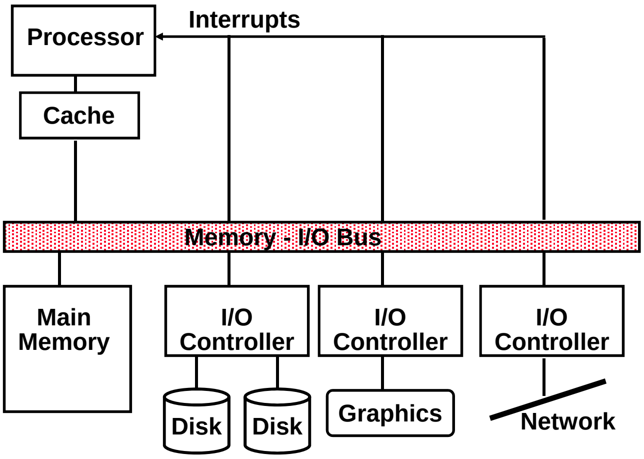
# Chapter 4: Input/Output System

## A Typical I/O System



## Input and Output Devices

- I/O devices are incredibly diverse with respect to
  - Behavior – input, output or storage
  - Partner – human or machine
  - Data rate – the peak rate at which data can be transferred between the I/O device and the main memory or processor

| Device | Behavior | Partner | Data rate (Mb/s) |
|---|---|---|---|
| Keyboard | input | human | 0.0001 |
| Mouse | input | human | 0.0038 |
| Laser printer | output | human | 3.2000 |
| Magnetic disk | storage | machine | 800.0000-3000.0000 |
| Graphics display | output | human | 800.0000-8000.0000 |
| Network/LAN | input or output | machine | 100.0000-10000.0000 |

8 orders of magnitude range

## I/O Performance Measures

- I/O bandwidth (throughput) – amount of information that can be input (output) and communicated across an interconnect (e.g., a bus) to the processor/memory (I/O device) per unit time
  1. How much data can we move through the system in a certain time?
  2. How many I/O operations can we do per unit time?
- I/O response time (latency) – the total elapsed time to accomplish an input or output operation
  - An especially important performance metric in real-time systems
- Expandability – is there any easy way to connect another disk to the system?
- Resilience – if this I/O controller (network) fails, is it going to affect the rest of the network?

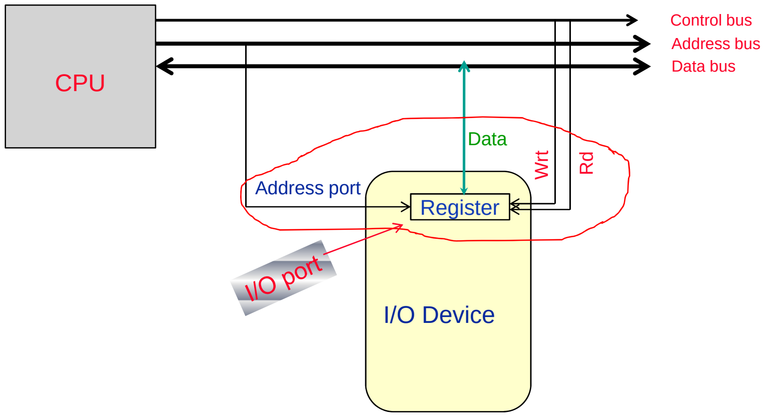
# Input/output Ports

## 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

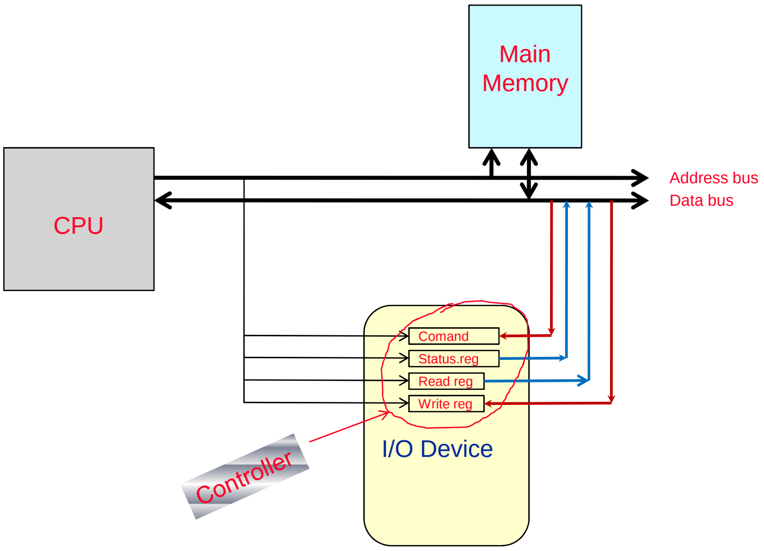
# I/O ports



# 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



# 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

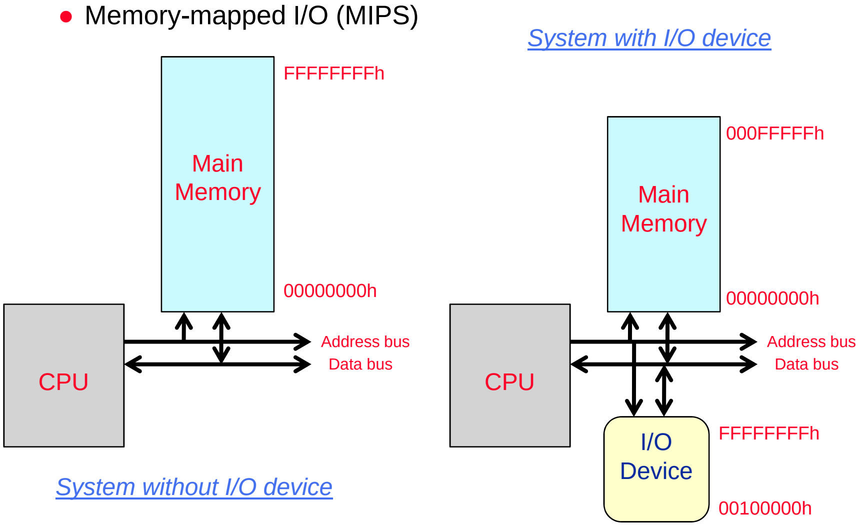
❑ How the processor directs the I/O devices

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)

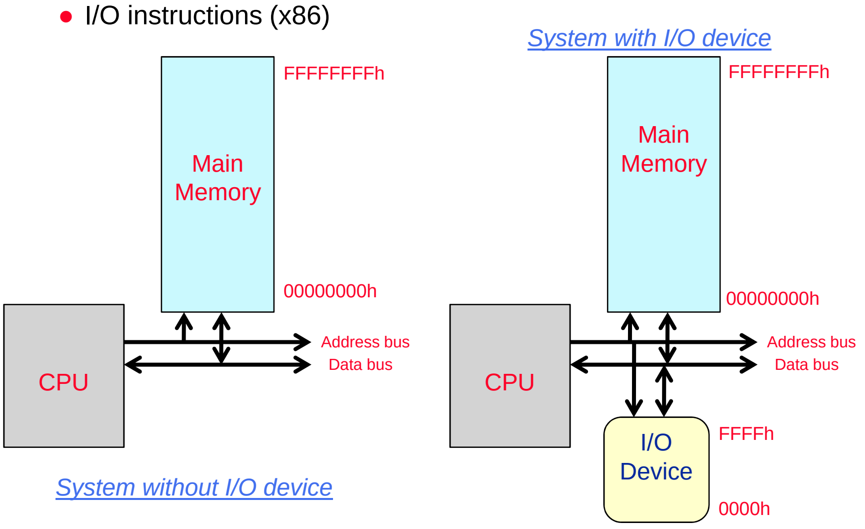


# Communication of I/O Devices and Processor

- 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: $\overline{\text{IO}}$/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)

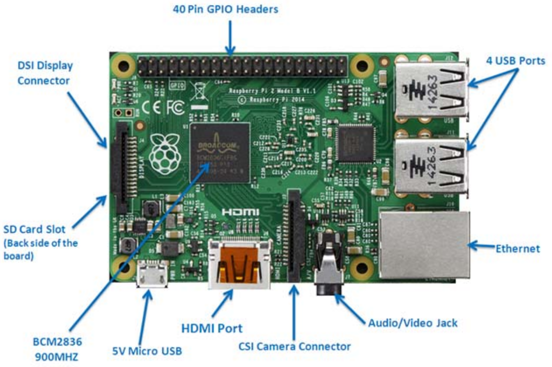


System with I/O device

System without I/O device

# Raspberry Pi

- Raspberrypi 2 B: a 900MHz quad-core ARM Cortex-A7 CPU 1GB RAM
- Like the (Pi 1) Model B+, it also has: 100 Base Ethernet, 4 USB ports, 40 GPIO pins, full HDMI port, combined 3.5mm audio jack and composite video, camera interface (CSI), display interface (DSI), micro SD card slot, videoCore IV 3D graphics core.



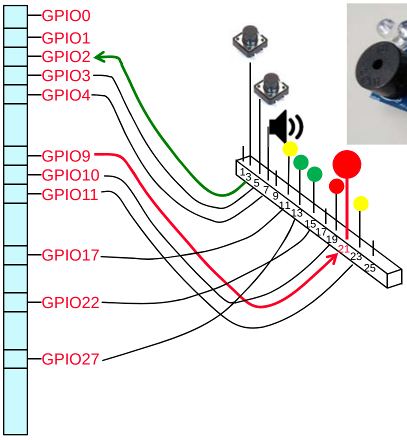
# Raspberry Pi: GPIO port

| | | Pin No. | | | |
|---|---|---|---|---|---|
| | 3.3V | 1 | 2 | 5V | |
| | GPIO2 | 3 | 4 | 5V | |
| | GPIO3 | 5 | 6 | GND | |
| 🔊 | GPIO4 | 7 | 8 | GPIO14 | Tx |
| | GND | 9 | 10 | GPIO15 | Rx |
| ● | 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 | |

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

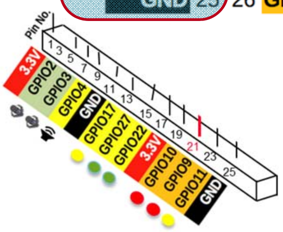
# GPIO pins

- GPIO: two rows with 13 pines
- We will use the inner row for connecting an external board

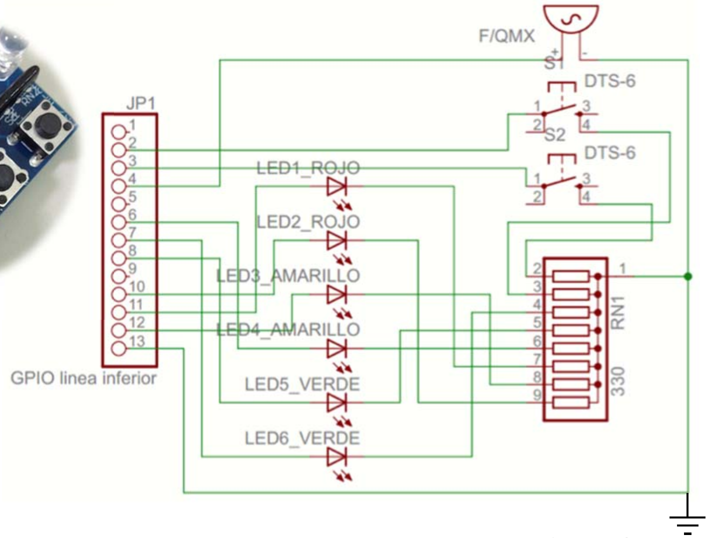


Pin No.

| | | | | |
|---|---|---|---|---|
| | 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 |



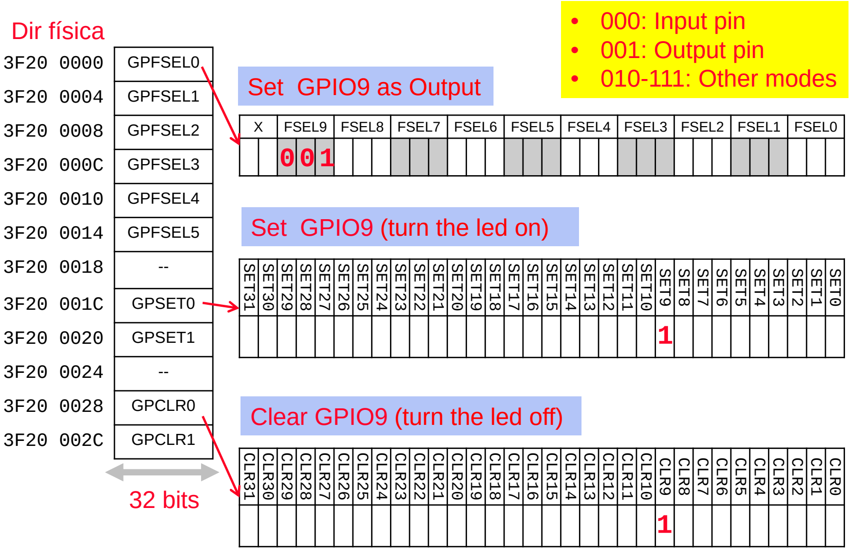
# External board connected to GPIO



# 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



| Dir física | |
|---|---|
| 3F20 0000 | GPFSEL0 |
| 3F20 0004 | GPFSEL1 |
| 3F20 0008 | GPFSEL2 |
| 3F20 000C | GPFSEL3 |
| 3F20 0010 | GPFSEL4 |
| 3F20 0014 | GPFSEL5 |
| 3F20 0018 | -- |
| 3F20 001C | GPSET0 |
| 3F20 0020 | GPSET1 |
| 3F20 0024 | -- |
| 3F20 0028 | GPCLR0 |
| 3F20 002C | GPCLR1 |

