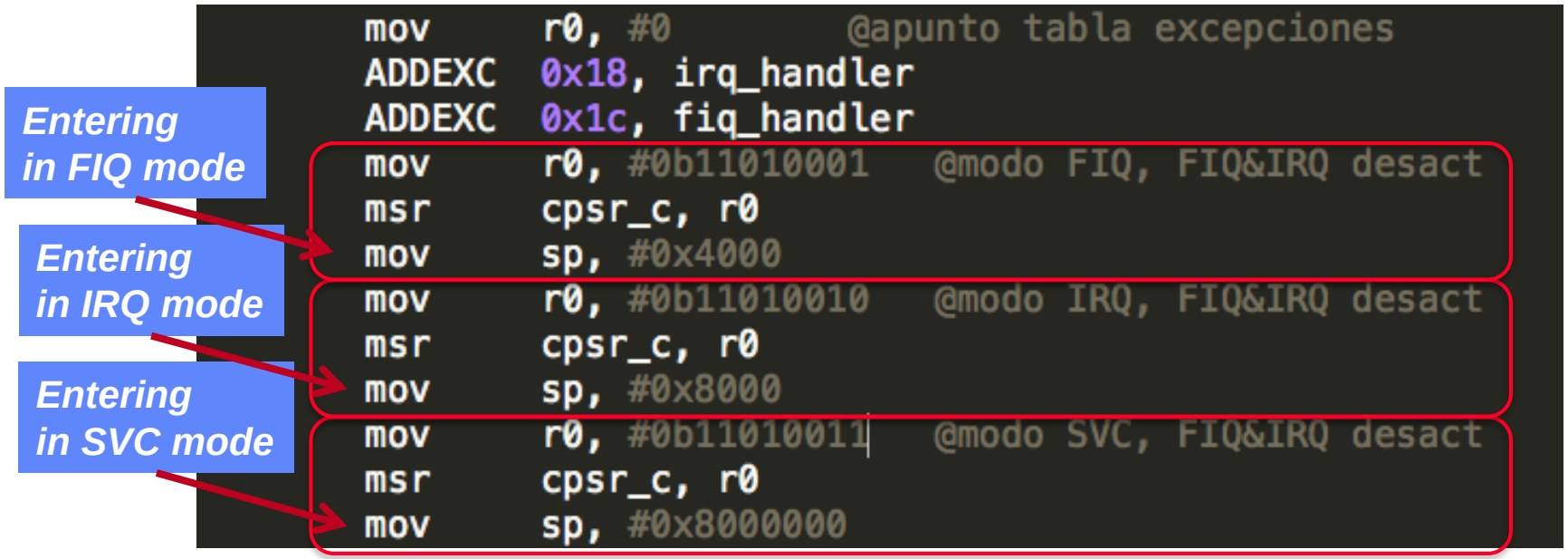
```
mov     r0, #0          @Vector table Base = 0
ADDEXC  0x18, irq_handler
```

- ADDEXC is a macro that computes the offset of the exception handler and writes the Vector in the Vector Table.

Mem. address	Contents
0x00000000	b reset_handler_routine
0x00000004	b Unexisingcode_hadeler_routine
0x00000008	b SVC_handler_routine
0x0000000C	b Abort1_handler_routine
0x00000010	b Abort2_handler_rountie
0x00000014	-
0x00000018	***b irq_handler***
0x0000001C	b FIQ_hadler_routine

2- 3 Initialize the stack (and disable interrupts)

- Each mode has its stack pointer (sp)
  - Change the mode (via cpsr_c)
  - Instructions msr (sr <- reg) y mrs (reg <-sr).
  - Initialize the corresponding sp register
- Initial state in BareMetal is SVC
  - sp_fiq=0x4000, sp_irq=0x8000, sp_svc=0x8000000:



❺ Configure sources of interruption

- GPIO interruption (push-buttons): use GPRENn, GPFENn, GPHENn, GPLENn, GPARENn y GPAFENn

Configure push-button interruption (GPIO 2)

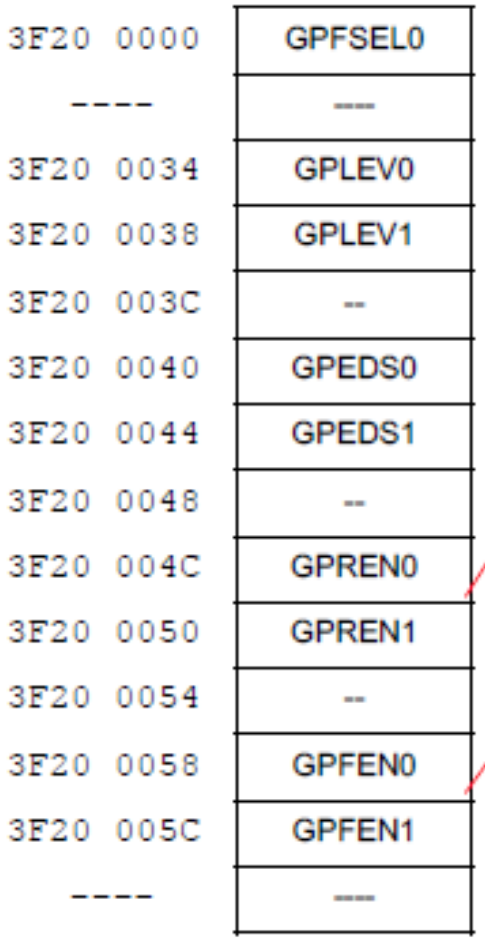
```
ldr   r0,=GPBASE
mov   r1, #0b00000000000000000000000000000100
str   r1, [r0, #GPFEN0]
```

- System Timer: write the final count (microseconds) in STC1/STC3

Configure timer IRQ

```
ldr   r0, =STBASE
ldr   r1, [r0, #STCLO]
add   r1, #y    @y microseconds
str   r1, [r0, #STC1]
```

Configure GPIO interrupt

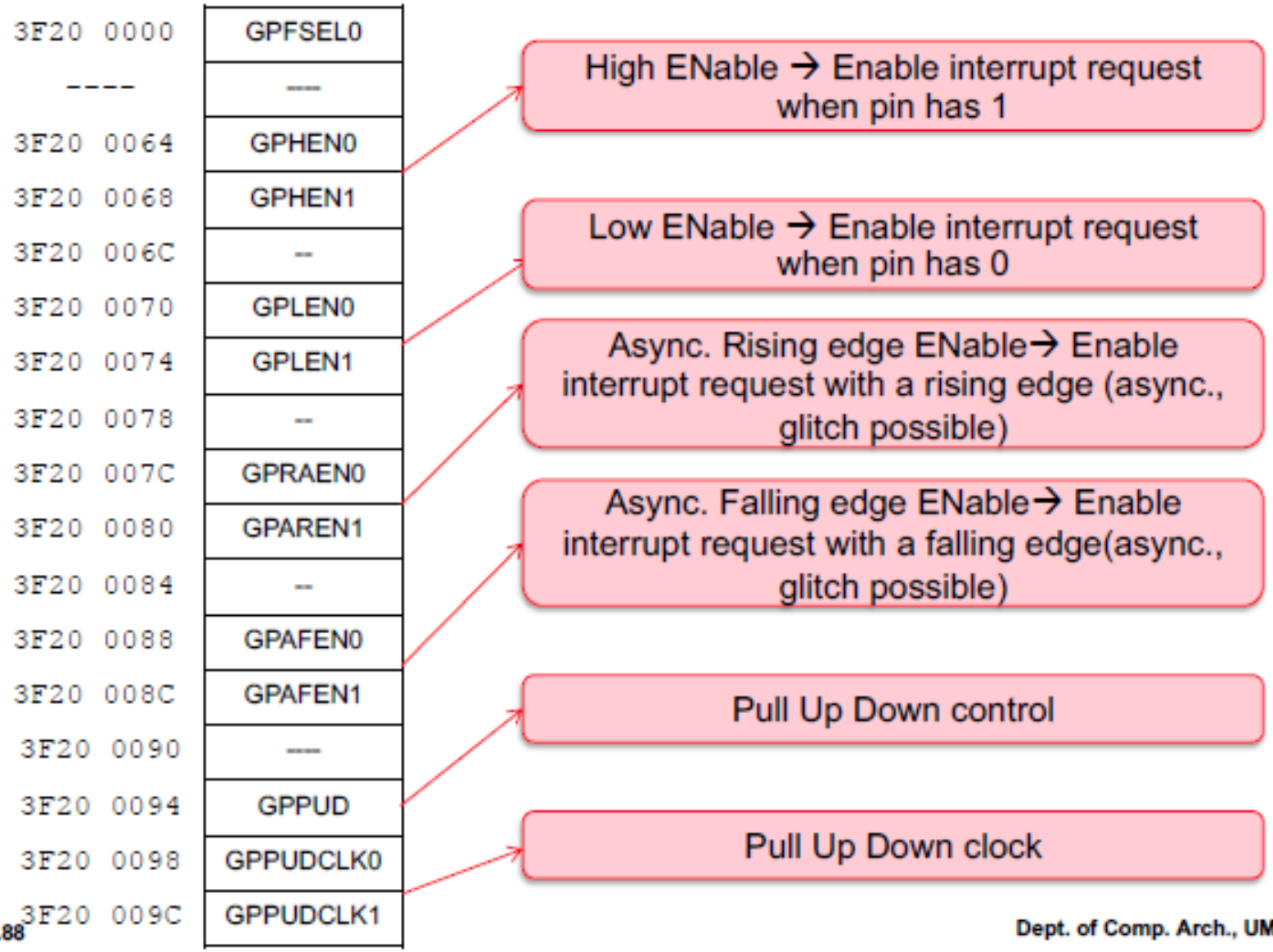


Rising edge ENable→ Enable interrupt request with a rising edge (sync., avoid glitch)

