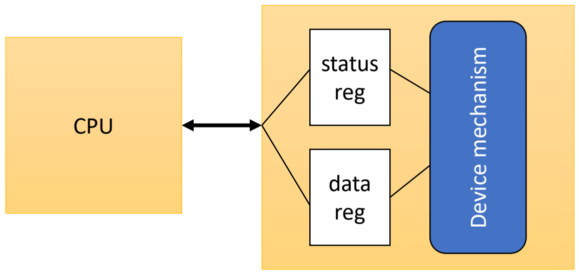
CPUs

COEN 421/6341: Embedded Systems Design

1

Programming Input and Output

- Input and output devices usually have some analog or nonelectronic component
  - e.g., disk drive has a rotating disk and analog read/write electronics
- Typical digital interface to CPU



- **Status reg.:** provide information about the device's operation (e.g., whether the current transaction has completed)
- **Data reg.:** hold values that are treated as data by the device (e.g., temperature)

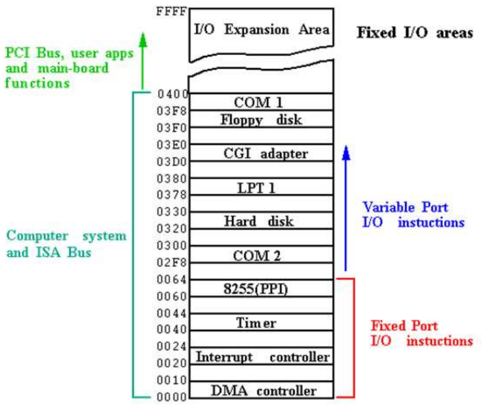
2

Input and Output Primitives

- Microprocessors can provide programming support for input and output in two ways
  - 1) I/O instructions
    - Some architectures, such as the Intel x86, provide special instructions (in and out in the case of the Intel x86) for input and output
    - ***IN*** and ***OUT*** transfer data between an I/O device and the microprocessor's accumulator (AL, AX or EAX)
    - ***IN*** *AL*, 19H;8-bits are saved to AL from I/O port 19H

Input and Output Primitives

- Microprocessors can provide programming support for input and output in two ways
  - 2) Memory-mapped I/O
    - Provides addresses for the registers in each I/O device
    - It is the most common way to implement I/O is by memory mapping

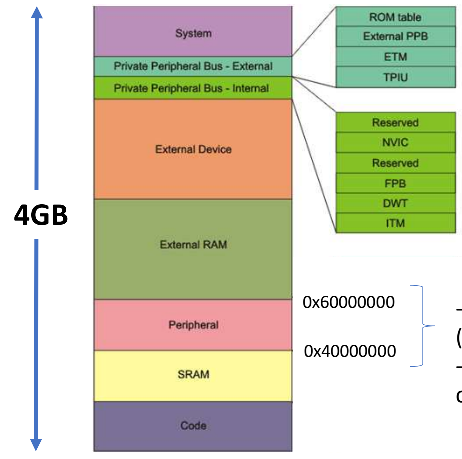


I/O Primitives / MMIO

- ARM Cortex-M processors use MMIO to access peripheral registers
- Peripheral registers are mapped to a small memory region:
  - Starting at 0x40000000 on STM32L4

I/O Primitives / MMIO

- ARM Cortex-M processors:



0x60000000

0x40000000

- Includes the memory address of each peripheral register (e.g., GPIO, timers, UAER, SPI, ADC)
- Memory address of each peripheral register is determined by chip manufacturers

I/O Primitives / MMIO

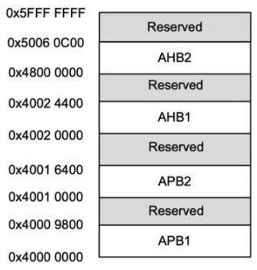
- ARM Cortex-M processors use MMIO to access peripheral registers
- Peripheral registers are mapped to a small memory region:
  - Starting at 0x40000000 on STM32L4
    - Includes the memory addresses of all on-chip peripherals:
      - GPIO
      - Timers
      - UART
      - SPI
      - ADC
      - …

I/O Primitives / MMIO

- ARM Cortex-M processors use MMIO to access peripheral registers
- Peripheral registers are mapped to a small memory region:
  - Starting at 0x40000000 on STM32L4
    - Includes the memory addresses of all on-chip peripherals:
- Address of each peripheral register:
  - Determined by chip manufacturers

I/O Primitives / MMIO

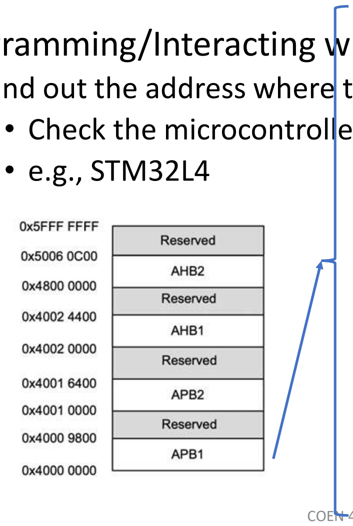
- Programming/Interacting with I/O peripherals
  - Find out the address where the peripheral is memory mapped
    - Check the microcontroller datasheet, which provides a memory map
    - e.g., STM32L4