32 bits

- 000: Input pin
- 001: Output pin
- 010-111: Other modes

Set GPIO9 as Output

| X | FSEL9 | FSEL8 | FSEL7 | FSEL6 | FSEL5 | FSEL4 | FSEL3 | FSEL2 | FSEL1 | FSEL0 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 001 | | | | | | | | | |

Set GPIO9 (turn the led on)

| SET31 | SET30 | SET29 | SET28 | SET27 | SET26 | SET25 | SET24 | SET23 | SET22 | SET21 | SET20 | SET19 | SET18 | SET17 | SET16 | SET15 | SET14 | SET13 | SET12 | SET11 | SET10 | SET9 | SET8 | SET7 | SET6 | SET5 | SET4 | SET3 | SET2 | SET1 | SET0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | 1 | | | | | | | | | |

Clear GPIO9 (turn the led off)

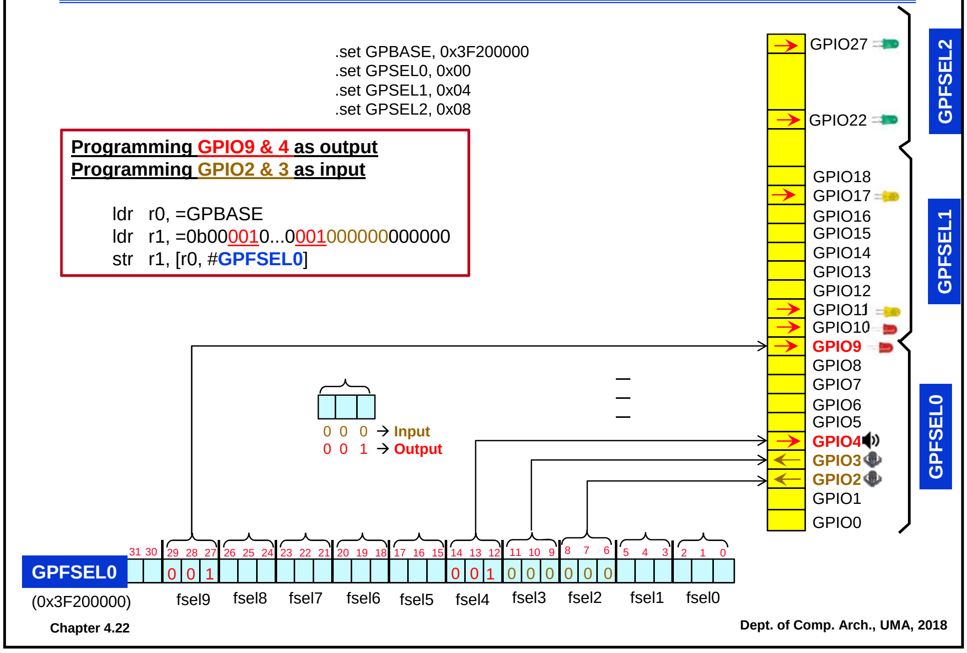
| CLR31 | CLR30 | CLR29 | CLR28 | CLR27 | CLR26 | CLR25 | CLR24 | CLR23 | CLR22 | CLR21 | CLR20 | CLR19 | CLR18 | CLR17 | CLR16 | CLR15 | CLR14 | CLR13 | CLR12 | CLR11 | CLR10 | CLR9 | CLR8 | CLR7 | CLR6 | CLR5 | CLR4 | CLR3 | CLR2 | CLR1 | CLR0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | | | | 1 | | | | | | | | | |

# Configure GPIO pins as input/output

```
.set GPBASE, 0x3F200000
.set GPSEL0, 0x00
.set GPSEL1, 0x04
.set GPSEL2, 0x08
```

**Programming GPIO9 & 4 as output**
**Programming GPIO2 & 3 as input**

```
ldr   r0, =GPBASE
ldr   r1, =0b000010...0001000000000000
str   r1, [r0, #GPFSEL0]
```

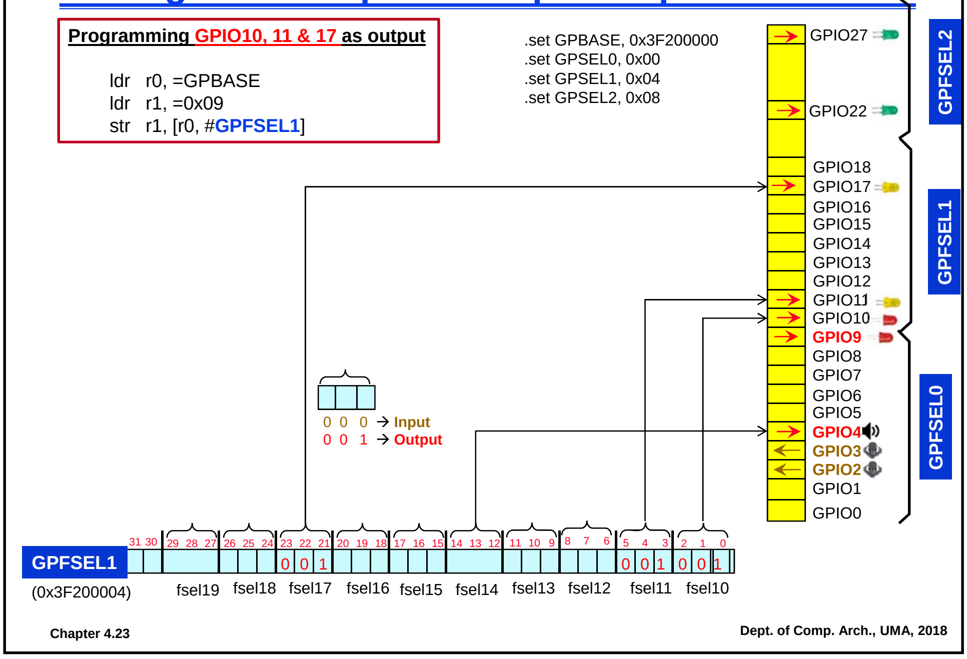


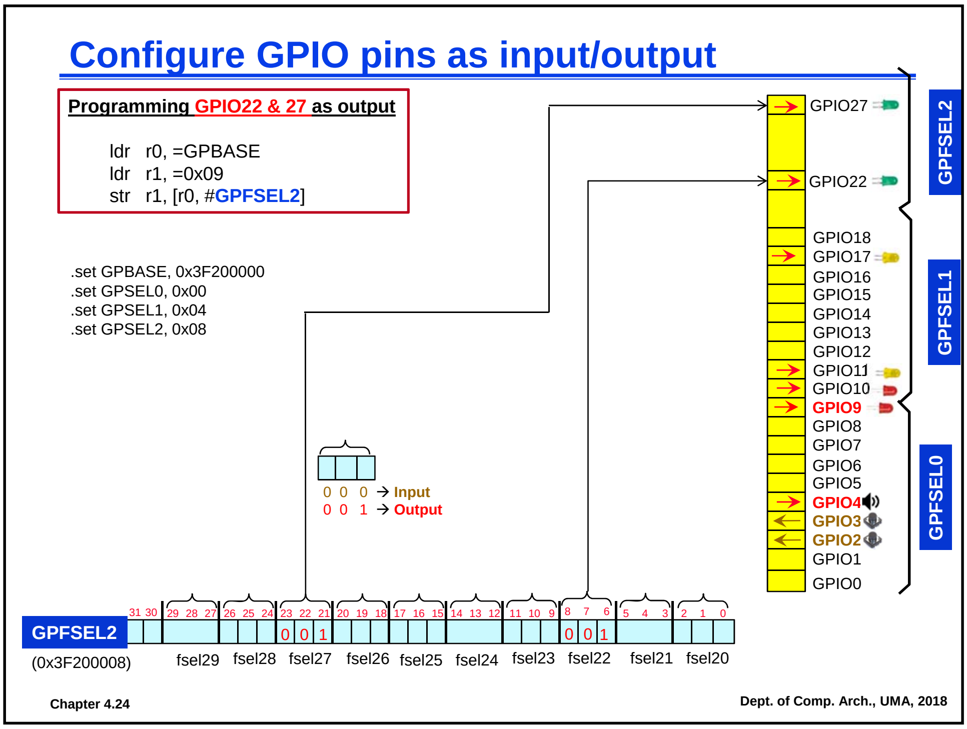
# Configure GPIO pins as input/output

**Programming GPIO10, 11 & 17 as output**

```
ldr   r0, =GPBASE
ldr   r1, =0x09
str   r1, [r0, #GPFSEL1]
```

```
.set GPBASE, 0x3F200000
.set GPSEL0, 0x00
.set GPSEL1, 0x04
.set GPSEL2, 0x08
```



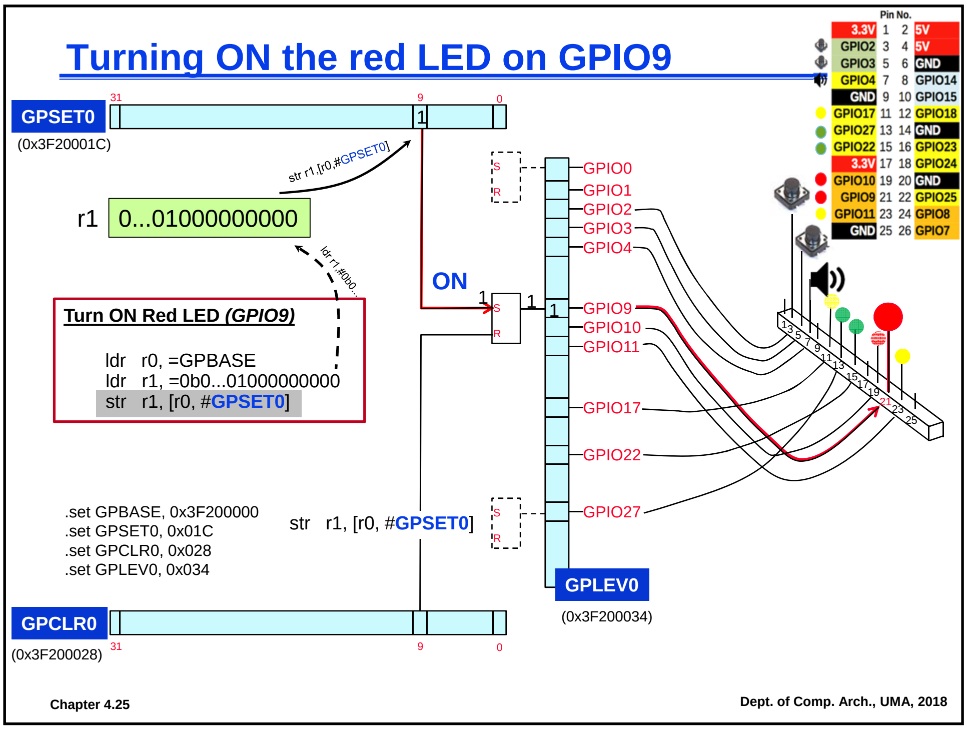


# Configure GPIO pins as input/output

**Programming GPIO22 & 27 as output**

```
ldr   r0, =GPBASE
ldr   r1, =0x09
str   r1, [r0, #GPFSEL2]
```

```
.set GPBASE, 0x3F200000
.set GPSEL0, 0x00
.set GPSEL1, 0x04
.set GPSEL2, 0x08
```



# Turning ON the red LED on GPIO9

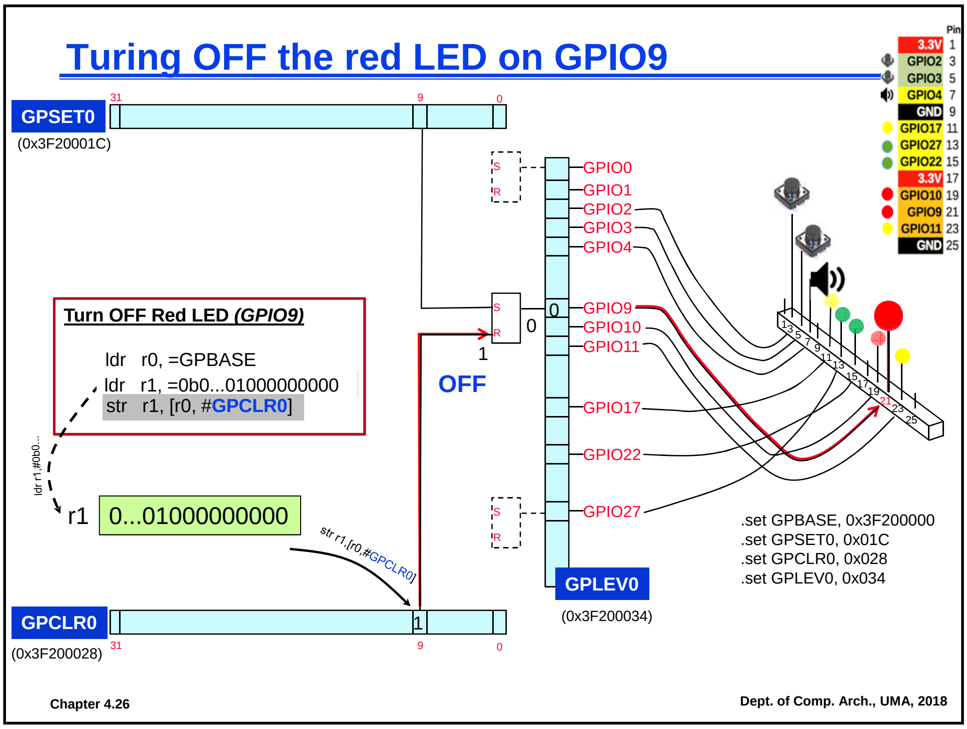
**Turn ON Red LED *(GPIO9)***

```
ldr   r0, =GPBASE
ldr   r1, =0b0...01000000000
str   r1, [r0, #GPSET0]
```

```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

| 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 |

# Turing OFF the red LED on GPIO9



**Turn OFF Red LED *(GPIO9)***

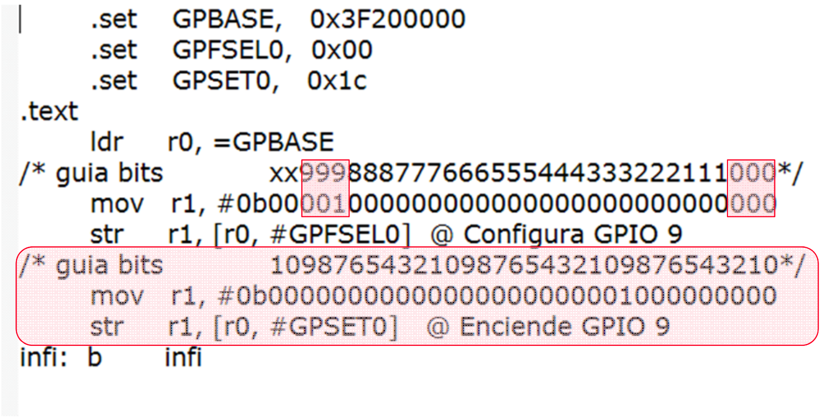
```
ldr   r0, =GPBASE
ldr   r1, =0b0...01000000000
str   r1, [r0, #GPCLR0]
```

r1 0...01000000000