Falling edge Enable→ Enable interrupt request with a falling edge (sync., avoid glitch)

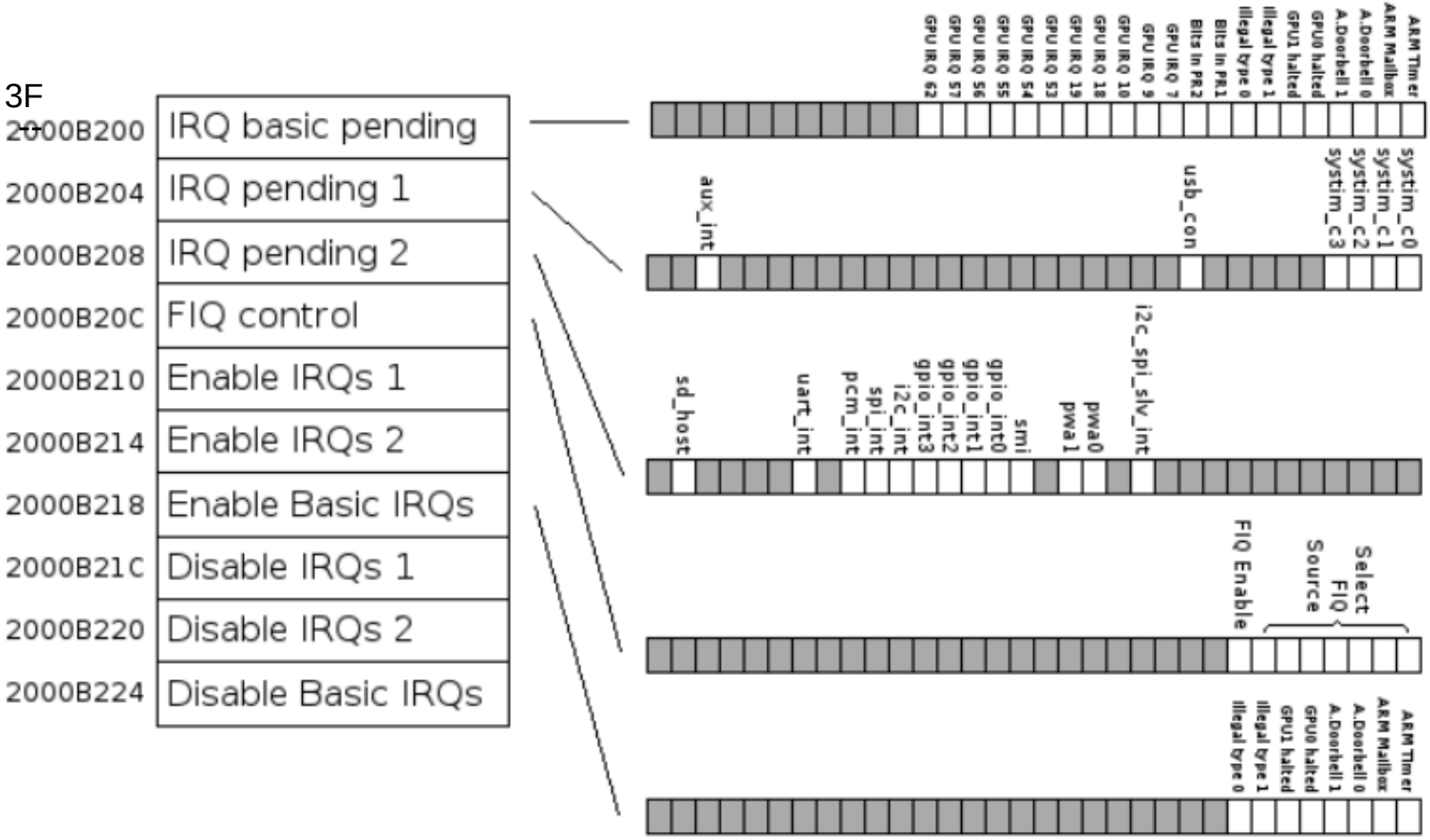
The suppression of glitches is done by sampling the pin using the system clock and then looking for a “011” (rising) or “100” (falling edge) pattern on the sampled signal.

Configure GPIO interruptons



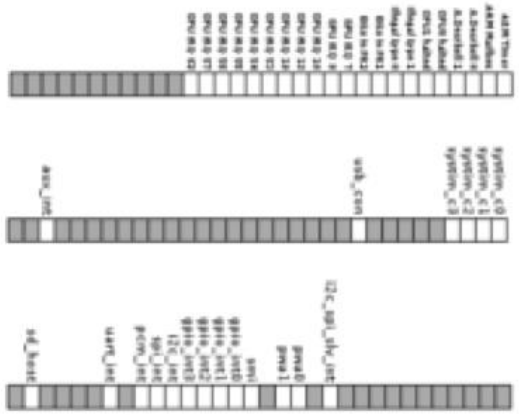
5 Configure sources of interruption

- Ports for managing interruptions



6 Configure sources of interruption cont.

Pending group	Enable Group	Disable group
IRQ basic pending	Enable Basic IRQs	Disable Basic IRQs
IRQ pending 1	Enable IRQs 1	Disable IRQs 1
IRQ pending 2	Enable IRQs 2	Disable IRQs 2

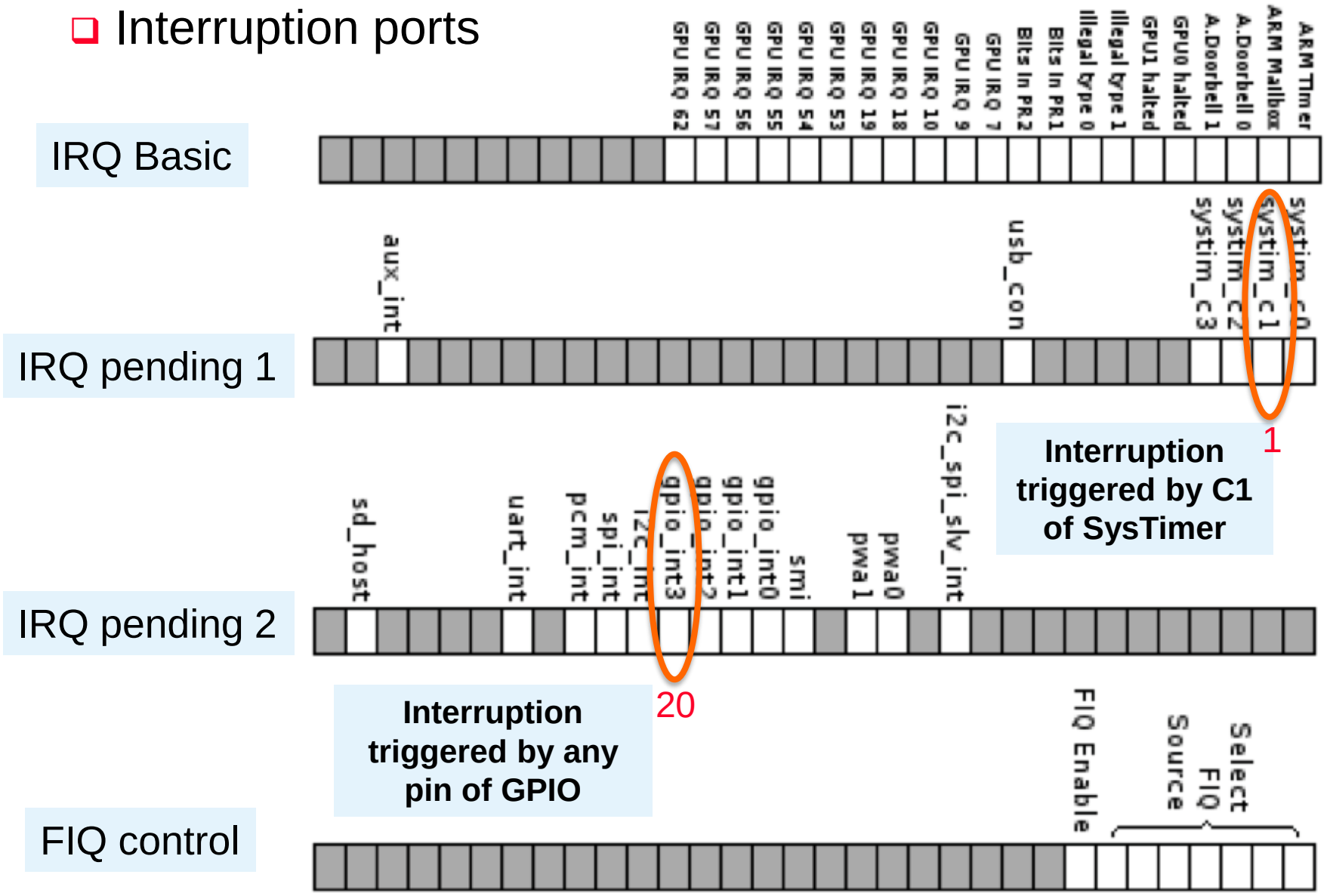


3F 2000B200	IRQ basic pending	Pending
2000B204	IRQ pending 1	
2000B208	IRQ pending 2	
2000B20C	FIQ control	
2000B210	Enable IRQs 1	Enable
2000B214	Enable IRQs 2	
2000B218	Enable Basic IRQs	
2000B21C	Disable IRQs 1	Disable
2000B220	Disable IRQs 2	
2000B224	Disable Basic IRQs	