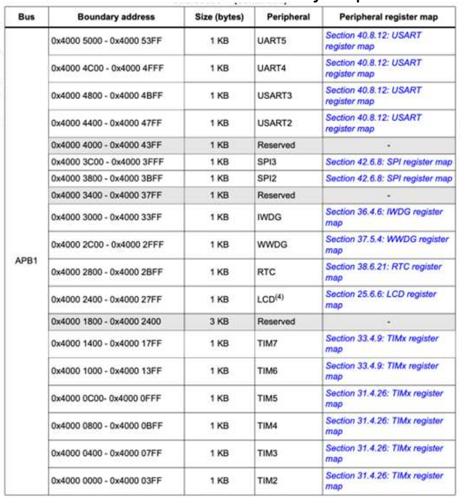
I/O Primitives / MMIO

- Programming/Interacting with I
  - Find out the address where the p
    - Check the microcontroller da
    - e.g., STM32L4

nap



*just part of it



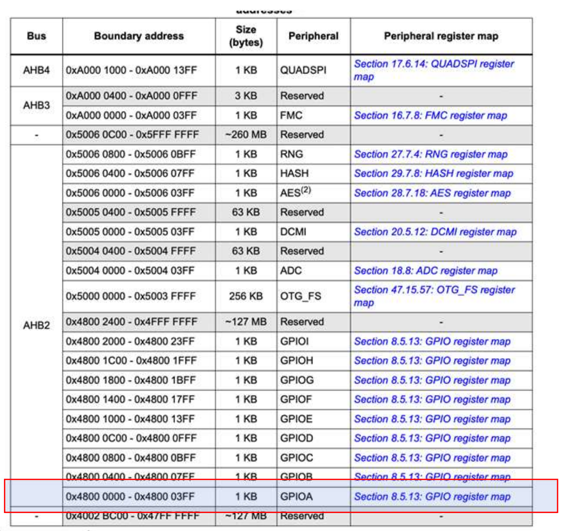
Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
APB1	0x4000 5000 - 0x4000 53FF	1 KB	UART5	Section 40.8.12: USART register map
	0x4000 4C00 - 0x4000 4FFF	1 KB	UART4	Section 40.8.12: USART register map
	0x4000 4800 - 0x4000 4BFF	1 KB	USART3	Section 40.8.12: USART register map
	0x4000 4400 - 0x4000 47FF	1 KB	USART2	Section 40.8.12: USART register map
	0x4000 4000 - 0x4000 43FF	1 KB	Reserved	-
	0x4000 3C00 - 0x4000 3FFF	1 KB	SPI3	Section 42.6.8: SPI register map
	0x4000 3800 - 0x4000 3BFF	1 KB	SPI2	Section 42.6.8: SPI register map
	0x4000 3400 - 0x4000 37FF	1 KB	Reserved	-
	0x4000 3000 - 0x4000 33FF	1 KB	IWDG	Section 36.4.6: IWDG register map
	0x4000 2C00 - 0x4000 2FFF	1 KB	WWDG	Section 37.5.4: WWDG register map
	0x4000 2800 - 0x4000 2BFF	1 KB	RTC	Section 38.6.21: RTC register map
	0x4000 2400 - 0x4000 27FF	1 KB	LCD(4)	Section 25.6.6: LCD register map
	0x4000 1800 - 0x4000 2400	3 KB	Reserved	-
	0x4000 1400 - 0x4000 17FF	1 KB	TIM7	Section 33.4.9: TIMx register map
	0x4000 1000 - 0x4000 13FF	1 KB	TIM6	Section 33.4.9: TIMx register map
	0x4000 0C00- 0x4000 0FFF	1 KB	TIM5	Section 31.4.26: TIMx register map
	0x4000 0800 - 0x4000 0BFF	1 KB	TIM4	Section 31.4.26: TIMx register map
	0x4000 0400 - 0x4000 07FF	1 KB	TIM3	Section 31.4.26: TIMx register map
	0x4000 0000 - 0x4000 03FF	1 KB	TIM2	Section 31.4.26: TIMx register map

I/O Primitives / MMIO

- Programming/Interacting with I/O peripherals
  - Find out the address where the peripheral is memory mapped
    - Check the microcontroller datasheet, which provides a memory map
  - Find out the exact register address in order to program the registers
    - Check the peripheral datasheet
      - Control registers
      - Data registers
      - status registers
      - …

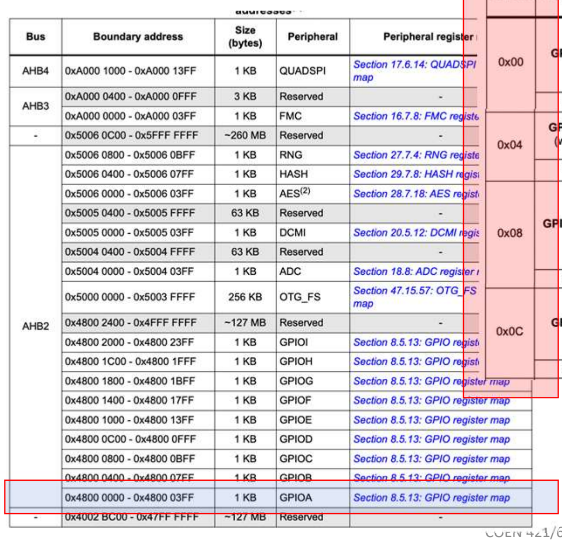
I/O Primitives / MMIO

- Programming/Interacting with I/O peripherals
  - Find out the address where the peripheral is memory mapped
    - Check the microcontroller datasheet, which provides a memory map
  - Find out the exact register address in order to program the registers
    - e.g., STM32L4 GPIOA registers