```
.set GPBASE, 0x3F200000
.set GPSETO, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

# 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          xx999888777666555444333222111000*/
        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
```

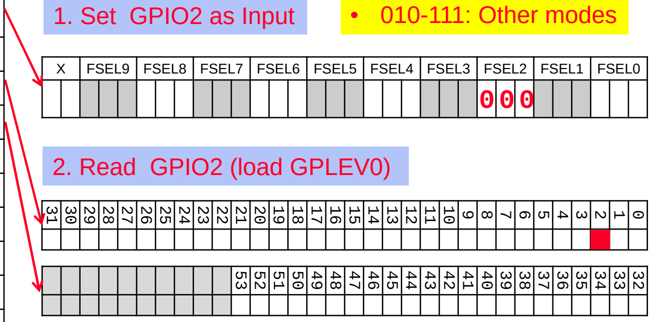
## GPIO memory mapping cont

- 000: Input pin
- 001: Output pin
- 010-111: Other modes

| | |
|---|---|
| 3F20 0000 | GPFSEL 0 |
| ---- | ---- |
| 3F20 0034 | GPLEV0 |
| 3F20 0038 | GPLEV1 |
| 3F20 003C | -- |
| 3F20 0040 | GPEDS0 |
| 3F20 0044 | GPEDS1 |
| 3F20 0048 | -- |
| 3F20 004C | GPREN0 |
| 3F20 0050 | GPREN1 |
| 3F20 0054 | -- |
| 3F20 0058 | GPFEN0 |
| 3F20 005C | GPFEN1 |
| ---- | ---- |

1. Set GPIO2 as Input

2. Read GPIO2 (load GPLEV0)



GPLEVn: GPIO Pin Level Registers:

- Returns 0 if level is 0V or 1 when level is 3.3V

Many more ports, some of them will be explained later!

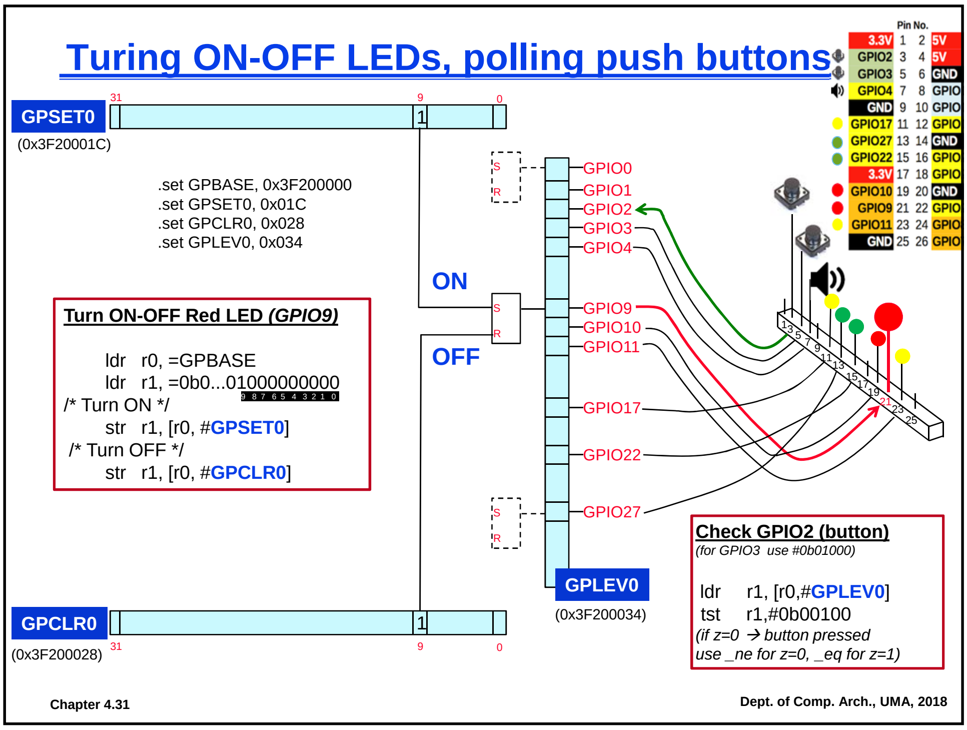
## Communication of I/O Devices and Processor

- 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

## Turing ON-OFF LEDs, polling push buttons



| | Pin No. | | |
|---|---|---|---|
| 3.3V | 1 | 2 | 5V |
| GPIO2 | 3 | 4 | 5V |
| GPIO3 | 5 | 6 | GND |
| GPIO4 | 7 | 8 | GPIO |
| GND | 9 | 10 | GPIO |
| GPIO17 | 11 | 12 | GPIO |
| GPIO27 | 13 | 14 | GND |
| GPIO22 | 15 | 16 | GPIO |
| 3.3V | 17 | 18 | GPIO |
| GPIO10 | 19 | 20 | GND |
| GPIO9 | 21 | 22 | GPIO |
| GPIO11 | 23 | 24 | GPIO |
| GND | 25 | 26 | GPIO |

```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

**Turn ON-OFF Red LED *(GPIO9)***

```
      ldr   r0, =GPBASE
      ldr   r1, =0b0...01000000000
                      9 87 65 4 3 2 1 0
/* Turn ON */
      str   r1, [r0, #GPSET0]
 /* Turn OFF */
      str   r1, [r0, #GPCLR0]
```

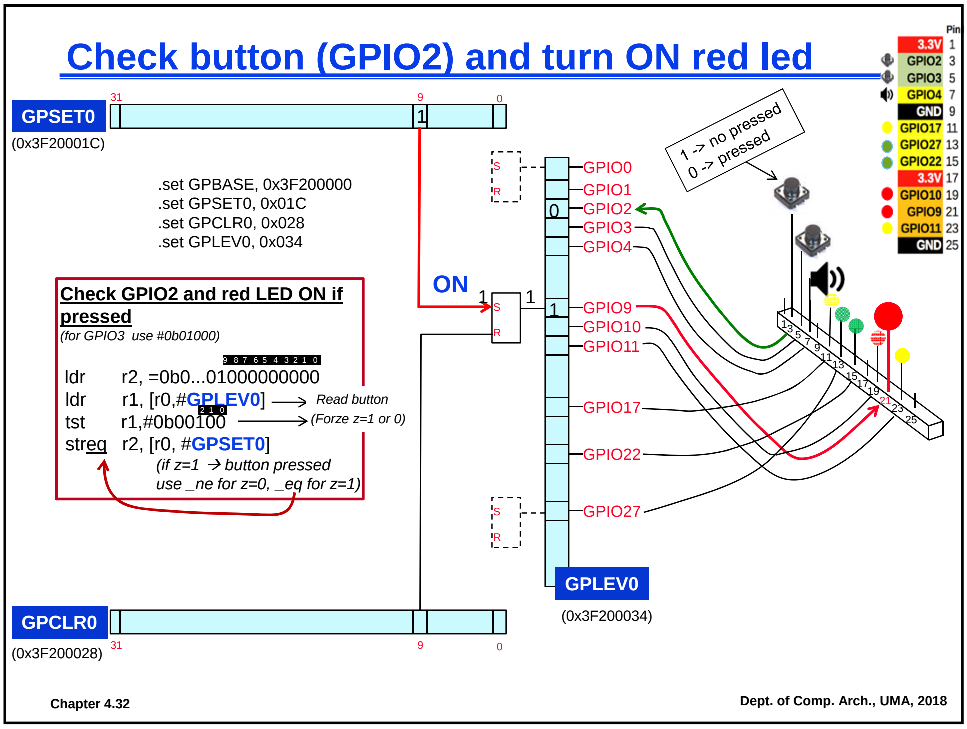
**Check GPIO2 (button)**

*(for GPIO3  use #0b01000)*

```
ldr     r1, [r0,#GPLEV0]
tst     r1,#0b00100
```

*(if z=0 → button pressed*
*use _ne for z=0, _eq for z=1)*

# Check button (GPIO2) and turn ON red led



```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

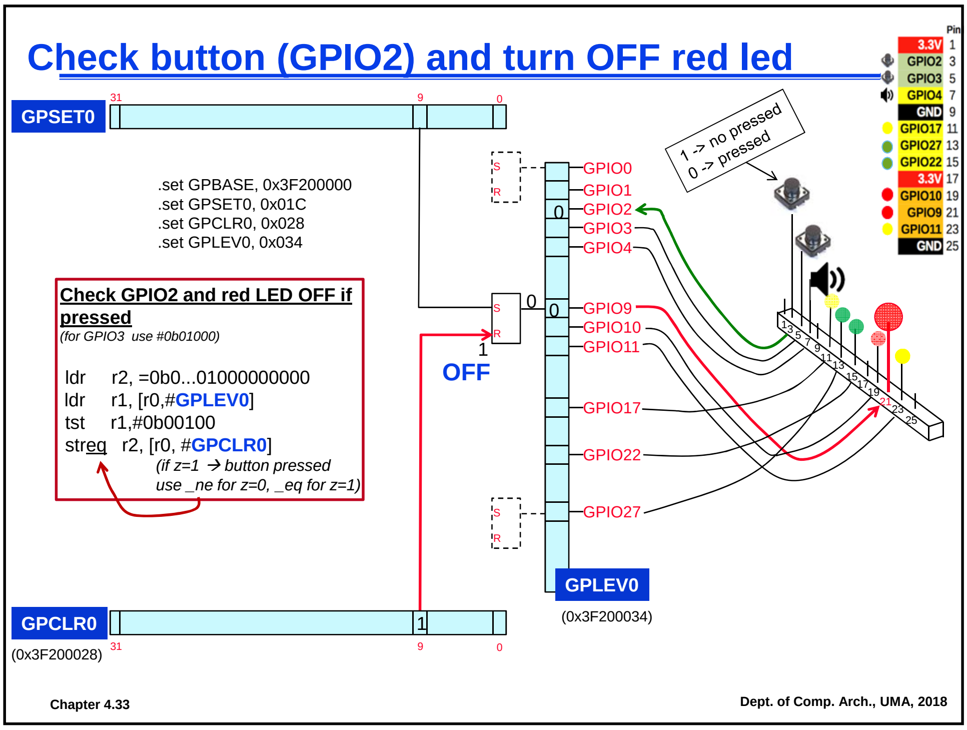
**Check GPIO2 and red LED ON if pressed**

*(for GPIO3  use #0b01000)*

```
                  9 8 7 6 5 4 3 2 1 0
ldr    r2, =0b0...01000000000
ldr    r1, [r0,#GPLEV0]  ---> Read button
                  2 1 0
tst    r1,#0b00100       ---> (Forze z=1 or 0)
streq  r2, [r0, #GPSET0]
```

*(if z=1 → button pressed*
*use _ne for z=0, _eq for z=1)*

# Check button (GPIO2) and turn OFF red led



```
.set GPBASE, 0x3F200000
.set GPSET0, 0x01C
.set GPCLR0, 0x028
.set GPLEV0, 0x034
```

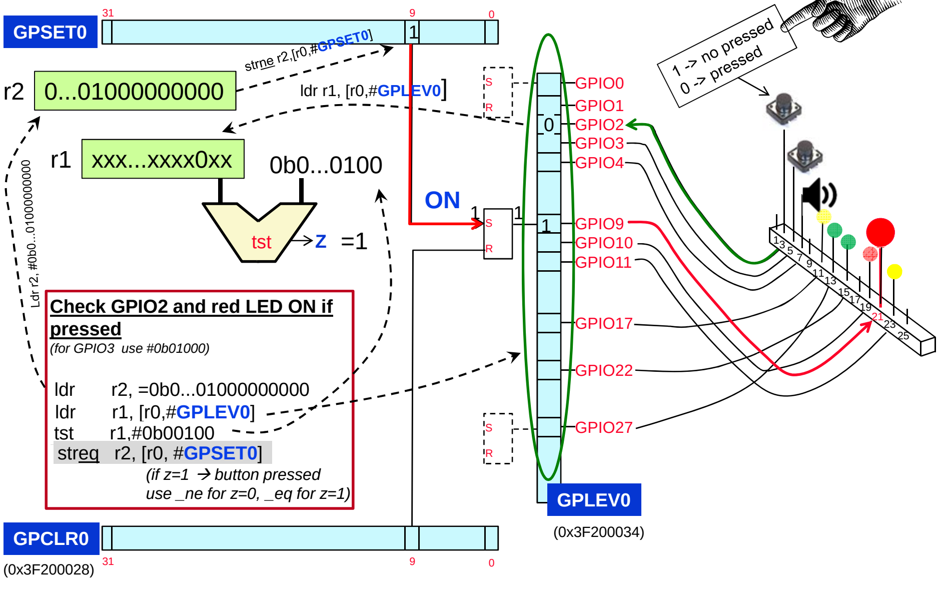
**Check GPIO2 and red LED OFF if pressed**

*(for GPIO3  use #0b01000)*