- 3 groups for IRQ (Pending, Enable, Disable)
  - Each group
    - IRQ Basic: summary
    - IRQs 1 y IRQs 2: detail
- 1 port for FIQ
  - Control

6 Configure sources of interruption cont.

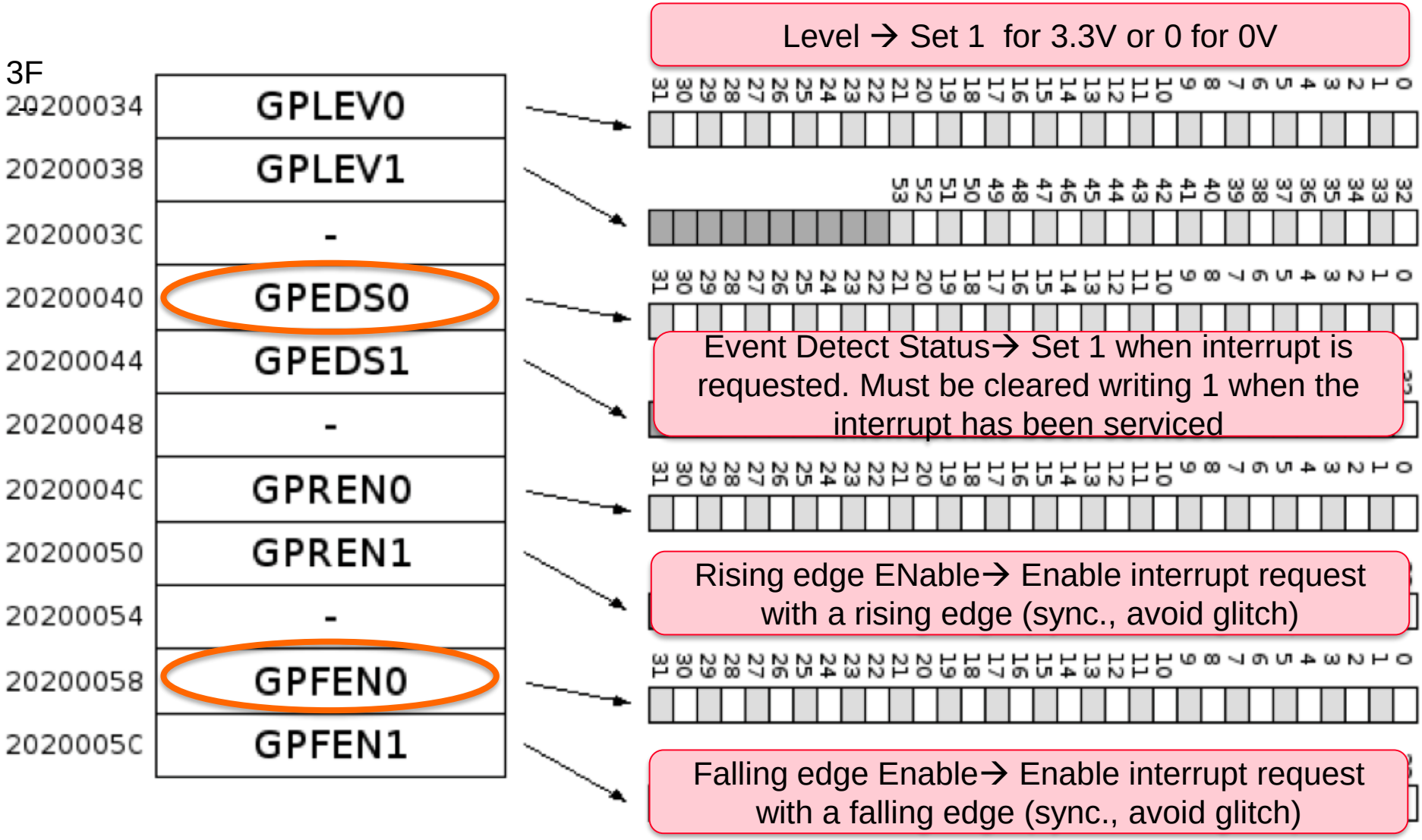
- Interruption ports



6 Configure sources of interruption cont.

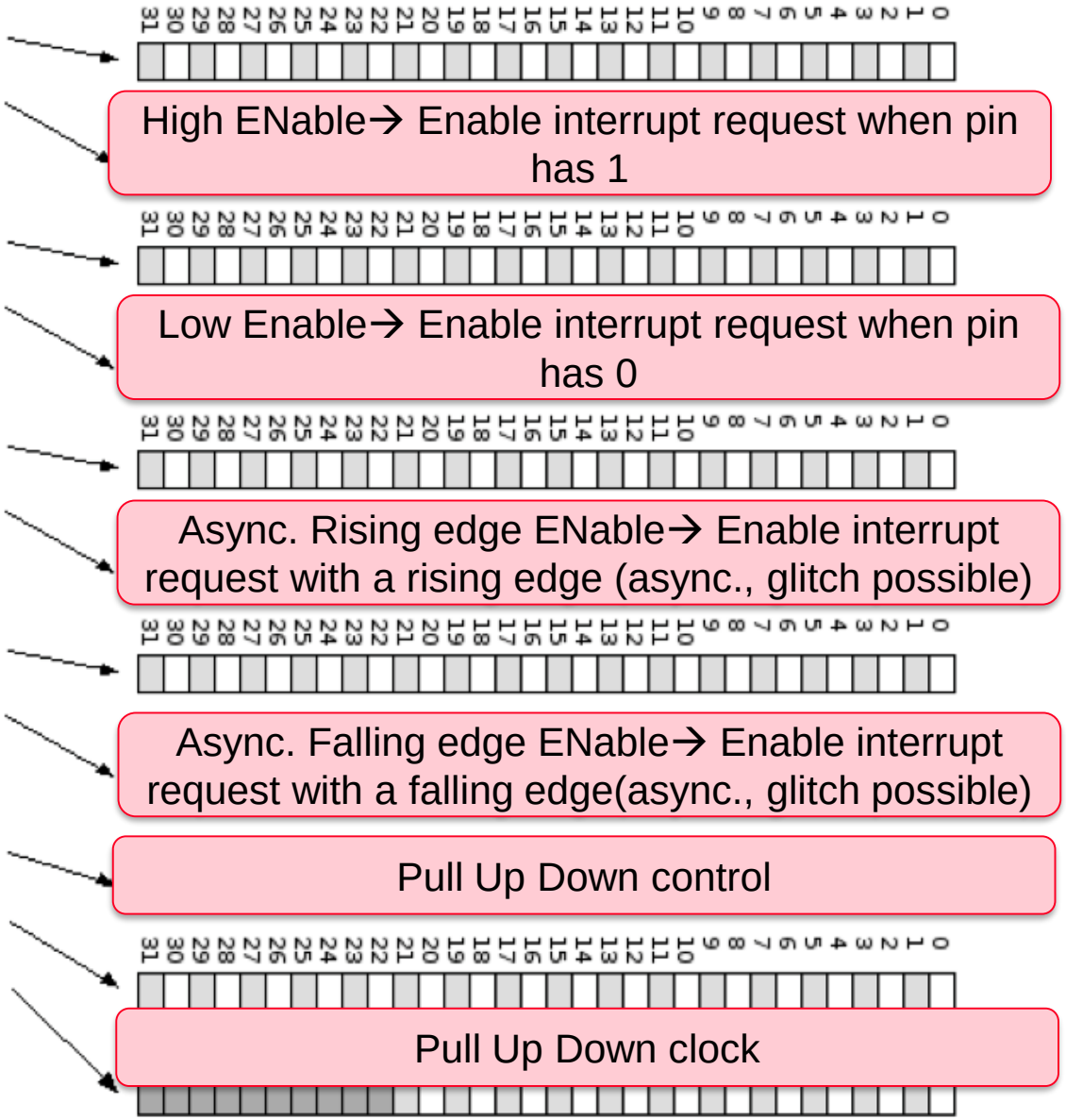
- Set the corresponding bit in the appropriate Enable IRQs port
  - GPIO interruption enable: use GPRENn, GPFENn, GPHENn, GPLENn, GPARENn y GPAFENn
  - System Timer: each counter (C0-C3) can be enabled/disabled in Enable IRQ1 / Disable IRQ1 ports.
- In case of interruption, the handler must identify the source reading the IRQ pending ports
  - GPIO interruption detection: use GPEDSn.
  - System Timer interrupt detection: STCS notifies interruption due to C0 : C3 counters

GPIO memory mapping cont.



GPIO memory mapping cont.