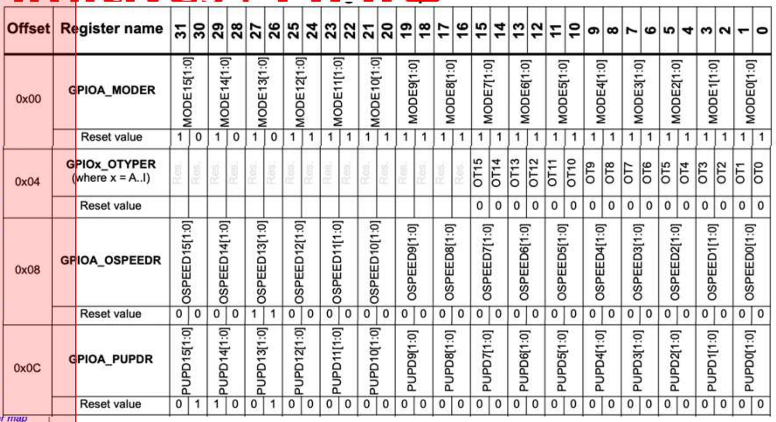


Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
AHB4	0xA000 1000 - 0xA000 13FF	1 KB	QUADSPI	Section 17.6.14: QUADSPI register map
AHB3	0xA000 0400 - 0xA000 0FFF	3 KB	Reserved	-
	0xA000 0000 - 0xA000 03FF	1 KB	FMC	Section 16.7.8: FMC register map
-	0x5006 0C00 - 0x5FFF FFFF	~260 MB	Reserved	-
AHB2	0x5006 0800 - 0x5006 0BFF	1 KB	RNG	Section 27.7.4: RNG register map
	0x5006 0400 - 0x5006 07FF	1 KB	HASH	Section 29.7.8: HASH register map
	0x5006 0000 - 0x5006 03FF	1 KB	AES(2)	Section 28.7.18: AES register map
	0x5005 0400 - 0x5005 FFFF	63 KB	Reserved	-
	0x5005 0000 - 0x5005 03FF	1 KB	DCMI	Section 20.5.12: DCMI register map
	0x5004 0400 - 0x5004 FFFF	63 KB	Reserved	-
	0x5004 0000 - 0x5004 03FF	1 KB	ADC	Section 18.8: ADC register map
	0x5000 0000 - 0x5003 FFFF	256 KB	OTG_FS	Section 47.15.57: OTG_FS register map
	0x4800 2400 - 0x4FFF FFFF	~127 MB	Reserved	-
	0x4800 2000 - 0x4800 23FF	1 KB	GPIOI	Section 8.5.13: GPIO register map
	0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH	Section 8.5.13: GPIO register map
	0x4800 1800 - 0x4800 1BFF	1 KB	GPIOG	Section 8.5.13: GPIO register map
	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF	Section 8.5.13: GPIO register map
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE	Section 8.5.13: GPIO register map
	0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD	Section 8.5.13: GPIO register map
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC	Section 8.5.13: GPIO register map
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB	Section 8.5.13: GPIO register map
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA	Section 8.5.13: GPIO register map
-	0x4002 BC00 - 0x47FF FFFF	~127 MB	Reserved	-

I/O Primitives / MMIO



Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register
AHB4	0xA000 1000 - 0xA000 13FF	1 KB	QUADSPI	Section 17.6.14: QUADSPI map
AHB3	0xA000 0400 - 0xA000 0FFF	3 KB	Reserved	-
	0xA000 0000 - 0xA000 03FF	1 KB	FMC	Section 16.7.8: FMC registe
-	0x5006 0C00 - 0x5FFF FFFF	~260 MB	Reserved	-
AHB2	0x5006 0800 - 0x5006 0BFF	1 KB	RNG	Section 27.7.4: RNG registe
	0x5006 0400 - 0x5006 07FF	1 KB	HASH	Section 29.7.8: HASH regis
	0x5006 0000 - 0x5006 03FF	1 KB	AES(2)	Section 28.7.18: AES regist
	0x5005 0400 - 0x5005 FFFF	63 KB	Reserved	-
	0x5005 0000 - 0x5005 03FF	1 KB	DCMI	Section 20.5.12: DCMI regis
	0x5004 0400 - 0x5004 FFFF	63 KB	Reserved	-
	0x5004 0000 - 0x5004 03FF	1 KB	ADC	Section 18.8: ADC register
	0x5000 0000 - 0x5003 FFFF	256 KB	OTG_FS	Section 47.15.57: OTG_FS map
	0x4800 2400 - 0x4FFF FFFF	~127 MB	Reserved	-
	0x4800 2000 - 0x4800 23FF	1 KB	GPIOI	Section 8.5.13: GPIO regist
	0x4800 1C00 - 0x4800 1FFF	1 KB	GPIOH	Section 8.5.13: GPIO regist
	0x4800 1800 - 0x4800 1BFF	1 KB	GPIOG	Section 8.5.13: GPIO register map
	0x4800 1400 - 0x4800 17FF	1 KB	GPIOF	Section 8.5.13: GPIO register map
	0x4800 1000 - 0x4800 13FF	1 KB	GPIOE	Section 8.5.13: GPIO register map
	0x4800 0C00 - 0x4800 0FFF	1 KB	GPIOD	Section 8.5.13: GPIO register map
	0x4800 0800 - 0x4800 0BFF	1 KB	GPIOC	Section 8.5.13: GPIO register map
	0x4800 0400 - 0x4800 07FF	1 KB	GPIOB	Section 8.5.13: GPIO register map
	0x4800 0000 - 0x4800 03FF	1 KB	GPIOA	Section 8.5.13: GPIO register map
-	0x4002 BC00 - 0x47FF FFFF	~127 MB	Reserved	-