```
ldr    r2, =0b0...01000000000
ldr    r1, [r0,#GPLEV0]
tst    r1,#0b00100
streq  r2, [r0, #GPCLR0]
```

*(if z=1 → button pressed*
*use _ne for z=0, _eq for z=1)*

# Check button (GPIO2) and turn on red led



**Check GPIO2 and red LED ON if pressed**

*(for GPIO3 use #0b01000)*

```
ldr     r2, =0b0...01000000000
ldr     r1, [r0,#GPLEV0]
tst     r1,#0b00100
streq   r2, [r0, #GPSET0]
```

*(if z=1 → button pressed*
*use _ne for z=0, _eq for z=1)*

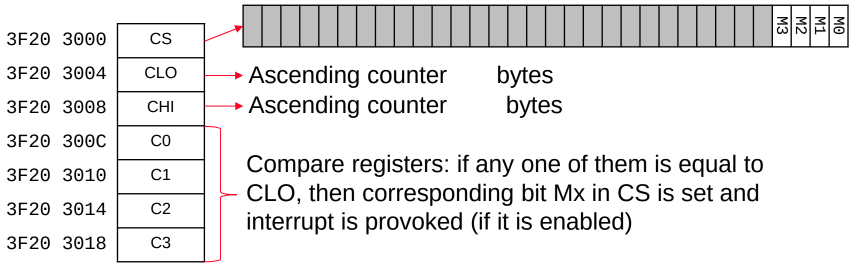
# Example 2

- Code for checking push button (GPIO2) and turning led on:

```
        .set GPBASE,   0x3F200000
        .set GPFSEL0,  0x00
        .set GPSET0,   0x1c
        .set GPLEV0,   0x34
.text
        ldr r0, =GPBASE
/* guia bits        xx999888777666555444333222111000 */
        mov r1,  #0b00001000000000000000000000000000
        str r1, [r0, #GPFSEL0]
/* mask for testing GPIO2 */
        mov r2, #0b00000000000000000000000000000100
bucle:
        ldr r3, [r0, #GPLEV0]
        tst r3, r2
        bne bucle
/* guia bits       10987654321098765432109876543210
        mov r1, #0b00000000000000000000001000000000
        str r1, [r0, #GPSET0]
infi:  b infi
```

# Raspberry Pi: System Timer

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

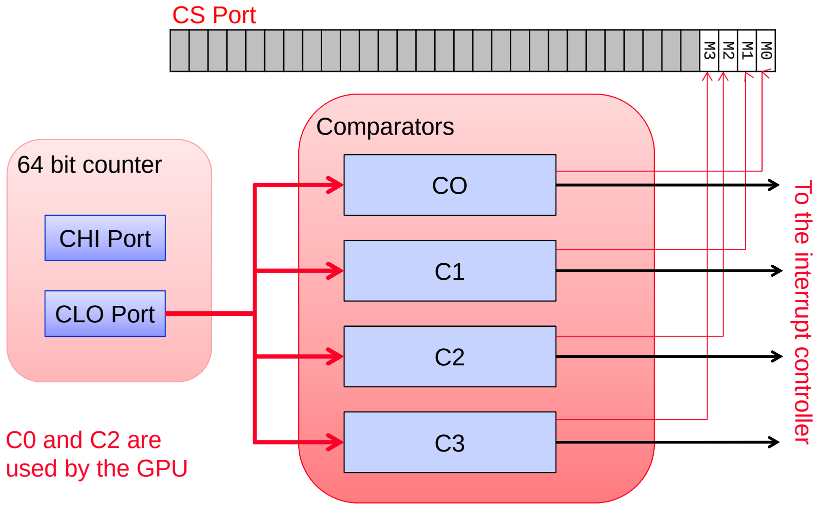


| | |
|---|---|
| 3F20 3000 | CS |
| 3F20 3004 | CLO |
| 3F20 3008 | CHI |
| 3F20 300C | C0 |
| 3F20 3010 | C1 |
| 3F20 3014 | C2 |
| 3F20 3018 | C3 |

Ascending counter      bytes
Ascending counter       bytes

Compare registers: if any one of them is equal to CLO, then corresponding bit Mx in CS is set and interrupt is provoked (if it is enabled)

# System Timer

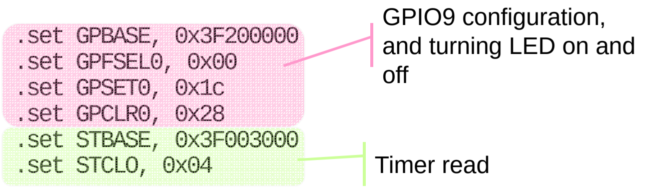


C0 and C2 are used by the GPU

## Example 3: red LED blinking

- We must:
  1. Configure GPIO9
  2. Turn the led on
  3. Wait some time
  4. Turn the led off
  5. Wait some time
  6. Repeat steps 2-5 forever
- We need:
  - A routine that “waits”

## Example 3: red LED blinking



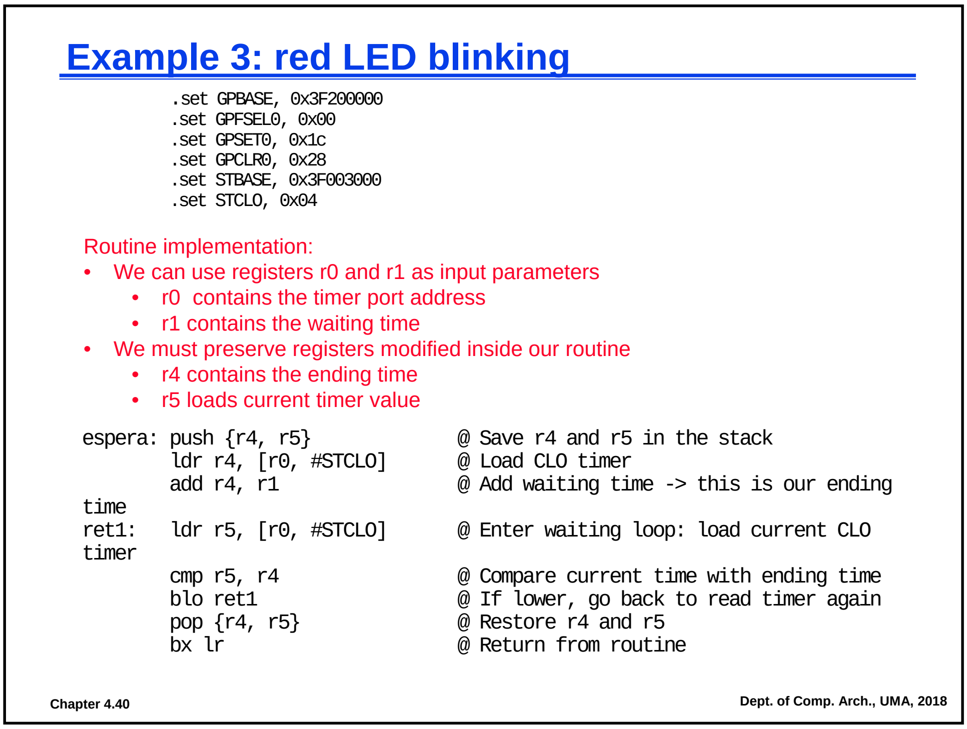
```
.set GPBASE, 0x3F200000
.set GPFSEL0, 0x00
.set GPSET0, 0x1c
.set GPCLR0, 0x28
.set STBASE, 0x3F003000
.set STCLO, 0x04
```

GPIO9 configuration, and turning LED on and off

Timer read

Our “waiting” routine will:
1. Read waiting time (input parameter)
2. Repeat

```
        Read current timer value
        while it is lower than waiting time
```



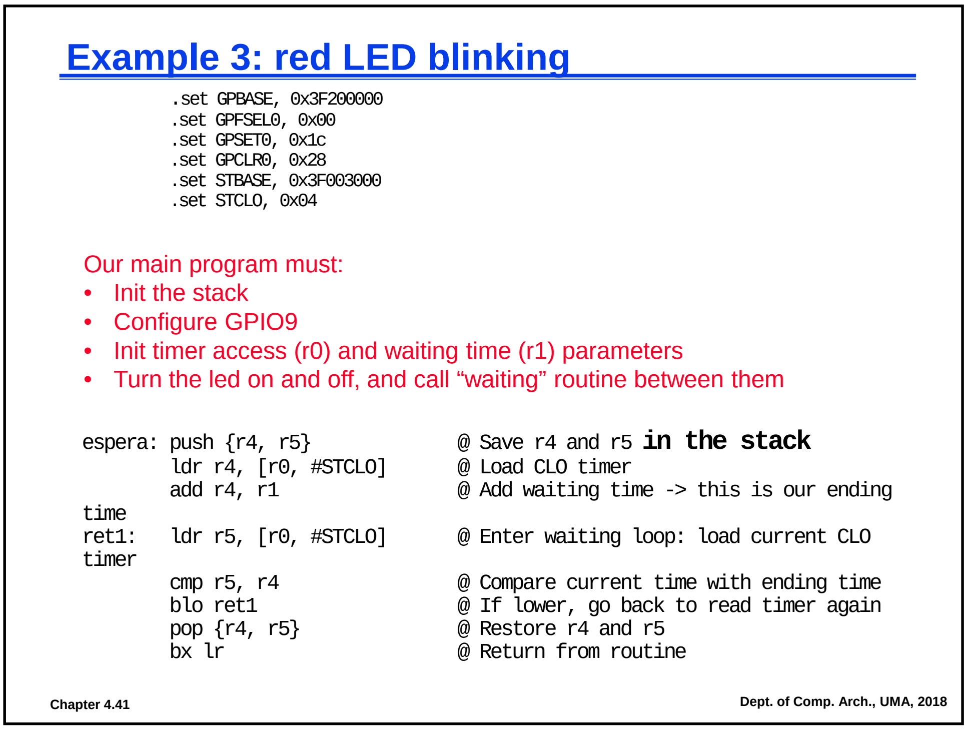
# Example 3: red LED blinking

```
        .set GPBASE, 0x3F200000
        .set GPFSEL0, 0x00
        .set GPSET0, 0x1c
        .set GPCLR0, 0x28
        .set STBASE, 0x3F003000
        .set STCLO, 0x04
```

Routine implementation:

- We can use registers r0 and r1 as input parameters
  - r0 contains the timer port address
  - r1 contains the waiting time
- We must preserve registers modified inside our routine
  - r4 contains the ending time
  - r5 loads current timer value

```
espera: push {r4, r5}           @ Save r4 and r5 in the stack
        ldr r4, [r0, #STCLO]    @ Load CLO timer
        add r4, r1              @ Add waiting time -> this is our ending
time
ret1:   ldr r5, [r0, #STCLO]    @ Enter waiting loop: load current CLO
timer
        cmp r5, r4              @ Compare current time with ending time
        blo ret1                @ If lower, go back to read timer again
        pop {r4, r5}            @ Restore r4 and r5
        bx lr                   @ Return from routine
```



# Example 3: red LED blinking

```
        .set GPBASE, 0x3F200000
        .set GPFSEL0, 0x00
        .set GPSET0, 0x1c
        .set GPCLR0, 0x28
        .set STBASE, 0x3F003000
        .set STCLO, 0x04
```

Our main program must:

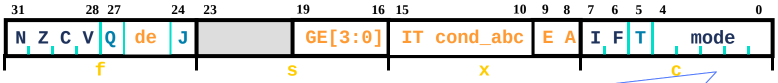
- Init the stack
- Configure GPIO9
- Init timer access (r0) and waiting time (r1) parameters
- Turn the led on and off, and call “waiting” routine between them

```
espera: push {r4, r5}           @ Save r4 and r5 in the stack
        ldr r4, [r0, #STCLO]    @ Load CLO timer
        add r4, r1              @ Add waiting time -> this is our ending
time
ret1:   ldr r5, [r0, #STCLO]    @ Enter waiting loop: load current CLO
timer
        cmp r5, r4              @ Compare current time with ending time
        blo ret1                @ If lower, go back to read timer again
        pop {r4, r5}            @ Restore r4 and r5
        bx lr                   @ Return from routine
```

# Example 3: red LED blinking