3F20xxxx	
20200064	GPHEN0
20200068	GPHEN1
2020006C	-
20200070	GPLEN0
20200074	GPLEN1
20200078	-
2020007C	GPAREN0
20200080	GPAREN1
20200084	-
20200088	GPAFEN0
2020008C	GPAFEN1
20200090	-
20200094	GPPUD
20200098	GPPUDCLK0
2020009C	GPPUDCLK1



❼ Global enabling of sources of interruption

- Enable interruptions globally in SVC mode (flags I / F)

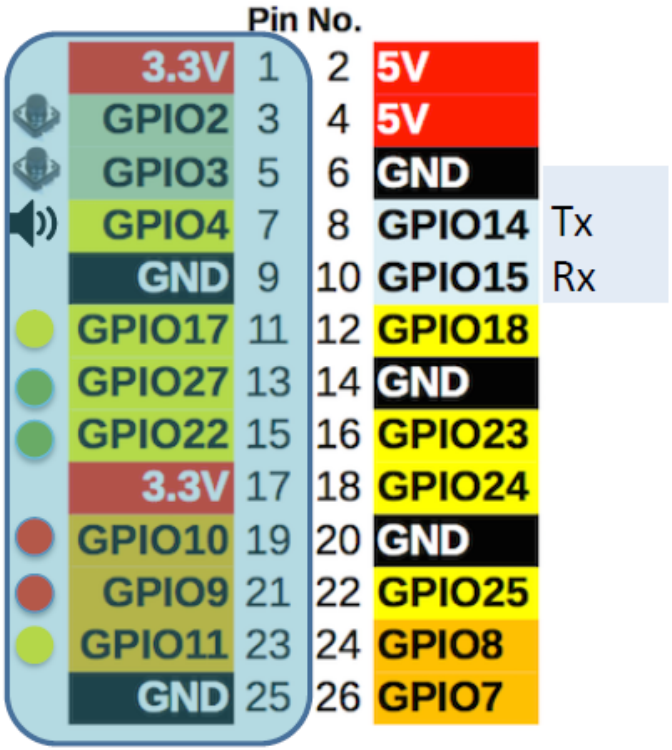
Enable IRQ (SVC mode):

```
mov    r1, #0b01010011 @ flag I
msr    cpsr_c, r1
```

Enable FIQ (SVC mode):

```
mov    r1, #0b10010011 @ flag F
msr    cpsr_c, r1
```

Rpi cheat sheet



Code structure:

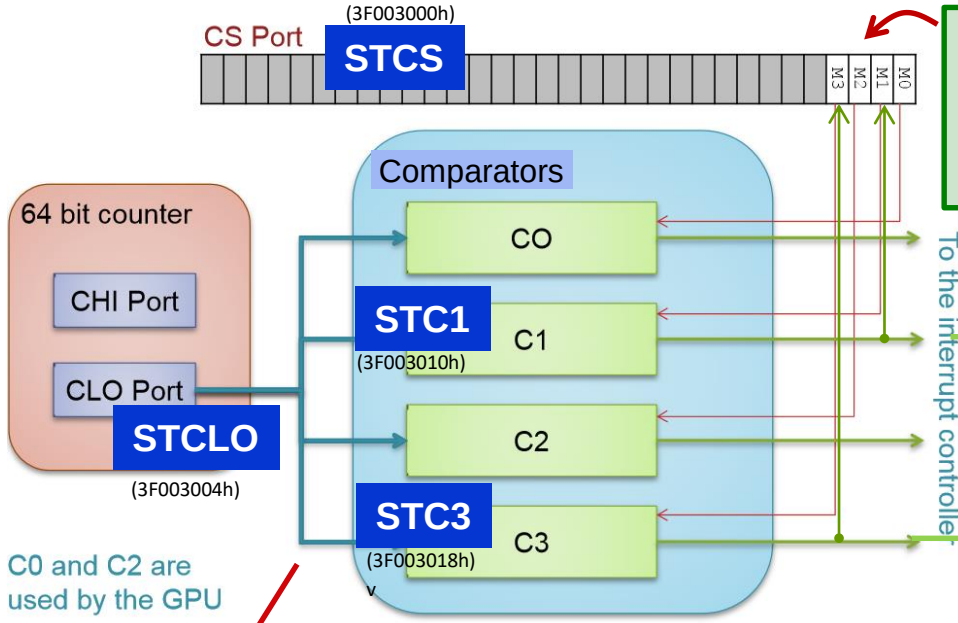
Main program (.text):

1. Initialize Vector Table (IRQ/FIQ)
2. Init the stack/s for FIQ/IRQ modes
3. Init the stack for SVC mode (SVC mode selected)
4. Configure GPIOs (I&O)
5. Configure peripheral interruption: timer/push-buttons)
6. Local enabling of configured interrupts
7. Global enabling of interrupts (SVC mode)
8. Infinite loop (polling of device/s?)

IRQ/FIQ Handler:

1. Push registers to be used
2. Source of interruption?
3. Perform handler work depending on 2
4. Clear event (notify to device IRQ/FIQ has been served)
5. Pop registers
6. Return from handler

Regular Interrupts by timer *(using comparator C1)*



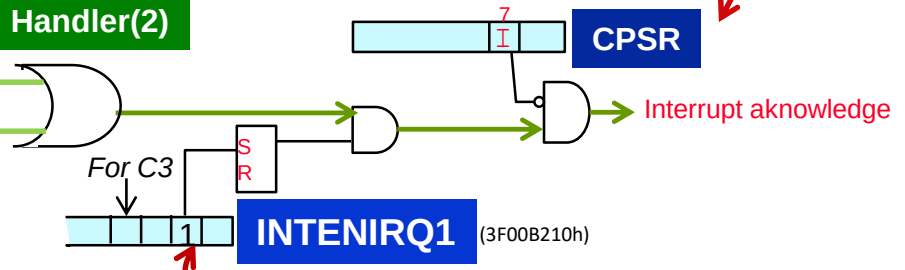
Source of interrupt ?

```
ldr     r0, =STBASE
ldr     r2, [r0, #STCS]
tst     r2,#0b0010(1000 for C3)
```

Handler(2)

Enable IRQ (SVC mode):

```
mov     r1, #0b01010011
msr     cpsr_c, r1
```



Enable timer IRQ by comparator C1:

```
ldr     r0,=INTBASE
mov     r1, #0b0010 @(#0b1000 for C3)
str     r1,[r0,#INTENIRQ1]
```

Main(6)

Preparing C1 to send an interrupt after "*y*" microseconds

```
ldr     r0, =STBASE
ldr     r1, [r0, #STCLO]
add     r1, #y      @y microseconds
str     r1, [r0, #STC1] @#STC3 for C3
```

Main(5) and/or handler(3)

Reset (end) of the timer interrupt by comparator C1 *(to allow a new interrupt)*:

```
ldr     r0, =STBASE
mov     r1, #0b0010   @(#0b1000 for C3)
str     r1,[r0,#STCS]
   (insert just here for rearming the timer)
subs    pc, lr, #4    @to return to main
```

Handler(4&6))

Main(7)

Example 7: turn on a red led after 4 seconds

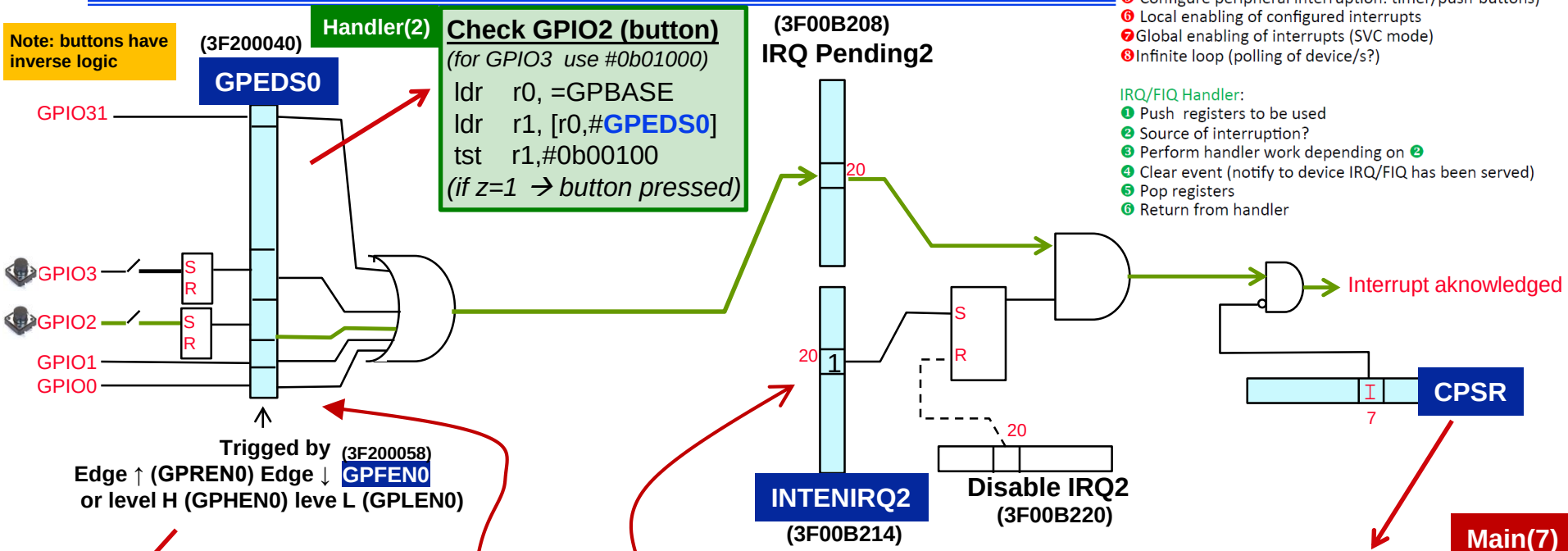
```
            .includes "inter.inc"

.text
            mov     r0, #0                                  Initialize vector table   ❶
            ADDEXC  0x18, irq_handler

            . . .

            ldr     r0, =GPBASE
            ldr     r1, =0b00001000000000000000000000000000  Set GPIO9 as Output      ❹
            str     r1, [r0, #GPFSEL0]

            ldr     r0, =STBASE
            ldr     r1, [r0, #STCLO]                        Load CLO, add 4 sec       ❺
            add     r1, #0x400000   @ 4.19 seconds          and store result in C1
            str     r1, [r0, #STC1]

            ldr     r0, =INTBASE
            mov     r1, #0b0010                             Enable C1 interruption    ❻
            str     r1, [r0, #INTENIRQ1]

            mov     r0, #0b01010011 @ SVC mode, IRQ enabled Enable I flag             ❼
            msr     cpsr_c, r0
buc:        b       buc

irq_handler:
            push    {r0, r1}                                                          ❶
            ldr     r0, =GPBASE
            mov     r1, #0b00000000000000000000001000000000  Turn on RED led (GPIO9)  ❸
            str     r1, [r0, #GPSET0]
            pop     {r0, r1}                                                          ❺
            subs    pc, lr, #4                              PC← LR - 4                ❻
```