Offset	Register name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOA_MODER	MODE15[1:0]		MODE14[1:0]		MODE13[1:0]		MODE12[1:0]		MODE11[1:0]		MODE10[1:0]		MODE9[1:0]		MODE8[1:0]		MODE7[1:0]		MODE6[1:0]		MODE5[1:0]		MODE4[1:0]		MODE3[1:0]		MODE2[1:0]		MODE1[1:0]		MODE0[1:0]	
	Reset value	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x04	GPIOx_OTYPER (where x = A..I)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	GPIOA_OSPEEDR	OSPEED15[1:0]		OSPEED14[1:0]		OSPEED13[1:0]		OSPEED12[1:0]		OSPEED11[1:0]		OSPEED10[1:0]		OSPEED9[1:0]		OSPEED8[1:0]		OSPEED7[1:0]		OSPEED6[1:0]		OSPEED5[1:0]		OSPEED4[1:0]		OSPEED3[1:0]		OSPEED2[1:0]		OSPEED1[1:0]		OSPEED0[1:0]	
	Reset value	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	GPIOA_PUPDR	PUPD15[1:0]		PUPD14[1:0]		PUPD13[1:0]		PUPD12[1:0]		PUPD11[1:0]		PUPD10[1:0]		PUPD9[1:0]		PUPD8[1:0]		PUPD7[1:0]		PUPD6[1:0]		PUPD5[1:0]		PUPD4[1:0]		PUPD3[1:0]		PUPD2[1:0]		PUPD1[1:0]		PUPD0[1:0]	
	Reset value	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

…

13

I/O Primitives / MMIO

Example 3.1 Memory-Mapped I/O on ARM

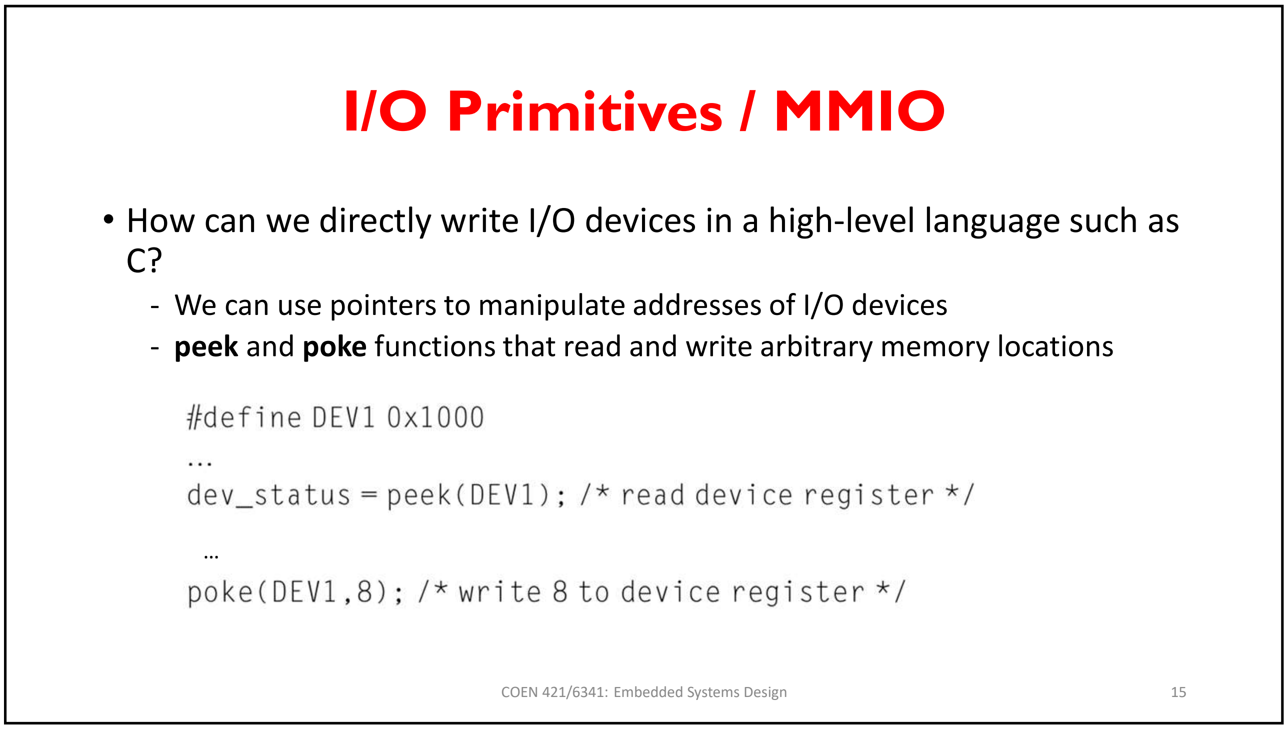
We can use the EQU pseudo-op to define a symbolic name for the memory location of our I/O device:

```
DEV1    EQU 0x1000
```

Given that name, we can use the following standard code to read and write the device register:

```
LDR r1,#DEV1        ; set up device address
LDR r0,[r1]         ; read DEV1
LDR r0,#8           ; set up value to write

STR r0,[r1]         ; write 8 to device
```