```
        .set GPBASE, 0x3F200000
        .set GPFSEL0, 0x00
        .set GPSET0, 0x1c
        .set GPCLR0, 0x28
        .set STBASE, 0x3F003000
        .set STCLO, 0x04
```

Program Status Register



It is not strictly necessary but it is better to make sure SVC (10011) mode is enabled

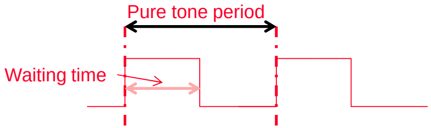
```
espera: push {r4, r5}           @ Save r4 and r5 in the stack
        ldr r4, [r0, #STCLO]    @ Load CLO timer
        add r4, r1              @ Add waiting time -> this is our ending
time
ret1:   ldr r5, [r0, #STCLO]    @ Enter waiting loop: load current CLO
timer
        cmp r5, r4              @ Compare current time with ending time
        blo ret1                @ If lower, go back to read timer again
        pop {r4, r5}            @ Restore r4 and r5
        bx lr                   @ Return from routine
```

# Example 3: red LED blinking

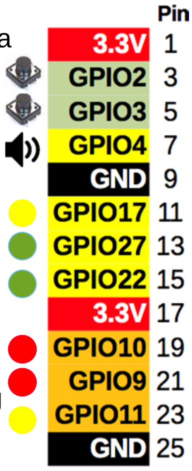
```
        .set GPBASE, 0x3F200000
        .set GPFSEL0, 0x00
        .set GPSET0, 0x1c
        .set GPCLR0, 0x28
        .set STBASE, 0x3F003000
        .set STCLO, 0x04
.text   mov r0, #0b11010011
        msr cpsr_c, r0         @ SVC mode enabled
        mov sp, #0x08000000    @ Init stack in SVC mode
        ldr r4, =GPBASE
        mov r5, #0b00001000000000000000000000000000
        str r5, [r4, #GPFSEL0]          @ Configure GPIO9
        mov r5, #0b00000000000000000000001000000000
        ldr r0, =STBASE        @ r0 is an input parameter (ST base address)
        ldr r1, =500000        @ r1 is an input parameter (waiting time in microseconds)
bucle:  bl espera              @ Call waiting routine
        str r5, [r4, #GPSET0]           @ Turn LED on
        bl espera              @ Call waiting routine
        str r5, [r4, #GPCLR0]           @ Turn LED off
        b bucle
espera: push {r4, r5}          @ Save r4 and r5 in the stack
        ldr r4, [r0, #STCLO]   @ Load CLO timer
        add r4, r1             @ Add waiting time -> this is our ending time
ret1:   ldr r5, [r0, #STCLO]   @ Enter waiting loop: load current CLO timer
        cmp r5, r4             @ Compare current time with ending time
        blo ret1               @ If lower, go back to read timer again
        pop {r4, r5}           @ Restore r4 and r5
        bx lr                  @ Return from routine
```

# Example 4: sound generation

- ❑ A square wave can simulate a sound
  - A pure tone is a sinusoidal waveform with a single frequency
- ❑ Very similar to led blinking
  - We use GPIO4 instead of GPIO9
  - The waiting time is equal to half the period



  - E.g: a 440 Hz tone (La) has a period of $\frac{1}{440s^{-1}} = 2.272\ ms$, thus the needed waiting time is $\frac{2.272}{2} = 1136\mu s$



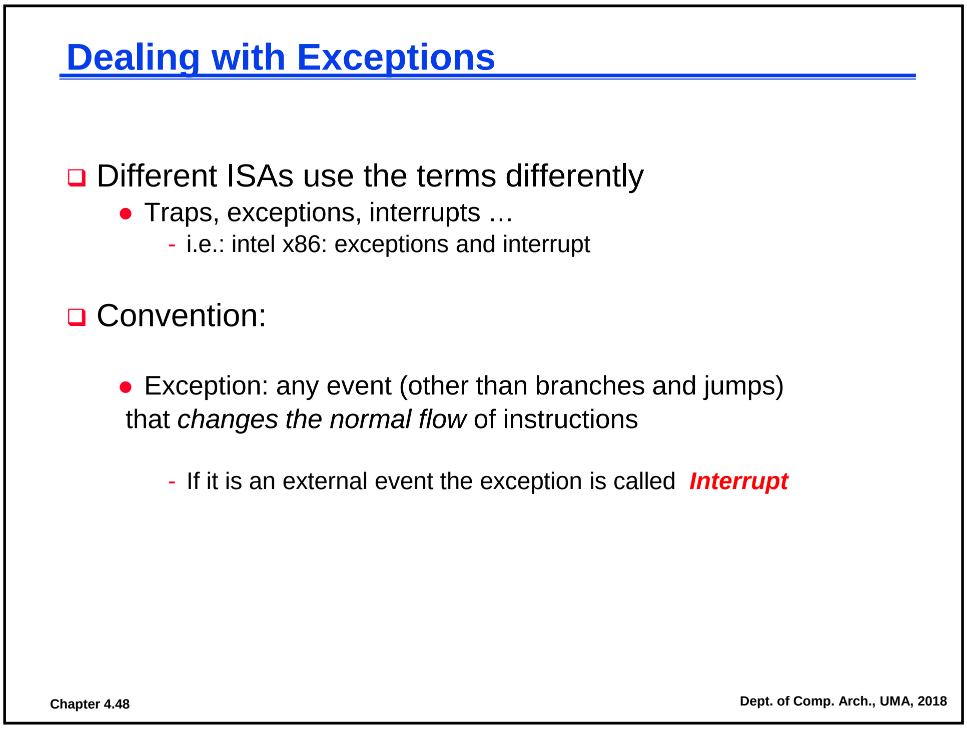
# Example 4: sound generation

```
        .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, #0x08000000     @  Init stack in SVC mode
        ldr r4, =GPBASE
        mov r5, #0b00000000000000000001000000000000
        str r5, [r4, #GPFSEL0]  @ Configure GPIO4
        mov r5, #0b00000000000000000000000000010000
        ldr r0, =STBASE         @ r0 is an input parameter (ST base address)
        ldr r1, =1136           @ r1 is an input parameter (waiting time in microseconds)
bucle:  bl espera               @ Call waiting routine
        str r5, [r4, #GPSET0]   @ Turn LED on
        bl espera               @ Call waiting routine
        str r5, [r4, #GPCLR0]   @ Turn LED off
        b bucle
espera: push {r4, r5}           @ Save r4 and r5 in the stack
        ldr r4, [r0, #STCLO]    @ Load CLO timer
        add r4, r1              @ Add waiting time -> this is our ending time
ret1:   ldr r5, [r0, #STCLO]    @ Enter waiting loop: load current CLO timer
        cmp r5, r4              @ Compare current time with ending time
        blo ret1                @ If lower, go back to read timer again
        pop {r4, r5}            @ Restore r4 and r5
        bx lr                   @ Return from routine
```

# Exceptions

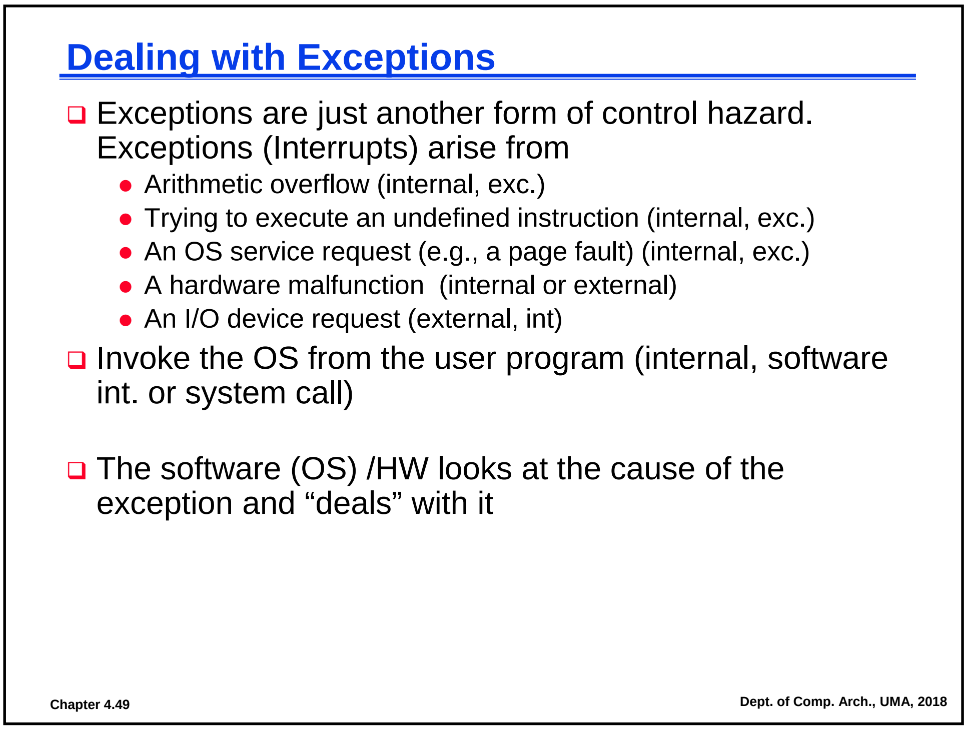
## Exceptions

- ❑ "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)
- ❑ Dealing with them without sacrificing performance is hard



## Dealing with Exceptions

- Different ISAs use the terms differently
  - Traps, exceptions, interrupts …
    - i.e.: intel x86: exceptions and interrupt
- Convention:
  - Exception: any event (other than branches and jumps) that *changes the normal flow* of instructions
    - If it is an external event the exception is called ***Interrupt***



## Dealing with Exceptions

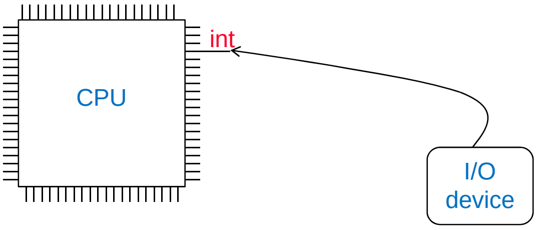
- Exceptions are just another form of control hazard. Exceptions (Interrupts) arise from
  - Arithmetic overflow (internal, exc.)
  - Trying to execute an undefined instruction (internal, exc.)
  - An OS service request (e.g., a page fault) (internal, exc.)
  - A hardware malfunction  (internal or external)
  - 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

## Two Types of Exceptions

- Internal exception – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler:
    - stop the offending instruction *midstream* in the pipeline
    - pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

## Two Types of Exceptions

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



# Interrupt Driven I/O

- An I/O interrupt is asynchronous wrt instruction execution
  - Is not associated with any instruction so doesn't prevent any instruction from completing
    - You can pick your own convenient point to handle the interrupt
    - Control unit needs only check for a pending I/O interrupt at the time it starts a new instruction
- 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.
    - **Auto-vectored interrupts:** each exception has vector associated to it.
    - When the handle 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

- 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
    - NI*4 is the offset to the exception vectors 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

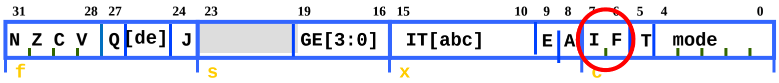
- Type of exceptions:
  - Reset: pins in P6 fire a bootload
  - Undefined instruction: op. code not valid
  - Software interruptions: system calls
  - Prefecth abort /data abort: memory misalignment, access privilege errors
  - IRQ: interruptions due to external devices
  - FIQ: fast interruptions

# Exception priorities

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

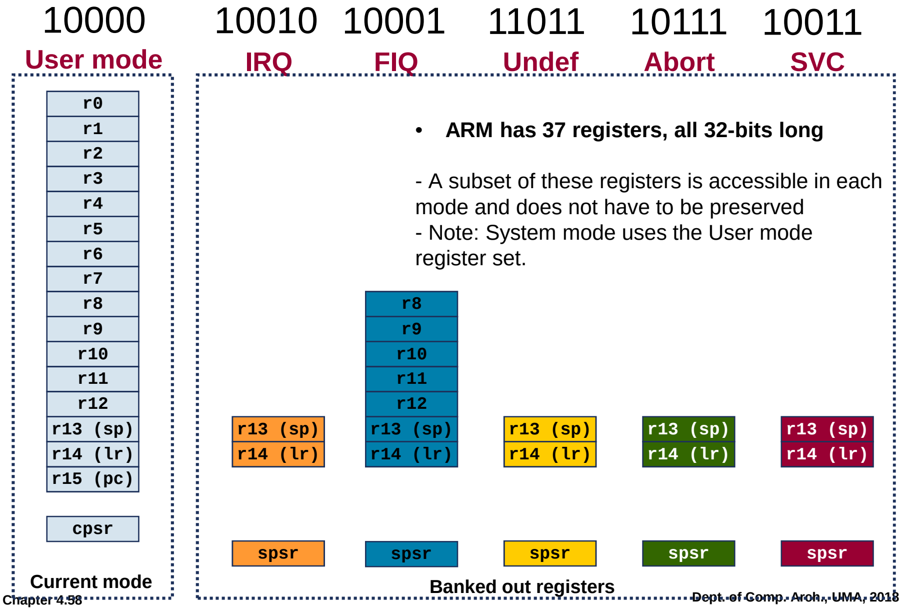
| Priority | | Exception |
|---|---|---|
| Higuest | 1 | Reset |
| | 2 | Precise Data Abort |
| | 3 | FIQ |
| | 4 | IRQ |
| | 5 | Prefetch Abort |
| | 6 | Imprecise Data Abort |
| Lowest | 7 | BKPT<br>Undefined Instruction<br>SVC<br>SMC |

## 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

## Banking of registers



- **ARM has 37 registers, all 32-bits long**

- A subset of these registers is accessible in each mode and does not have to be preserved
- Note: System mode uses the User mode register set.

# Handling exceptions

- **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>

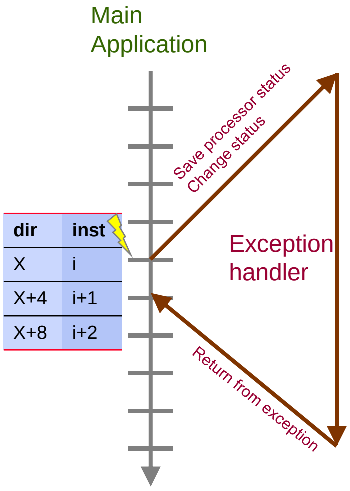
| | |
|---|---|
| 0x1C | FIQ |
| 0x18 | IRQ B irq_handler |
| 0x14 | (Reserved) |
| 0x10 | Data Abort |
| 0x0C | Prefetch Abort |
| 0x08 | Supervisor Call |
| 0x04 | Undefined Instruction |
| 0x00 | Reset |

Vector Table

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

This can only be done in ARM state.

# Handling exceptions

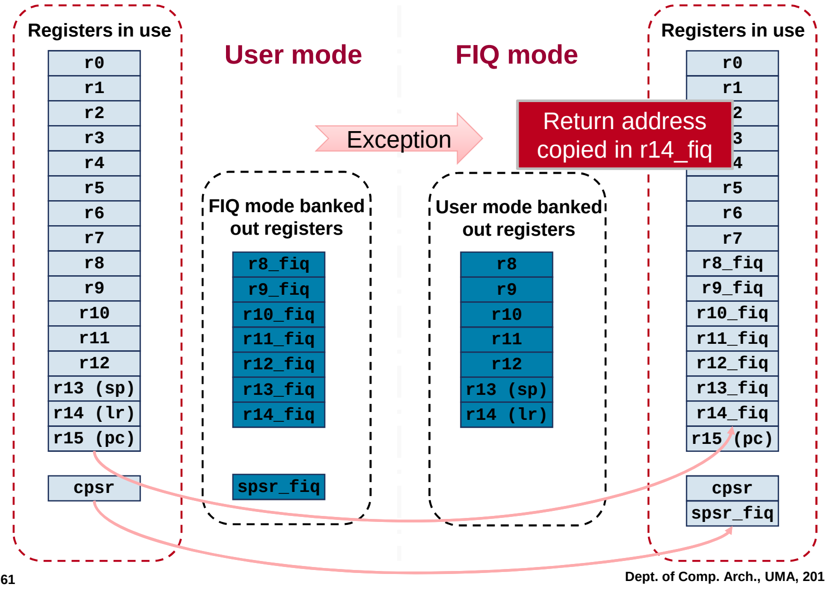


| dir | inst |
|---|---|
| X | i |
| X+4 | i+1 |
| X+8 | i+2 |

1. **Save processor status**
   - 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**
   - Forces the CPSR mode bits to a value (depends on the exception)
   - Sets PC to vector address
3. **Execute exception handler**
   - <user code>
4. **Return to main application**
   - 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**

## User mode → FIQ mode



## Exception handler

❑ Basic structure of a exception handler

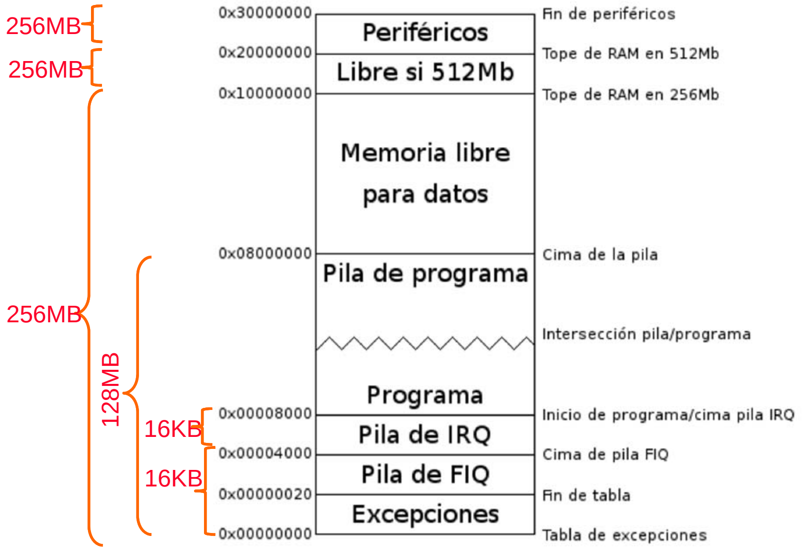
- Interruption: the return is done by lr-4
- Internal exception (as data abort): the return is done by lr-8
- User must manage A, I and F flags to disable/enable nesting of new exceptions and interruptions.
  - Initially the interruptions are disabled (I=F=1).