Regular int. (IRQ) by push buttons

Main program (.text):
1. Initialize Vector Table (IRQ/FIQ)
2. Init the stack/s for FIQ/IRQ modes
3. Init the stack for SVC mode (SVC mode selected)
4. Configure GPIOs (I&O)
5. Configure peripheral interruption: timer/push-buttons)
6. Local enabling of configured interrupts
7. Global enabling of interrupts (SVC mode)
8. Infinite loop (polling of device/s?)

IRQ/FIQ Handler:
1. Push registers to be used
2. Source of interruption?
3. Perform handler work depending on 2
4. Clear event (notify to device IRQ/FIQ has been served)
5. Pop registers
6. Return from handler

Note: buttons have inverse logic



Handler(2)

Check GPIO2 (button)

(for GPIO3 use #0b01000)

```
ldr   r0, =GPBASE
ldr   r1, [r0,#GPEDS0]
tst   r1,#0b00100
```

(if z=1 → button pressed)

GPED0 Trigged by falling edge

(for both GPIO2 & 3)

```
ldr   r0, =GPBASE
mov   r1, #0b011100
str   r1,[r0,#GPFEN0]
```

Main(5)

Enable local interrupt IRQ

```
ldr   r0, =INTBASE
ldr   r1, =0x00100000
str   r1,[r0,#INTENIRQ2]
```

Main(6)

Main(7)

Enable global interrupt IRQ

(for SVC mode)

```
mov   r1, #0b01010011
msr   cpsr_c, r1
```

Reset (end) of interrupt IRQ2 (GPIO2&3)

(to allow a new interrupt via GPIO2&3):

```
ldr   r0, =GPBASE
mov   r1, #0b011100
str   r1,[r0,#GPEDS0]
  ...   .............
subs  pc, lr, #4  @Return
```

Handler(4&6)

Example 10: turn on a red led after pushing a button

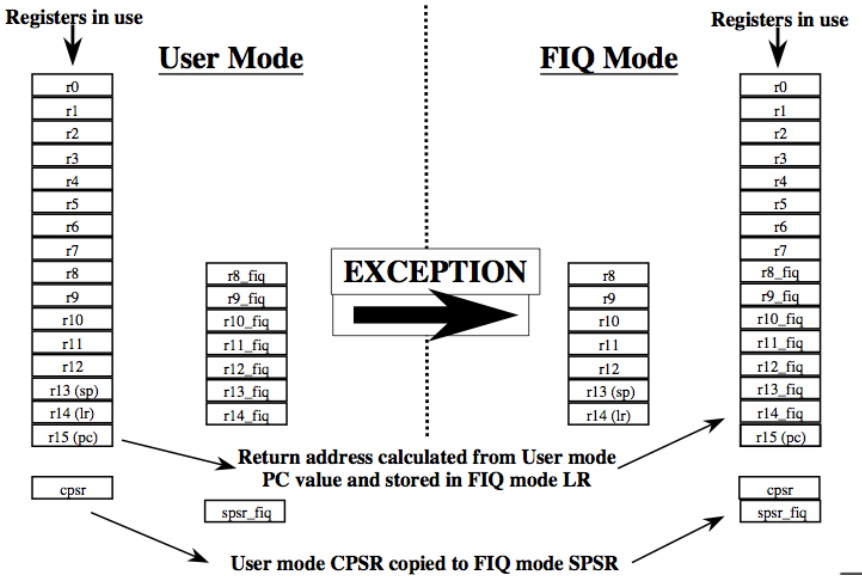
```
.include  "inter.inc"
.text
          mov     r0, #0                                                    ❶
          ADDEXC  0x18, irq_handler
          mov     r0, #0b11010010                                           ❷
          msr     cpsr_c, r0          Stack init for IRQ mode
          mov     sp, #0x8000
          mov     r0, #0b11010011                                           ❸
          msr     cpsr_c, r0          Stack init for SVC mode
          mov     sp, #0x8000000
          ldr     r0, =GPBASE
          mov     r1, #0b00001000000000000000000000000000                   ❹
          str     r1, [r0, #GPFSEL0]  Set GPIO9 as output
          mov     r1, #0b00000000000000000000000000000100                   ❺
          str     r1, [r0, #GPFEN0]   Configure falling edge interruptions through GPIO2
          ldr     r0, =INTBASE
          mov     r1, #0b00000000000100000000000000000000                   ❻
          str     r1, [r0, #INTENIRQ2] Allow interruptions from any GPIO pin
          mov     r0, #0b01010011                                           ❼
          msr     cpsr_c, r0          Set SVC mode with IRQ enabled
buc:      b       buc                                                       ❽
```

Example 10: IRQ handler

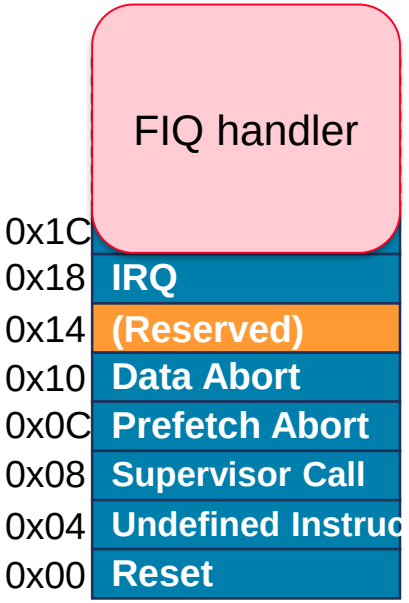
```
irq_handler:
        push    {r0, r1, r2}                                        1
        ldr     r0, =GPBASE
        ldr     r2, [r0, #GPEDS0]                                   2  Check GPIO2 was pressed
        ands    r2, #0b00000000000000000000000000000100
        movne   r1, #0b00000000000000000000001000000000
        strne   r1, [r0, #GPSET0]                                   3  Turn on GPIO9 red led
        movne   r1, #0b00000000000000000000000000000100
        strne   r1, [r0, #GPEDS0]                                   4  Clear GPIO2 event
        pop     {r0, r1, r2}                                        5
        subs    pc, lr, #4                                          6
```

Advantages of using FIQ

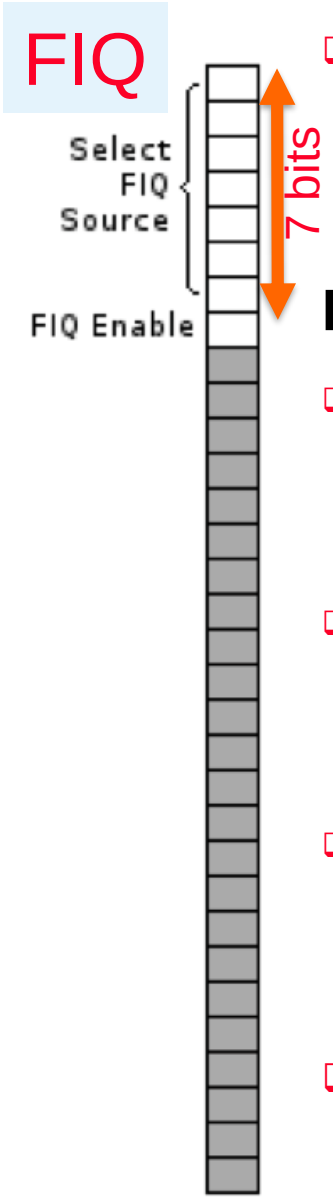
- We don't need to save context for r8 to r14
  - Safe use of these registers inside the handler (no lateral effects)