14



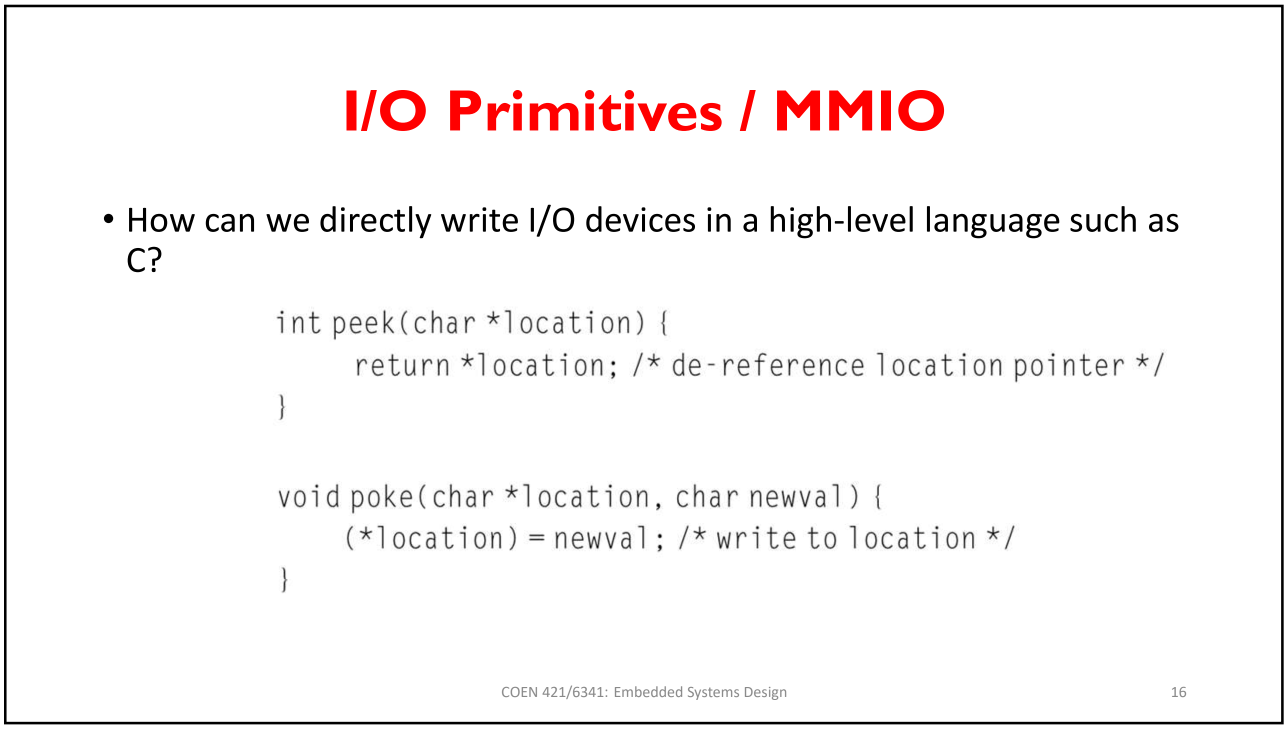
I/O Primitives / MMIO

- How can we directly write I/O devices in a high-level language such as C?
  - We can use pointers to manipulate addresses of I/O devices
  - **peek** and **poke** functions that read and write arbitrary memory locations

```
#define DEV1 0x1000
...
dev_status = peek(DEV1); /* read device register */
  …
poke(DEV1,8); /* write 8 to device register */
```

COEN 421/6341: Embedded Systems Design 15

15



I/O Primitives / MMIO

- How can we directly write I/O devices in a high-level language such as C?

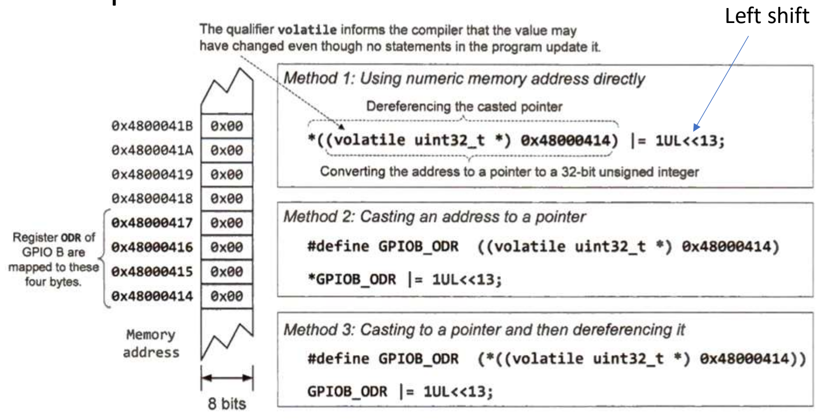
```
int peek(char *location) {
      return *location; /* de-reference location pointer */
}

void poke(char *location, char newval) {
     (*location) = newval; /* write to location */
}
```

COEN 421/6341: Embedded Systems Design 16

16

I/O Primitives / MMIO

- Another example:


Left shift
The qualifier volatile informs the compiler that the value may have changed even though no statements in the program update it.
Dereferencing the casted pointer
Converting the address to a pointer to a 32-bit unsigned integer
0x4800041B 0x00
0x4800041A 0x00
0x48000419 0x00
0x48000418 0x00
0x48000417 0x00
0x48000416 0x00
0x48000415 0x00
0x48000414 0x00
Register ODR of GPIO B are mapped to these four bytes.
Memory address
8 bits

Method 1: Using numeric memory address directly

```
*((volatile uint32_t *) 0x48000414) |= 1UL<<13;
```

Method 2: Casting an address to a pointer

```
#define GPIOB_ODR  ((volatile uint32_t *) 0x48000414)
*GPIOB_ODR |= 1UL<<13;
```

Method 3: Casting to a pointer and then dereferencing it

```
#define GPIOB_ODR  (*((volatile uint32_t *) 0x48000414))
GPIOB_ODR |= 1UL<<13;
```