```
irq_handler:
```

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

## Memory map

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



256MB
256MB
256MB
128MB
16KB
16KB

| Address | Region | Description |
|---|---|---|
| 0x30000000 | | Fin de periféricos |
| | Periféricos | |
| 0x20000000 | | Tope de RAM en 512Mb |
| | Libre si 512Mb | |
| 0x10000000 | | Tope de RAM en 256Mb |
| | Memoria libre para datos | |
| 0x08000000 | | Cima de la pila |
| | Pila de programa | |
| | | Intersección pila/programa |
| | Programa | |
| 0x00008000 | | Inicio de programa/cima pila IRQ |
| | Pila de IRQ | |
| 0x00004000 | | Cima de pila FIQ |
| | Pila de FIQ | |
| 0x00000020 | | Fin de tabla |
| | Excepciones | |
| 0x00000000 | | Tabla de excepciones |

## Main program: steps to set up the Interruptions

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
```

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

```
   sp_svc <- 0x08000000
```

❹ Configure GPIOs (I&O)

❺ Configure peripheral interruption: timer/push-buttons

❻ Local enabling of configured interrupts

❼Global enabling of interrupts (SVC mode)

❽ … (main program tasks)

## ❶ Initialize Vector Table

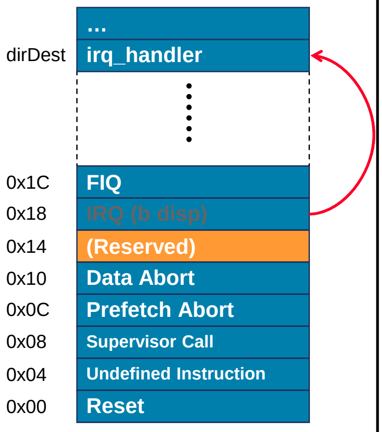
- To write in the Vector Table we can use a macro, ADDEXC, that computes the offset of the exception handler and writes the Vector in the Vector Table.

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

## macro ADDEXC offset, dirDest

- The IRQ handler is located at `dirDest`
- Vector table stores a branch instruction to the IRQ handler, `b disp`, located at `offset` (0x18)
- `disp` is the number of bytes between `dirDest` and `offset`, divided by 4
- While executing b `disp`, pc is incremented twice (pc = 0x18+8)
- Thus, `disp` must store

$$disp = \frac{(dirDest - (offset + 8))}{4}$$



| | |
|---|---|
| | ... |
| dirDest | irq_handler |
| | ⋮ |
| 0x1C | FIQ |
| 0x18 | IRQ (b disp) |
| 0x14 | (Reserved) |
| 0x10 | Data Abort |
| 0x0C | Prefetch Abort |
| 0x08 | Supervisor Call |
| 0x04 | Undefined Instruction |
| 0x00 | Reset |

Vector Table

# macro ADDEXC offset, dirDest

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ASM | b | | disp | | | | | |
| Binary | 1110 | 1010 | | | | | | |
| Hex | E | A | | | | | | |

$$disp = \frac{(dirDest - (offset + 8))}{4}$$

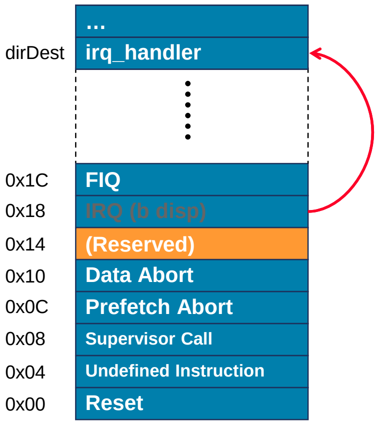
$$0xEA000000 + \frac{(dirDest-(offset+8))}{4} =$$

$$0xE9FFFFFE + \frac{(dirDest-offset)}{4} =$$

8 digits!

$$\frac{0x3A7FFFFF8 + (dirDest-offset)}{4}$$

We can divide by 4 by rotating right the numerator (after adding 3 to save the two most significant bits)



Vector Table

# macro ADDEXC offset, dirDest

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ASM | b | | disp | | | | | |
| Binary | 1110 | 1010 | | | | | | |
| Hex | E | A | | | | | | |

$disp =$

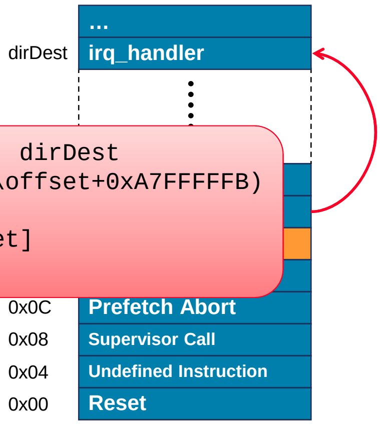
0xEA00

0xE9FFFFFE +

```
.macro    ADDEXC offset, dirDest
   ldr  r1, =(\dirDest-\offset+0xA7FFFFFB)
   ror  r1, #2
   str  r1, [r0, #\offset]
.endm
```

$$\frac{0x3A7FFFFF8 + (dirDest-offset)}{4}$$

We can divide by 4 by rotating right the numerator (after adding 3 to save the two most significant bits)



Vector Table

## ❷❸ Initialize the stack

- Each mode has its stack pointer (sp)
  - Change the mode (via cpsr_c)
  - Instructions msr (sr <- reg) and mrs (reg <-sr).
  - Initialize the corresponding sp register
- Initial state in BareMetal is SVC
  - sp_fiq=0x4000, sp_irq=0x8000,  sp_svc=0x08000000:

```
     mov    r0, #0 @ Pointer to vector table
     ADDEXC 0x18, irq_handler
     ADDEXC 0x1c, fiq_handler
     mov    r0, #0b11010001 @ FIQ mode, FIQ and IRQ disabled
❷    msr    cpsr_c, r0
     mov    sp, #0x4000
     mov    r0, #0b11010010 @ IRQ mode, FIQ and IRQ disabled
❷    msr    cpsr_c, r0
     mov    sp, #0x8000
     mov    r0, #0b11010011 @ SVC mode, FIQ and IRQ disabled
❸    msr    cpsr_c, r0
     mov    sp, #0x08000000
```

## ❺ Configure peripheral interruption

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

❺ **Configure push-button interruption (GPIO 2)**

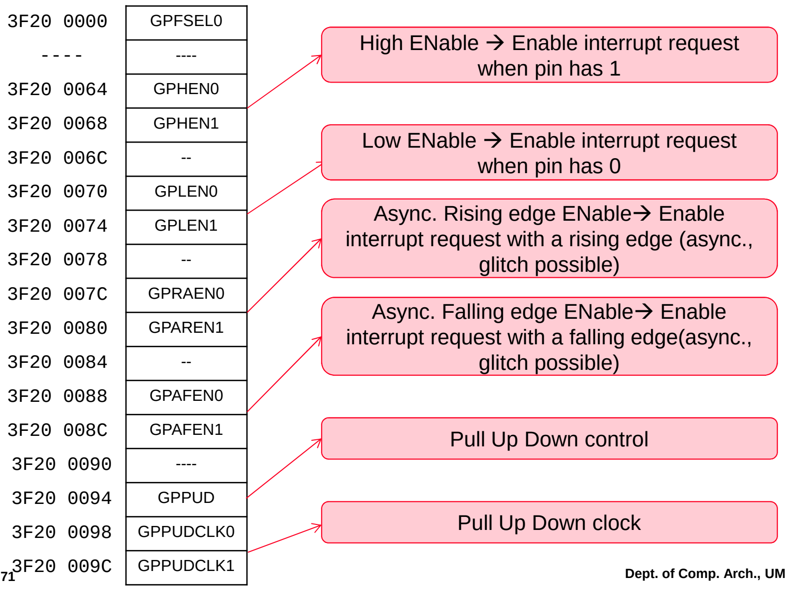
```
    ldr    r0,=GPBASE
    mov    r1, #0b00000000000000000000000000000100
    str    r1, [r0, #GPFEN0]
```

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

❺ **Configure timer IRQ**

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

## GPIO memory mapping



| | |
|---|---|
| 3F20 0000 | GPFSEL0 |
| ---- | ---- |
| 3F20 0064 | GPHEN0 |
| 3F20 0068 | GPHEN1 |
| 3F20 006C | -- |
| 3F20 0070 | GPLEN0 |
| 3F20 0074 | GPLEN1 |
| 3F20 0078 | -- |
| 3F20 007C | GPRAEN0 |
| 3F20 0080 | GPAREN1 |
| 3F20 0084 | -- |
| 3F20 0088 | GPAFEN0 |
| 3F20 008C | GPAFEN1 |
| 3F20 0090 | ---- |
| 3F20 0094 | GPPUD |
| 3F20 0098 | GPPUDCLK0 |
| 3F20 009C | GPPUDCLK1 |

High ENable → Enable interrupt request when pin has 1

Low ENable → Enable interrupt request when pin has 0

Async. Rising edge ENable→ Enable interrupt request with a rising edge (async., glitch possible)

Async. Falling edge ENable→ Enable interrupt request with a falling edge(async., glitch possible)