- The handler can starts in 0x1C (there is not interrupt vectors after this position)



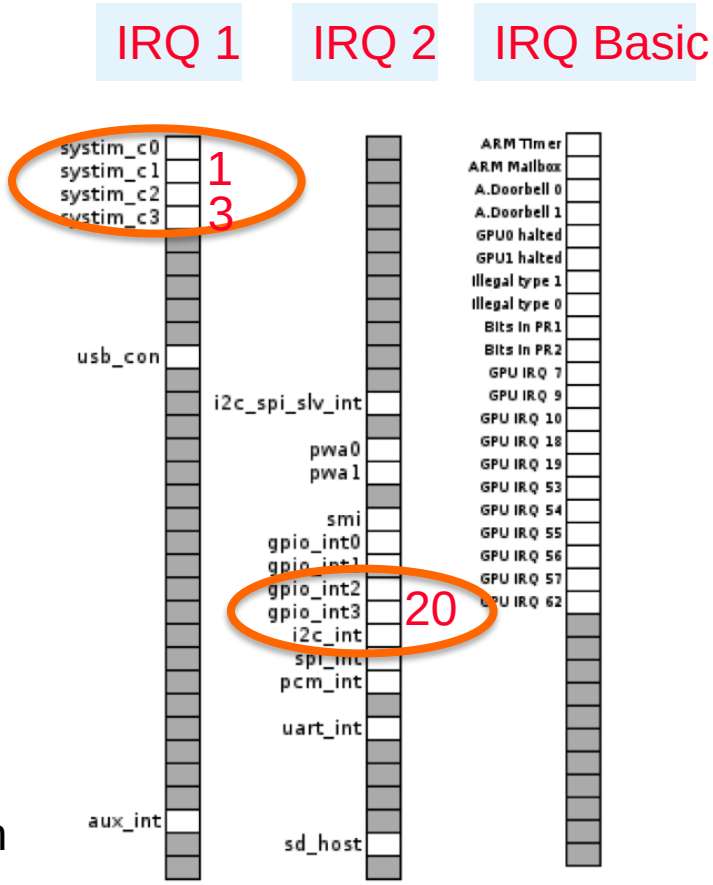
Using FIQ



- Select FIQ Source = 7 bits → 128 sources
  - 0-31 represent 32 interruption sources of IRQ 1
  - 32-63 represent 32 interruption sources of IRQ 2
  - 64-95 represent 32 interruption sources of IRQ basic

Examples:

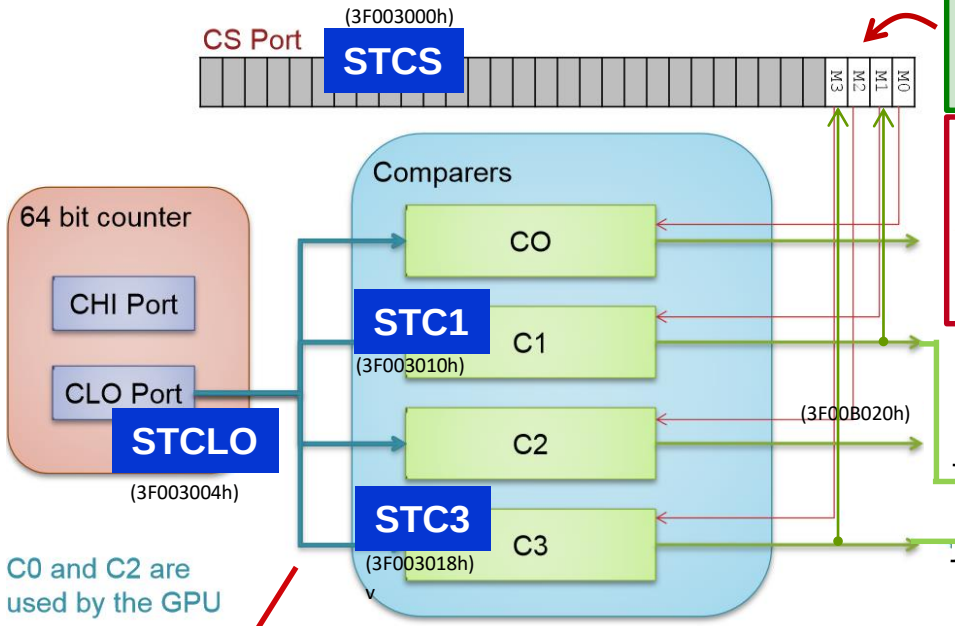
- Enable FIQ for C1 of SysTimer
  - Bit 1 of IRQ1 → Code 1
  - Also 1 in FIQ Enable
  - Result: 0b10000001 → 0x81
- Enable FIQ for C3 of SysTimer
  - Bit 3 of IRQ1 → Code 3
  - Also 1 in FIQ Enable
  - Result: 0b10000011 → 0x83
- Enable FIQ for GPIO_int3
  - Bit 20 of IRQ2 → Code 20+32
  - Also 1 in FIQ Enable
  - Result: 0b10110100 → 0xB4
- Disadvantage:
  - Just one source interruption can be enabled!



Example 11: Putting it all together

- Turn on and off the red led at GPIO9 every 4 seconds **or** when the push button at GPIO2 is pressed.
- Use IRQ to handle the timer and FIQ to handle the push button.
- Use a variable in memory to control the led state (on or off)

Fast Interrupts by timer *(using comparator C3)*



Handler(2)

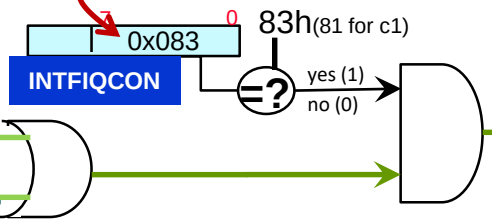
Source of interrupt ?

```
ldr   r0, =STBASE
ldr   r2, [r0, #STCS]
```

Main(6)

Enable FIQ

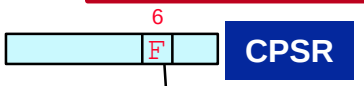
```
ldr   r0, =INTBASE
ldr   r1, =0x083
str   r1, [r0, #INTFIQCON
```



Main(7)

Enable FIQ (SVC mode):

```
mov   r1, #0b10010011
msr   cpsr_c, r1
```



Interrupt aknowledge

Main program (.text):
1. Initialize Vector Table (IRQ/FIQ)
2. Init the stack/s for FIQ/IRQ modes
3. Init the stack for SVC mode (SVC mode selected)
4. Configure GPIOs (I&O)
5. Configure peripheral interruption: timer/push-buttons)
6. Local enabling of configured interrupts
7. Global enabling of interrupts (SVC mode)
8. Infinite loop (polling of device/s?)

IRQ/FIQ Handler:
1. Push registers to be used
2. Source of interruption?
3. Perform handler work depending on 2
4. Clear event (notify to device IRQ/FIQ has been served)
5. Pop registers
6. Return from handler

Preparing C3 to send an interrupt after "*y*" microseconds

```
ldr   r0, =STBASE
ldr   r1, [r0, #STCLO]
add   r1, #y     @y microseconds
str   r1, [r0, #STC3] @#STC1 for C1
```

Main(5) and/or handler(3)

Reset (end) of the timer interrupt by comparator C3 *(to allow a new interrupt)*:

```
ldr   r0, =STBASE
mov   r1, #0b01000 @ #0b0010 for C1
str   r1,[r0,#STCS]
   (insert just here for rearming the timer)
subs  pc, lr, #4
```

Handler(4&6)

Fast interrupts by push buttons

Main program (.text):
1. Initialize Vector Table (IRQ/FIQ)
2. Init the stack/s for FIQ/IRQ modes
3. Init the stack for SVC mode (SVC mode selected)
4. Configure GPIOs (I&O)
5. Configure peripheral interruption: timer/push-buttons)
6. Local enabling of configured interrupts
7. Global enabling of interrupts (SVC mode)
8. Infinite loop (polling of device/s?)