17

Busy-Wait I/O

- Simplest way to communicate with devices
- Devices are typically slower than the CPU and may require many cycles to complete an operation
- CPU must wait for one I/O operation to complete before starting the next one when performing multiple operations on a single device
- **Polling**
  - It is the process of asking an I/O device whether it is finished
  - This is done by reading the device's status register

18

Busy-Wait I/O

```
#define OUT_CHAR 0x1000 /* output device character register */
#define OUT_STATUS 0x1001 /* output device status register */

char *mystring = "Hello, world." /* string to write */
char *current_char; /* pointer to current position in string */

current_char = mystring; /* point to head of string */
while (*current_char != '\0') { /* until null character */
     poke(OUT_CHAR,*current_char); /* send character to device */
     while (peek(OUT_STATUS) != 0); /* keep checking status */
     current_char++; /* update character pointer */
}
```

Busy-Wait I/O

- Copying Characters from Input to Output using Busy-Wait I/O

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
```

- Input device status register
  - 1: new character is ready
  - 0: new character has been read
- Output device status register
  - 1: start writing the character
  - 0: writing operation has been concluded

Busy-Wait I/O

- Copying Characters from Input to Output using Busy-Wait I/O

```
while (TRUE) { /* perform operation forever */
    /* read a character into achar */
    while (peek(IN_STATUS) == 0); /* wait until ready */
    achar = (char)peek(IN_DATA); /* read the character */
    /* write achar */
    poke(OUT_DATA,achar);
    poke(OUT_STATUS,1); /* turn on device */
    while (peek(OUT_STATUS) != 0); /* wait until done */
}
```

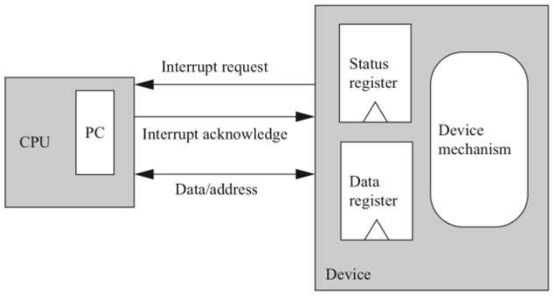
21

Interrupts

- Busy-wait I/O is extremely inefficient
  - CPU does nothing but test the device status while the I/O transaction is in progress
- Interrupt
  - Allows the processor to perform multiple tasks “simultaneously”
  - Allows devices to signal the CPU and to force execution of a particular piece of code
  - When an interrupt occurs:
    - The interrupt mechanism saves the value of the PC
    - The program counter’s value is changed to point to an **interrupt handler** routine (device driver)

22

Interrupts



- The I/O device’s logic decides when to interrupt
- The CPU may not be able to immediately service an interrupt request because it may be doing something else that must be finished first
- When CPU acknowledge the interrupt, it changes the PC to point to the device’s handler
- The interrupt handler operates much like a subroutine, except that it is not called by the executing program

Interrupts

- The CPU checks the interrupt request line at the beginning of execution of every instruction
- The CPU handles the interrupt if the interrupt request has been asserted
- The CPU does not fetch the instruction pointed to by the PC
  - CPU sets the PC to a predefined location (i.e., the beginning of the interrupt handling – interrupt service routine)

Interrupt Service Routine

- Special subroutine
- Invoked automatically by the hardware in response to an interrupt
- Each ISR has a default implementation in the system startup code
- There is an ISR associated with each type of interrupt
- ISR are stored in a special memory part
- Interrupt vector table

Copying Characters from I to O with Basic Interrupts

- Assumption:
  - Write C functions that act as interrupt handlers
  - Functions will read and write status and data registers
  - Use of a global variable ***achar*** for the input handler to pass the character to the foreground program
  - Because the foreground program does not know when an interrupt occurs, we also use a global Boolean variable, ***gotchar***, to signal when a new character has been received
    - Foreground program: the program that runs when no interrupt is being handled

Copying Characters from I to O with Basic Interrupts

- Handlers