Pull Up Down control

Pull Up Down clock

## ❷ Source of interruption?

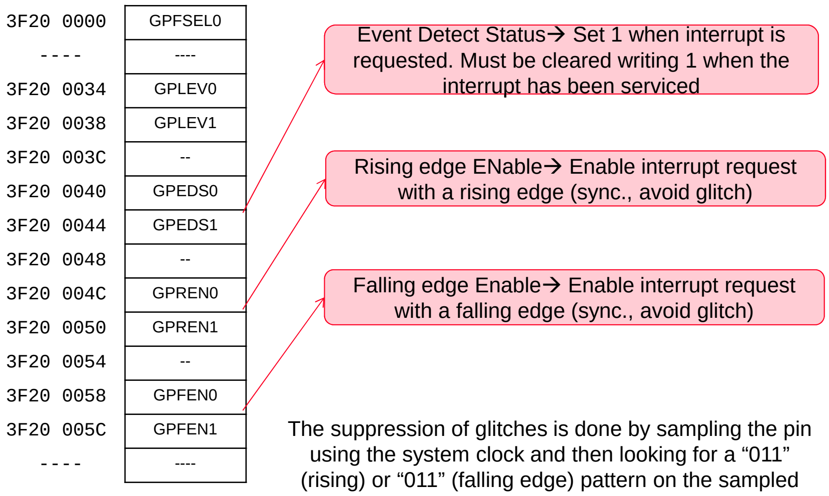
- ❑ In case of interruption, the handler must identify the source reading the IRQ pending ports
  - GPIO interruption detection: use GPEDSn.

```
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?
```

# GPIO memory mapping



| | |
|---|---|
| 3F20 0000 | GPFSEL0 |
| ---- | ---- |
| 3F20 0034 | GPLEV0 |
| 3F20 0038 | GPLEV1 |
| 3F20 003C | -- |
| 3F20 0040 | GPEDS0 |
| 3F20 0044 | GPEDS1 |
| 3F20 0048 | -- |
| 3F20 004C | GPREN0 |
| 3F20 0050 | GPREN1 |
| 3F20 0054 | -- |
| 3F20 0058 | GPFEN0 |
| 3F20 005C | GPFEN1 |
| ---- | ---- |

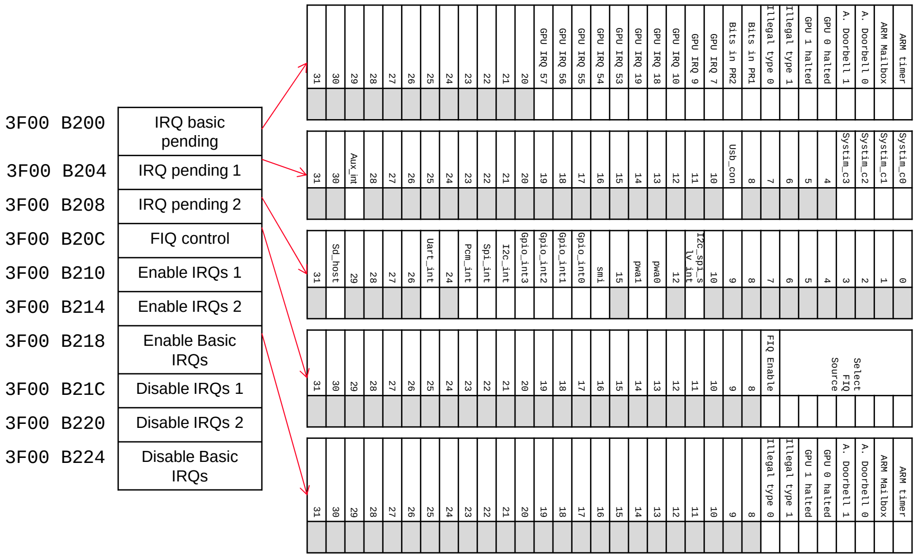
Event Detect Status→ Set 1 when interrupt is requested. Must be cleared writing 1 when the interrupt has been serviced

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 “011” (falling edge) pattern on the sampled signal.

# ❻ Local enabling of sources of interruption



| | |
|---|---|
| 3F00 B200 | IRQ basic pending |
| 3F00 B204 | IRQ pending 1 |
| 3F00 B208 | IRQ pending 2 |
| 3F00 B20C | FIQ control |
| 3F00 B210 | Enable IRQs 1 |
| 3F00 B214 | Enable IRQs 2 |
| 3F00 B218 | Enable Basic IRQs |
| 3F00 B21C | Disable IRQs 1 |
| 3F00 B220 | Disable IRQs 2 |
| 3F00 B224 | Disable Basic IRQs |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | GPU IRQ 57 | GPU IRQ 56 | GPU IRQ 55 | GPU IRQ 54 | GPU IRQ 53 | GPU IRQ 19 | GPU IRQ 18 | GPU IRQ 10 | GPU IRQ 9 | GPU IRQ 7 | Bits in PR2 | Bits in PR1 | Illegal type 0 | Illegal type 1 | GPU 1 halted | GPU 0 halted | A. Doorbell 1 | A. Doorbell 0 | ARM Mailbox | ARM timer |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | Aux_int | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | Usb_con | 8 | 7 | 6 | 5 | 4 | Systim_c3 | Systim_c2 | Systim_c1 | Systim_c0 |

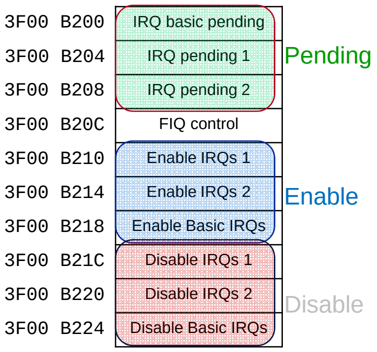
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | Sd_host | 29 | 28 | 27 | 26 | Uart_int | 24 | Pcm_int | Spi_int | I2c_int | Gpio_int3 | Gpio_int2 | Gpio_int1 | Gpio_int0 | smi | 15 | pwa1 | pwa0 | 12 | I2c_spi_slv_int | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | FIQ Enable | Select FIQ Source |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | Illegal type 0 | Illegal type 1 | GPU 1 halted | GPU 0 halted | A. Doorbell 1 | A. Doorbell 0 | ARM Mailbox | ARM timer |

# ❻ Local enabling of sources of interruption

- Three groups: pending, enable and disable



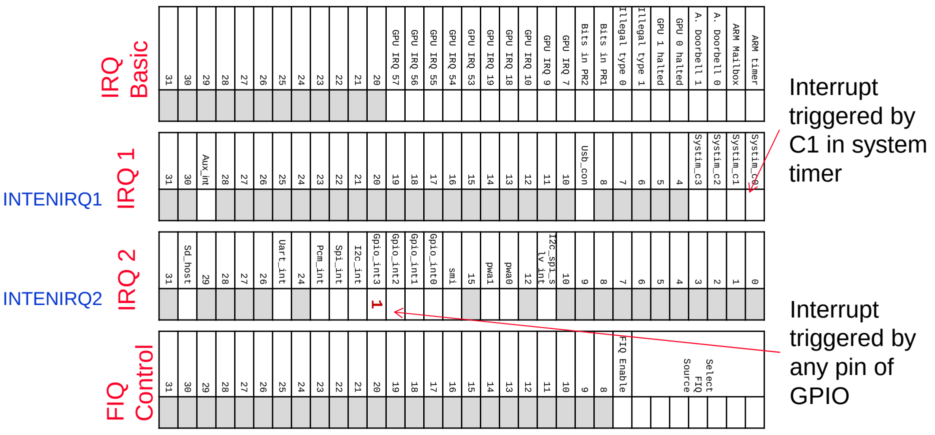
| | | |
|---|---|---|
| 3F00 B200 | IRQ basic pending | Pending |
| 3F00 B204 | IRQ pending 1 | |
| 3F00 B208 | IRQ pending 2 | |
| 3F00 B20C | FIQ control | |
| 3F00 B210 | Enable IRQs 1 | Enable |
| 3F00 B214 | Enable IRQs 2 | |
| 3F00 B218 | Enable Basic IRQs | |
| 3F00 B21C | Disable IRQs 1 | Disable |
| 3F00 B220 | Disable IRQs 2 | |
| 3F00 B224 | Disable Basic IRQs | |

In each group:

- IRQ basic: summary
- IRQs 1 and 2: in detail

There is also one port for FIQ control

# ❻ Local enabling of sources of interruption



| IRQ Basic | IRQ 1 (INTENIRQ1) | IRQ 2 (INTENIRQ2) | FIQ Control |
|---|---|---|---|
| 31 | 31 | 31 | 31 |
| 30 | 30 | Sd_host | 30 |
| 29 | Aux_int | 29 | 29 |
| 28 | 28 | 28 | 28 |
| 27 | 27 | 27 | 27 |
| 26 | 26 | 26 | 26 |
| 25 | 25 | Uart_int | 25 |
| 24 | 24 | 24 | 24 |
| 23 | 23 | Pcm_int | 23 |
| 22 | 22 | Spi_int | 22 |
| 21 | 21 | I2c_int | 21 |
| 20 | 20 | Gpio_int3<br>1 | 20 |
| GPU IRQ 57 | 19 | Gpio_int2 | 19 |
| GPU IRQ 56 | 18 | Gpio_int1 | 18 |
| GPU IRQ 55 | 17 | Gpio_int0 | 17 |
| GPU IRQ 54 | 16 | smi | 16 |
| GPU IRQ 53 | 15 | 15 | 15 |
| GPU IRQ 19 | 14 | pwa1 | 14 |
| GPU IRQ 18 | 13 | pwa0 | 13 |
| GPU IRQ 10 | 12 | 12 | 12 |
| GPU IRQ 9 | 11 | I2c_spi_slv_int | 11 |
| GPU IRQ 7 | 10 | 10 | 10 |
| Bits in PR2 | Usb_con | 9 | 9 |
| Bits in PR1 | 8 | 8 | 8 |
| Illegal type 0 | 7 | 7 | FIQ Enable |
| Illegal type 1 | 6 | 6 | Select FIQ Source |
| GPU 1 halted | 5 | 5 | |
| GPU 0 halted | 4 | 4 | |
| A. Doorbell 1 | Systim_c3 | 3 | |
| A. Doorbell 0 | Systim_c2 | 2 | |
| ARM Mailbox | Systim_c1 | 1 | |
| ARM timer | Systim_c0 | 0 | |

Interrupt triggered by C1 in system timer

Interrupt triggered by any pin of GPIO

## Example 5: turn on a red led after 4 seconds

```
            .include “inter.inc”
.text
            mov     r0, #0
            ADDEXC 0x18, irq_handler
```

Initialize vector table ❶

```
            . . .
            ldr     r0, =GPBASE
            ldr     r1, =0b00001000000000000000000000000000
            str     r1, [r0, #GPFSEL0]
```

Set GPIO9 as Output ❹

```
            ldr     r0, =STBASE
            ldr     r1, [r0, #STCLO]
            add     r1, #0x400000   @ 4.19 seconds
            str     r1, [r0, #STC1]
```

Load CLO, add 4 sec and store result in C1 ❺

```
            ldr     r0, =INTBASE
            mov     r1, #0b0010
            str     r1, [r0, #INTENIRQ1]
```

Enable C1 interruption ❻

```
            mov     r0, #0b01010011 @ SVC mode, IRQ enabled
            msr     cpsr_c, r0
```

Enable I flag ❼

```
buc:        b       buc
irq_handler:
            push    {r0, r1}
```

❶

```
            ldr     r0, =GPBASE
            mov     r1, #0b00000000000000000000001000000000
            str     r1, [r0, #GPSET0]
```

Turn on RED led (GPIO9) ❸

```
            pop     {r0, r1}
```

PC← LR - 4 ❻

```
            subs    pc, lr, #4
```

❺

## Example 5: inter.inc file

```
.macro    ADDEXC  vector, dirRTI
        ldr     r1, =(\dirRTI-\vector+0xa7fffffb)
        ror     r1, #2
        str     r1, [r0, #\vector]
.endm
        .set    GPBASE,    0x3F200000
        .set    GPFSEL0,         0x00
        .set    GPFSEL1,         0x04
        .set    GPFSEL2,         0x08
        .set    GPFSEL3,         0x0c
        .set    GPFSEL4,         0x10
        .set    GPFSEL5,         0x14
        .set    GPFSEL6,         0x18
        .set    GPSET0,          0x1c
        .set    GPSET1,          0x20
        .set    GPCLR0,          0x28
        .set    GPCLR1,          0x2c
        .set    GPLEV0,          0x34
        .set    GPLEV1,          0x38
        .set    GPEDS0,          0x40
        .set    GPEDS1,          0x44
        .set    GPFEN0,          0x58
        .set    GPFEN1,          0x5c
        .set    GPPUD,           0x94
        .set    GPPUDCLK0,       0x98
```

GPIO

```
        .set    STBASE,    0x3F003000
        .set    STCS,            0x00
        .set    STCLO,           0x04
        .set    STC1,            0x10
        .set    STC3,            0x18
```

Timer

```
        .set    INTBASE,   0x3F00b000
        .set    INTFIQCON,      0x20c
        .set    INTENIRQ1,      0x210
        .set    INTENIRQ2,      0x214
```

Interrupt

# Example 6: turn on a red led after pushing a button

```
.include  "inter.inc"
.text
          mov     r0, #0
          ADDEXC  0x18, irq_handler
```

❶

```
          mov     r0, #0b11010010
          msr     cpsr_c, r0
          mov     sp, #0x8000
```

Stack init for IRQ mode ❷

```
          mov     r0, #0b11010011
          msr     cpsr_c, r0
          mov     sp, #0x8000000
```

Stack init for SVC mode ❸

```
          ldr     r0, =GPBASE
          mov     r1, #0b00001000000000000000000000000000
          str     r1, [r0, #GPFSEL0]
```

Set GPIO9 as output ❹

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

Configure falling edge interruptions through GPIO2 ❺

```
          ldr     r0, =INTBASE
          mov     r1, #0b00000000000100000000000000000000
          str     r1, [r0, #INTENIRQ2]
```

Allow interruptions from any GPIO pin ❻

```
          mov     r0, #0b01010011
          msr     cpsr_c, r0
```