IRQ/FIQ Handler:
1. Push registers to be used
2. Source of interruption?
3. Perform handler work depending on 2
4. Clear event (notify to device IRQ/FIQ has been served)
5. Pop registers
6. Return from handler

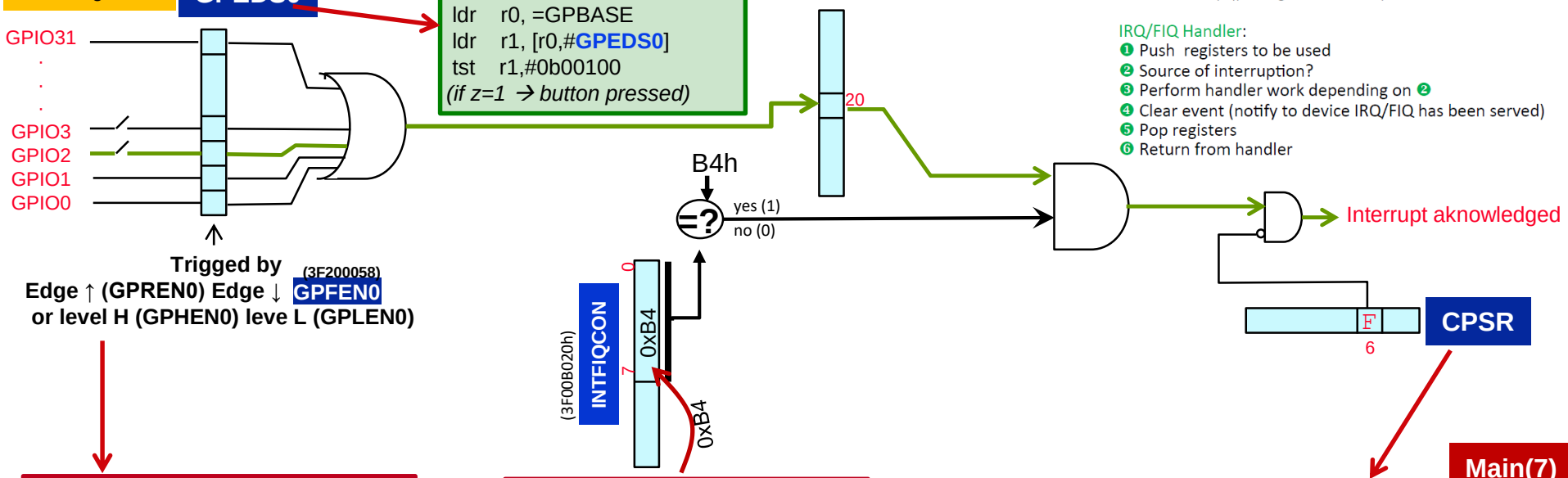
Note: buttons have inverse logic

(3F200040) **GPEDS0**

Handler(2)

Check GPIO2 (button)
(for GPIO3 use #0b01000)
```
ldr   r0, =GPBASE
ldr   r1, [r0,#GPEDS0]
tst   r1,#0b00100
```
(if z=1 → button pressed)

(3F00B208) **IRQ Pending2**



Trigged by
Edge ↑ (GPREN0) Edge ↓ (3F200058) GPFEN0
or level H (GPHEN0) leve L (GPLEN0)

Trigged by falling edge
(for both GPIO2 & 3)
```
ldr   r0, =GPBASE
mov   r1, #0b01100
str   r1,[r0,#GPFEN0]
```
Main(5)

Enable local FIQ **Main(6)**
```
ldr   r0, =INTBASE
ldr   r8, =0x0B4
str   r8, [r0, #INTFIQCON]
```

Main(7)

Enable global interrupt FIQ
(for SVC mode)
```
mov   r1, #0b10010011
msr   cpsr_c, r1
```

Reset (end) of interrupt IRQ2 (GPIO2&3)
(to allow a new interrupt):
```
ldr   r0, =GPBASE
mov   r1, #0b01100
str   r1,[r0,#GPEDS0]
.....
subs  pc, lr, #4
```
Handler (4&6)

Example 11: Timer in IRQ and push button in FIQ

```
.include  "inter.inc"
.text
            ADDEXC  0x18, irq_handler
            ADDEXC  0x1c, fiq_handler                                   ❶
            mov     r0, #0b11010001
            msr     cpsr_c, r0                 Stack init for FIQ mode  ❷
            mov     sp, #0x4000
            mov     r0, #0b11010010
            msr     cpsr_c, r0                 Stack init for IRQ mode  ❷
            mov     sp, #0x8000
            mov     r0, #0b11010011
            msr     cpsr_c, r0                 Stack init for SVC mode  ❸
            mov     sp, #0x8000000
            ldr     r0, =GPBASE
            mov     r1, #0b00001000000000000000000000000000   Set GPIO9 as output  ❹
            str     r1, [r0, #GPFSEL0]
            mov     r1, #0b00000000000000000000000000000100
            str     r1, [r0, #GPFEN0]          Configure FE ints through GPIO2  ❺
            ldr     r0, =STBASE
            ldr     r1, [r0, #STCLO]           Program timer to
            add     r1, #0x400000              interrupt in 4 seconds   ❺
            str     r1, [r0, #STC1]
            ldr     r0, =INTBASE
            mov     r1, #0b00000010            Enable C1 interruption   ❻
            str     r1, [r0, #INTENIRQ1]
            mov     r1, #0b10110100
            str     r1, [r0, #INTFIQCON]       Enable FIQ for GPIO_int3  ❻
            mov     r0, #0b00010011
            msr     cpsr_c, r0                 Set SVC mode with FIQ and IRQ enabled  ❼
bucle:      b       bucle                                               ❽
```

Example 11: IRQ and FIQ handlers

```
fiq_handler:
            push    {r0, r1, r2}                                        1
            ldr     r0, =GPBASE
            ldr     r1, =onoff
            ldr     r2, [r1]                                            3
            eors    r2, #1                          Update onoff variable
            str     r2, [r1]                        and test if its 0 or 1
            mov     r1, #0b00000000000000000000001000000000
            streq   r1, [r0, #GPCLR0]                                   3
            strne   r1, [r0, #GPSET0]               Turn on or off red led
            mov     r1, #0b00000000000000000000000000000100
            str     r1, [r0, #GPEDS0]               Clear GPIO2 interrupt   4
            pop     {r0, r1, r2}
            subs    pc, lr, #4                                          5 6
irq_handler:
            push    {r0, r1, r2}                                        1
            ldr     r0, =GPBASE
            ldr     r1, =onoff
            ldr     r2, [r1]                                            3
            eors    r2, #1                          Update onoff variable
            str     r2, [r1]                        and test if its 0 or 1
            mov     r1, #0b00000000000000000000001000000000
            strne   r1, [r0, #GPSET0]                                   3
            streq   r1, [r0, #GPCLR0]               Turn on or off red led
            ldr     r0, =STBASE
            mov     r1, #0b0010                                         4
            str     r1, [r0, #STCS]                 Clear timer interrupt
            ldr     r1, [r0, #STCLO]
            add     r1, #0x400000                   Program timer to
            str     r1, [r0, #STC1]                 interrupt in 4 seconds
            pop     {r0, r1}
            subs    pc, lr, #4                                          5 6
onoff: .word  0
```

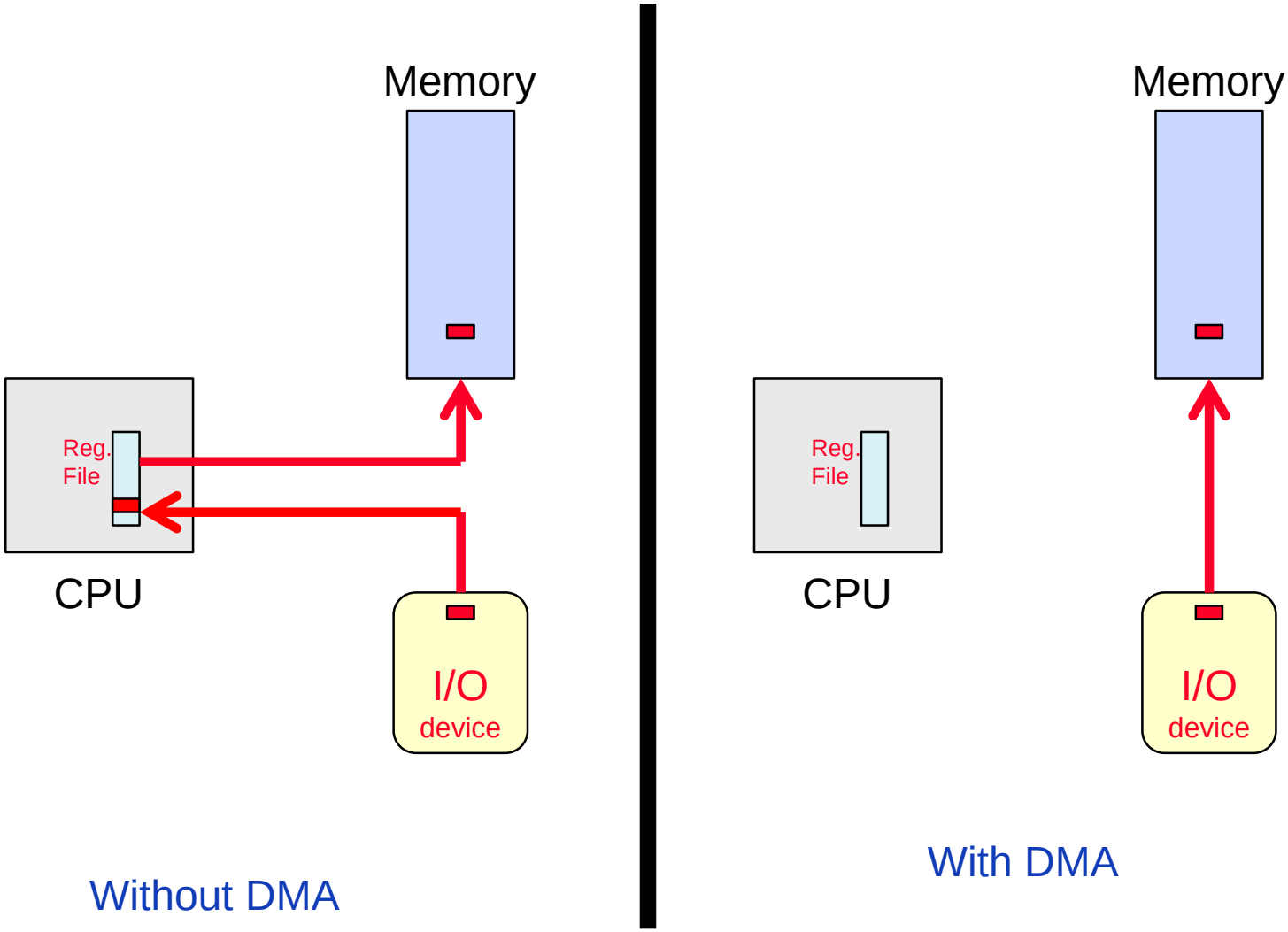
A variable

Direct Memory Access (DMA)