```
void input_handler() { /* get a character and put in global */
      achar = peek(IN_DATA); /* get character */
      gotchar = TRUE; /* signal to main program */
      poke(IN_STATUS,0); /* reset status to initiate next
transfer */
}
void output_handler() { /* react to character being sent */
    /* don't have to do anything */
}
```

Copying Characters from I to O with Basic Interrupts

- Main program

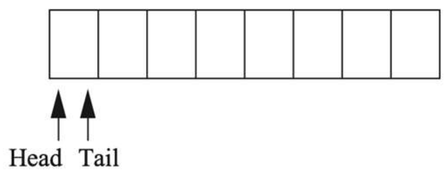
```
main() {
     while (TRUE) { /* read then write forever */
          if (gotchar) { /* write a character */
               poke(OUT_DATA,achar); /* put character in device */
               poke(OUT_STATUS,1); /* set status to initiate write */
               gotchar = FALSE; /* reset flag */
          }
     }
}
```

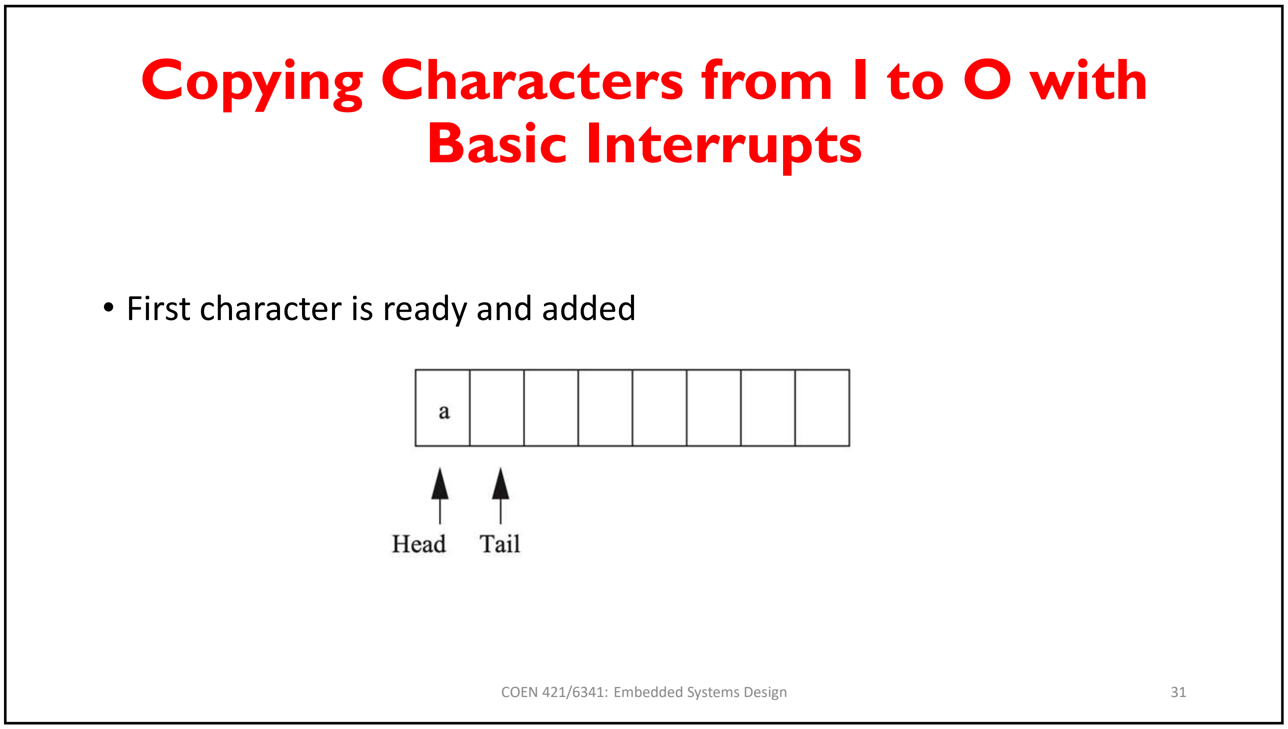
Copying Characters from I to O with Basic Interrupts

- The main program is somewhat simpler when interrupt handler was used
- However, it still does not let the foreground program do useful work
- Goal: Perform reads and writes independently
- Elastic buffer to hold the characters
  - It will allow the input and output devices to run at different rates

Copying Characters from I to O with Basic Interrupts

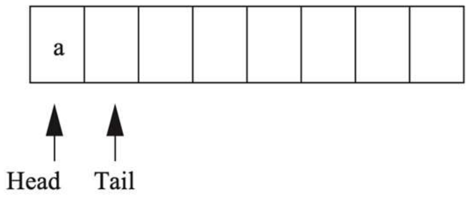
- Characters are added to the **tail** when received
- Characters are taken from the **head** when ready for output
- The head and tail wrap around the end of the buffer to make most efficient use of it



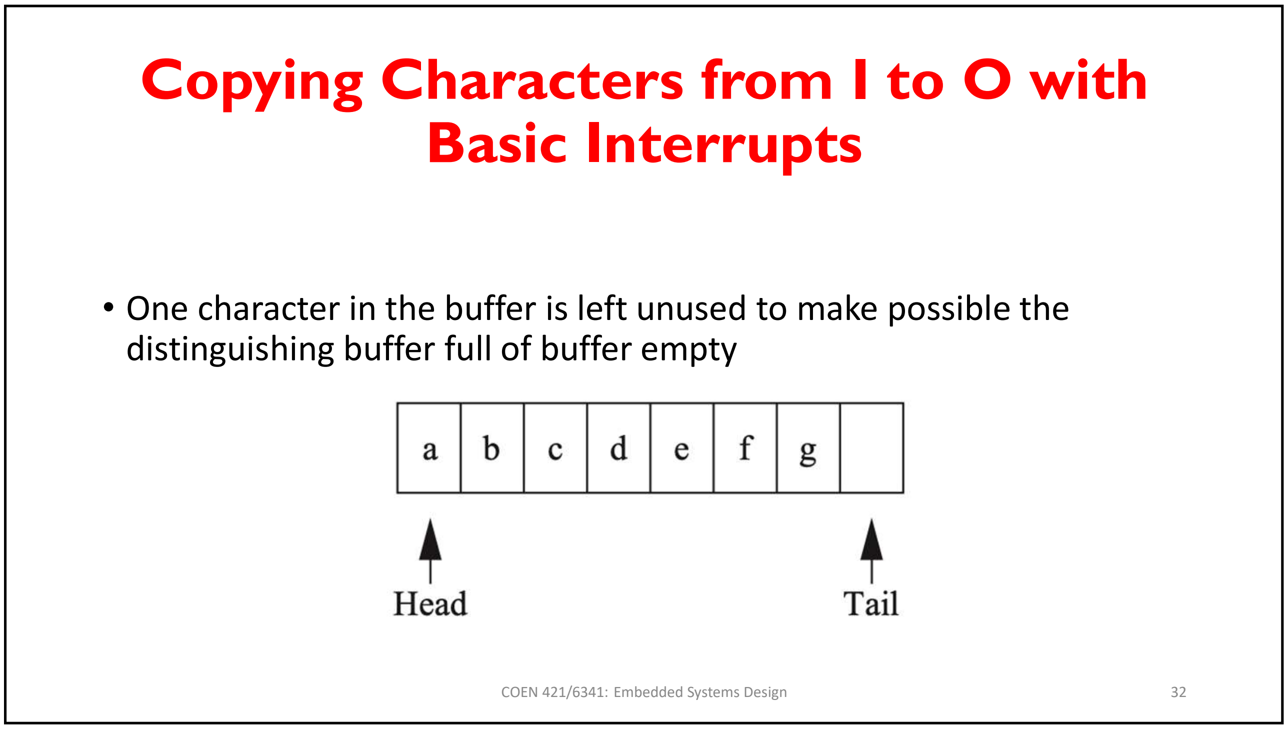


Copying Characters from I to O with Basic Interrupts

- First character is ready and added

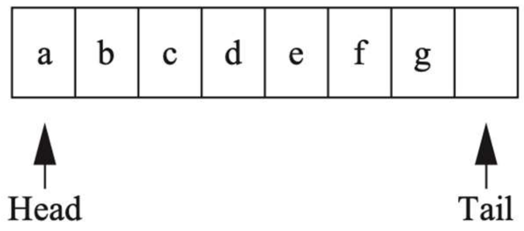


31



Copying Characters from I to O with Basic Interrupts

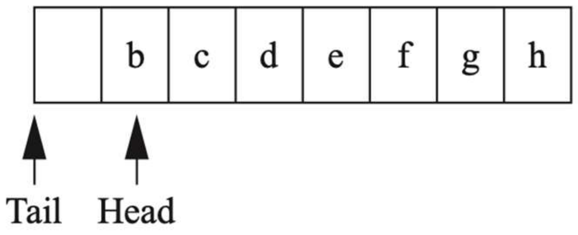
- One character in the buffer is left unused to make possible the distinguishing buffer full of buffer empty



32

Copying Characters from I to O with Basic Interrupts

- Output goes past the end of io_buf



33

Copying Characters from I to O with Basic Interrupts

- Assume two interrupt handling:
  - 1 for the input device
  - 1 for the output device

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
      char achar;
      if (full_buffer()) /* error */
            error = 1;
      else { /* read the character and update pointer */
            achar = peek(IN_DATA); /* read character */
            add_char(achar); /* add to queue */
      }
      poke(IN_STATUS,0); /* set status register back to 0 */
      /* if buffer was empty, start a new output transaction */
      if (nchars() == 1) { /* buffer had been empty until this
      interrupt */
            poke(OUT_DATA,remove_char()); /* send character */
            poke(OUT_STATUS,1); /* turn device on */
      }
}
```