Set SVC mode with IRQ enabled ❼

```
buc:      b       buc
```

❽

# Example 6: IRQ handler

```
irq_handler:
          push    {r0, r1, r2}
```

❶

```
          ldr     r0, =GPBASE
          ldr     r2, [r0, #GPEDS0]
          ands    r2, #0b00000000000000000000000000000100
```

Check GPIO2 was pressed ❷

```
          movne   r1, #0b00000000000000000000001000000000
          strne   r1, [r0, #GPSET0]
```

Turn on GPIO9 red led ❸

```
          movne   r1, #0b00000000000000000000000000000100
          strne   r1, [r0, #GPEDS0]
```

Clear GPIO2 event ❹

```
          pop     {r0, r1, r2}
```

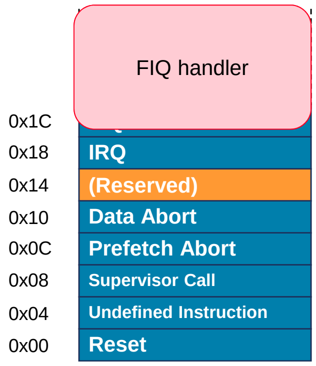
❺

```
          subs    pc, lr, #4
```

❻

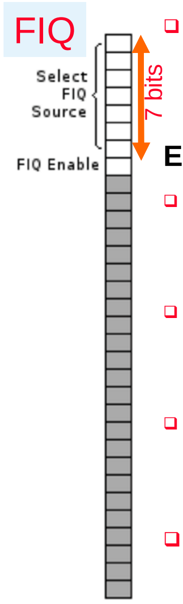
## Using FIQ

- ❑ Advantages:
  - Registers r8 to r14 are saved
  - FIQ handler can start after vector table: no need to branch
- ❑ Disadvantage: only one source of interruption can be handled



Vector Table

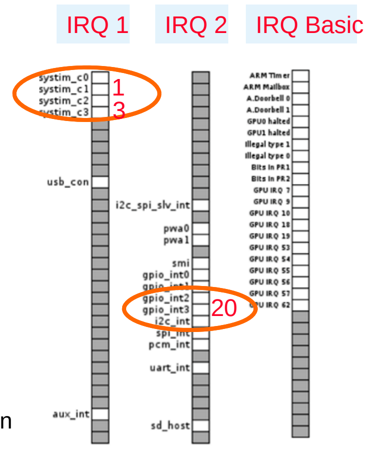
## Using FIQ



- ❑ 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 7: 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)

## Example 7: Timer in IRQ and push button in FIQ

```
.include  "inter.inc"
.text
            ADDEXC  0x18, irq_handler                                                  ❶
            ADDEXC  0x1c, fiq_handler
            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 7: IRQ and FIQ handlers

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

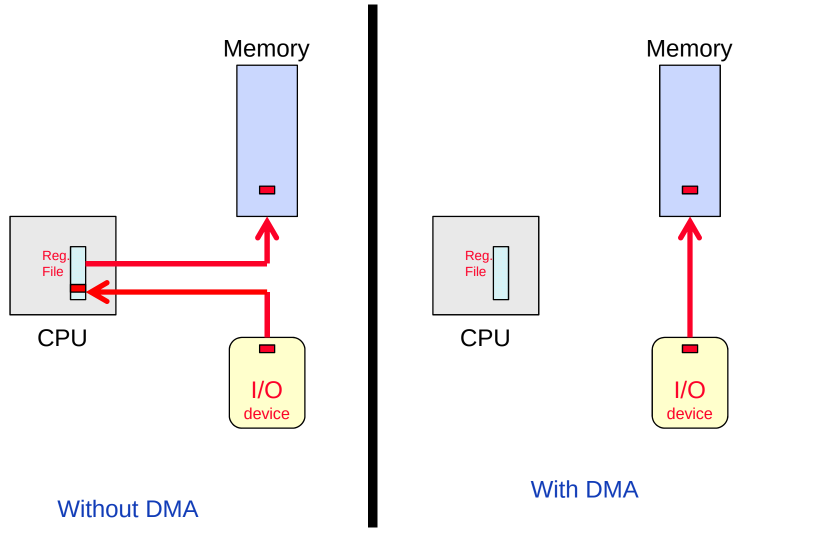
A variable

## 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)

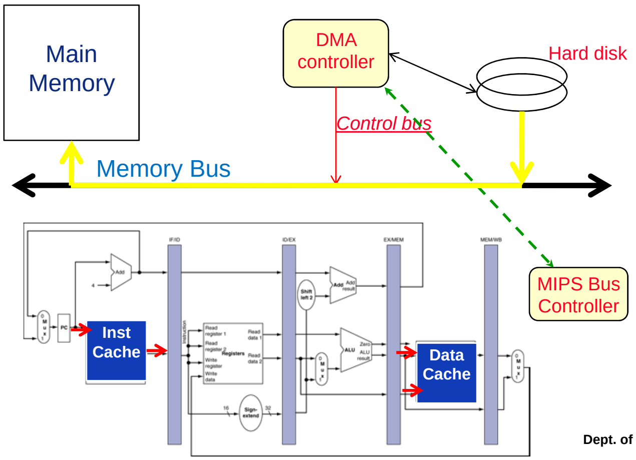


Without DMA

With DMA

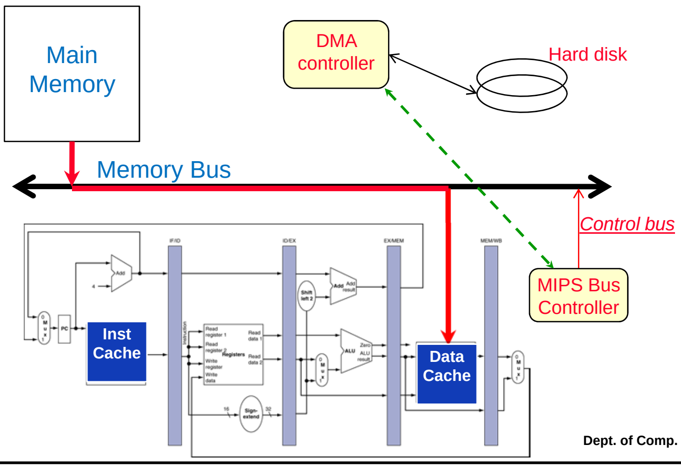
# Direct Memory Access (DMA)

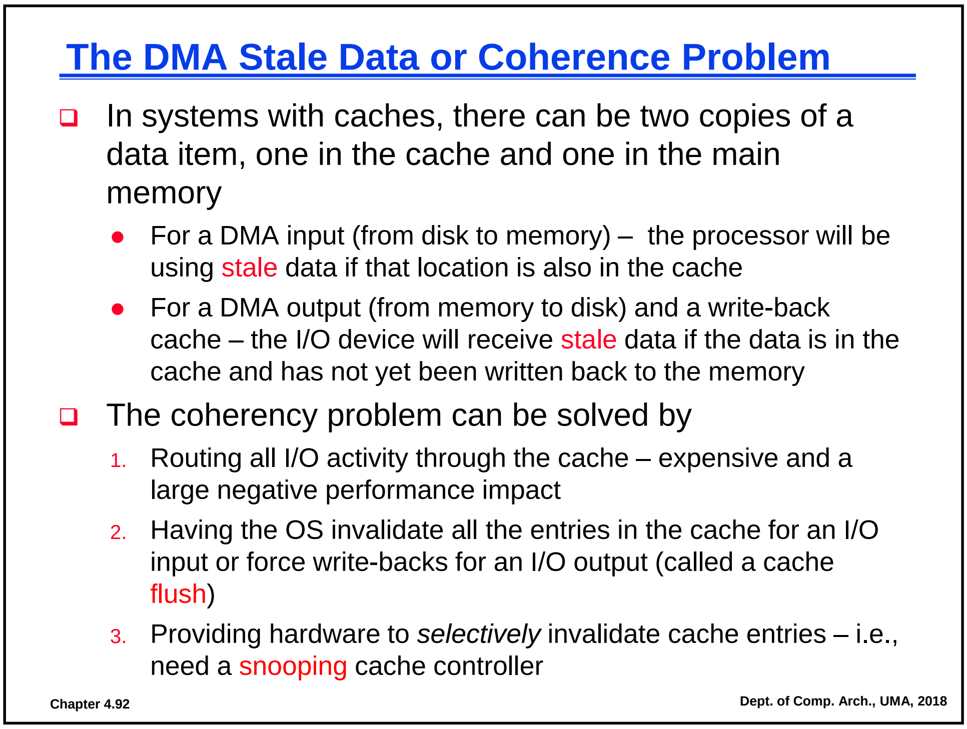
- Processor works in parallel with the DMA controller
  - Processor dealing with Cache Buses ( → )
  - DMA controler  dealing with Main Memory Bus



# Direct Memory Access (DMA)

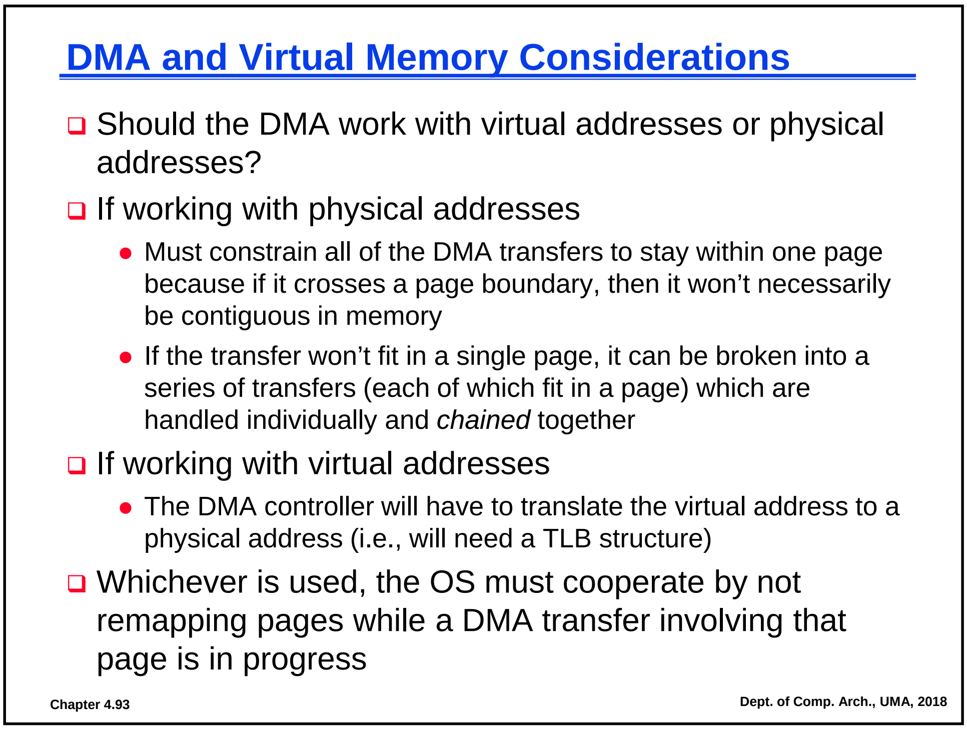
- Example of data cache miss (or updating in a write-through)
  - Processor is dealing with Main Memory Bus





## The DMA Stale Data or Coherence Problem

- In systems with caches, there can be two copies of a data item, one in the cache and one in the main memory
  - For a DMA input (from disk to memory) – the processor will be using stale data if that location is also in the cache
  - For a DMA output (from memory to disk) and a write-back cache – the I/O device will receive stale data if the data is in the cache and has not yet been written back to the memory
- The coherency problem can be solved by
  1. Routing all I/O activity through the cache – expensive and a large negative performance impact
  2. Having the OS invalidate all the entries in the cache for an I/O input or force write-backs for an I/O output (called a cache flush)
  3. Providing hardware to *selectively* invalidate cache entries – i.e., need a snooping cache controller



## DMA and Virtual Memory Considerations

- Should the DMA work with virtual addresses or physical addresses?
- If working with physical addresses
  - Must constrain all of the DMA transfers to stay within one page because if it crosses a page boundary, then it won't necessarily be contiguous in memory
  - If the transfer won't fit in a single page, it can be broken into a series of transfers (each of which fit in a page) which are handled individually and *chained* together
- If working with virtual addresses
  - The DMA controller will have to translate the virtual address to a physical address (i.e., will need a TLB structure)
- Whichever is used, the OS must cooperate by not remapping pages while a DMA transfer involving that page is in progress

## More intelligent controllers: I/O processors

- To further reduce the need to interrupt the processor the I/O controller can be made more intelligent: *I/O processors (I/O controllers* or *channel controllers*)
  - they execute a series of I/O operations (*I/O program* is stored in the I/O processor or in memory and fetched by the I/O processor) and interrupts the processor only when the entire program is completed.
  - The I/O program is setted up by the OS: I/O operations to be done, the size and transfer address for any reads or writes
- DMA processors are essentially special-purpose processors (single-chip and nonprogrammable), while I/O processors are often implemented with general-purpose microprocessors, which run a specialized I/O program

## Interfacing I/O Devices to the Processor, Memory, and OS

- The operating system acts as the interface between the I/O hardware and the program requesting I/O
  - It provides equitable access to the shared I/O resources, protects those I/O devices/activities to which a user program doesn’t have access, and schedules I/O requests to enhance system throughput
  - It handles interrupts generated by I/O devices
  - It supplies routines for low-level I/O device operations
    - OS must be able to give commands to the I/O devices
    - I/O device must be able to notify the OS about its status
    - Must be able to transfer data between the memory and the I/O device
- Software that communicates with an I/O device is called a **device driver,** and requires detailed knowledge about the I/O device hardware (ports list and its behavior)