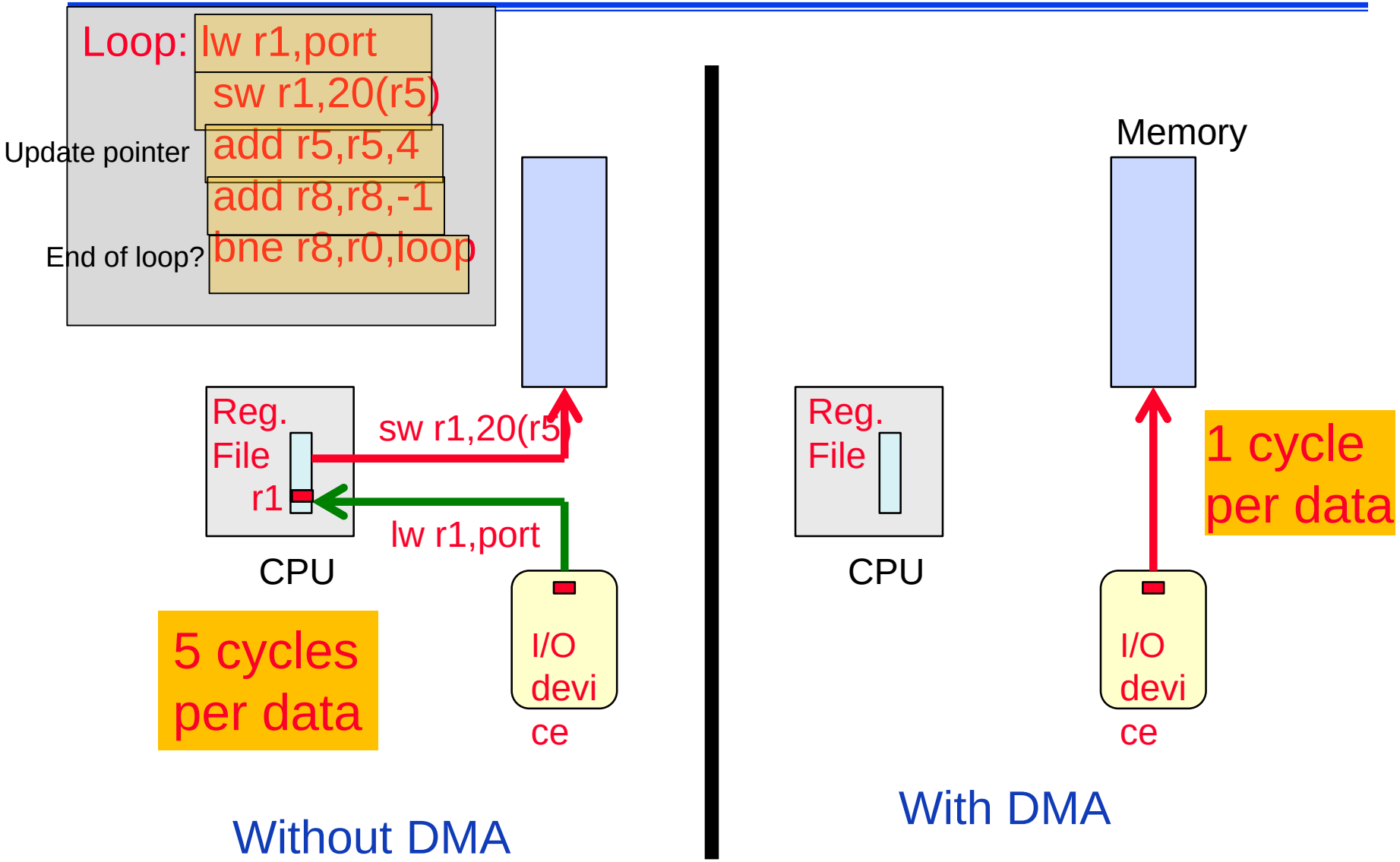
Direct Memory Access (DMA)

- For high-bandwidth devices (like disks) polling or interrupt-driven I/O would consume a *lot* of processor cycles
- With DMA, the DMA controller has the ability to transfer large blocks of data directly to/from the memory without involving the processor
  1. The processor initiates the DMA transfer by supplying the I/O device address (identity), the operation to be performed, the memory address destination/source, the number of bytes to transfer
  2. The DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
  3. When the DMA transfer is complete (or in case of error), the DMA controller interrupts the processor to let it know that the transfer is complete
- There may be multiple DMA devices in one system
  - E.g.: systems with a single memory bus and multiple I/O buses, each I/O bus controller will often contain a DMA
  - Processor and DMA controllers contend for bus cycles and for memory
    - The processor can be delayed when the memory is busy doing a DMA transfer

Direct Memory Access (DMA)

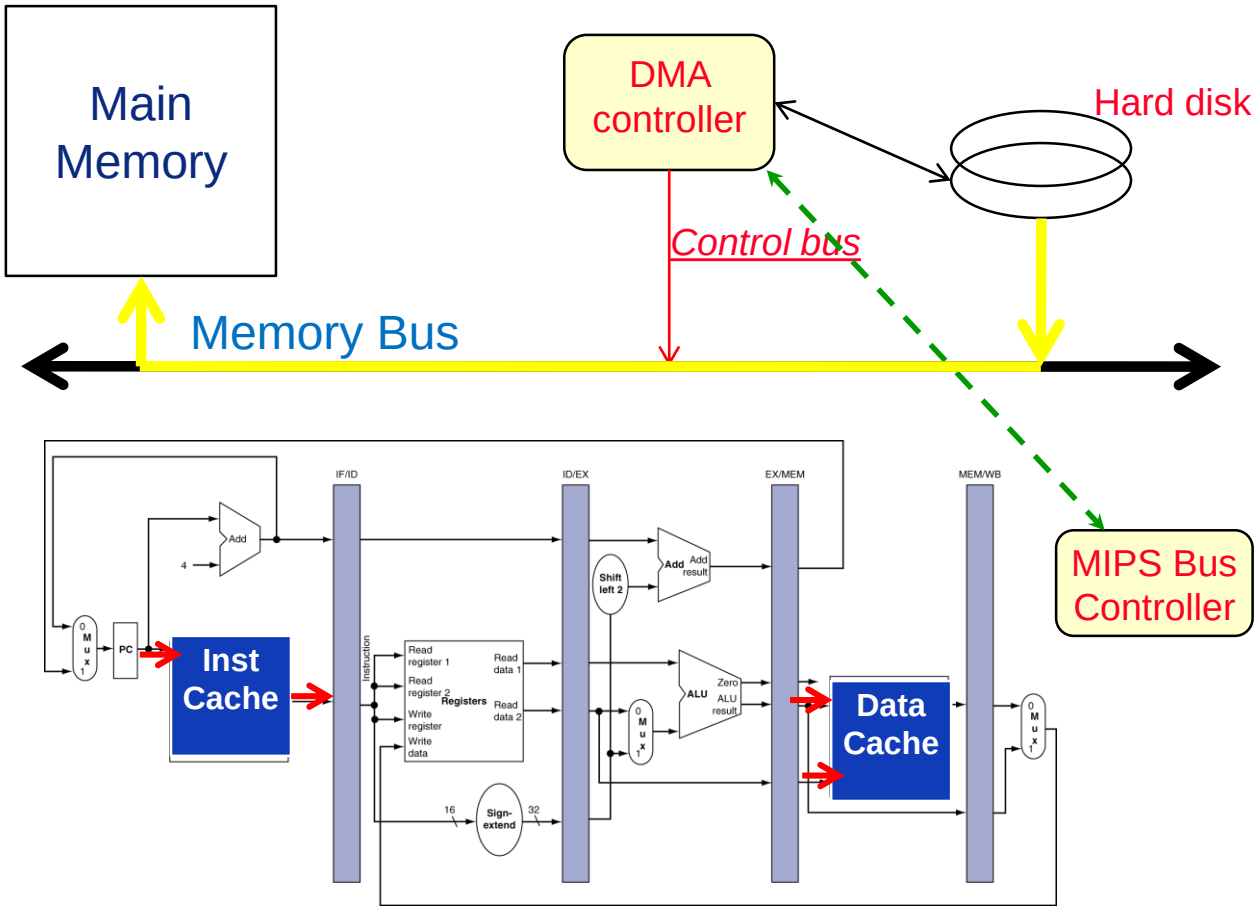


Direct Memory Access (DMA)



Direct Memory Access (DMA)

- Processor works in parallel with the DMA controller
  - Processor dealing with Cache Buses (➔)
  - DMA controler dealing with Main Memory Bus



Direct Memory Access (DMA)

- Example of data cache miss (or updating in a write-through)
  - Processor is dealing with Main Memory Bus