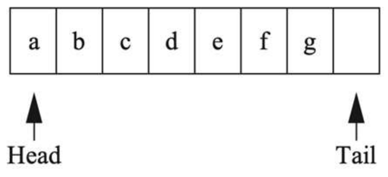
```
void full_buffer() { /* returns TRUE if buffer is full */
      (buf_tail+1) % BUF_SIZE == buf_head ;
}
```

34

Copying Characters from I to O with Basic Interrupts

- Full buffer (e.g.)

```
void full_buffer() { /* returns TRUE if buffer is full */
     (buf_tail+1) % BUF_SIZE == buf_head ;
}
```



Copying Characters from I to O with Basic Interrupts

- Assume two interrupt handling:
  - 1 for the input device
  - 1 for the output device

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
     char achar;
     if (full_buffer()) /* error */
          error = 1;
     else { /* read the character and update pointer */
          achar = peek(IN_DATA); /* read character */
          add_char(achar); /* add to queue */
     }
     poke(IN_STATUS,0); /* set status register back to 0 */
     /* if buffer was empty, start a new output transaction */
     if (nchars() == 1) { /* buffer had been empty until this
     interrupt */
          poke(OUT_DATA,remove_char()); /* send character */
          poke(OUT_STATUS,1); /* turn device on */
     }
}
```

```
void add_char(char achar) { /* add a character to the buffer
head */
                 ?
     io_buf[buf_tail++] = achar;
     /* check pointer */
     if (buf_tail == BUF_SIZE)
          buf_tail = 0;
}
```

Copying Characters from I to O with Basic Interrupts

- Assume two interrupt handling:
  - 1 for the input device
  - 1 for the output device

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
     char achar;
     if (full_buffer()) /* error */
          error = 1;
     else { /* read the character and update pointer */
          achar = peek(IN_DATA); /* read character */
          add_char(achar); /* add to queue */
     }
     poke(IN_STATUS,0); /* set status register back to 0 */
     /* if buffer was empty, start a new output transaction */
     if (nchars() == 1) { /* buffer had been empty until this
     interrupt */
          poke(OUT_DATA,remove_char()); /* send character */
          poke(OUT_STATUS,1); /* turn device on */
     }
}
```

It means a new character is being read and added in the buffer

Copying Characters from I to O with Basic Interrupts

- Assume two interrupt handling:
  - 1 for the input device
  - 1 for the output device

```
int nchars() { /* returns the number of characters in the
buffer */
     if (buf_head >= buf_tail) return buf_head – buf_tail;
     else return BUF_SIZE – buf_tail – buf_head;
}
```

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
     char achar;
     if (full_buffer()) /* error */
          error = 1;
     else { /* read the character and update pointer */
          achar = peek(IN_DATA); /* read character */
          add_char(achar); /* add to queue */
     }
     poke(IN_STATUS,0); /* set status register back to 0 */
     /* if buffer was empty, start a new output transaction */
     if (nchars() == 1) { /* buffer had been empty until this
     interrupt */
          poke(OUT_DATA,remove_char()); /* send character */
          poke(OUT_STATUS,1); /* turn device on */
     }
}
```

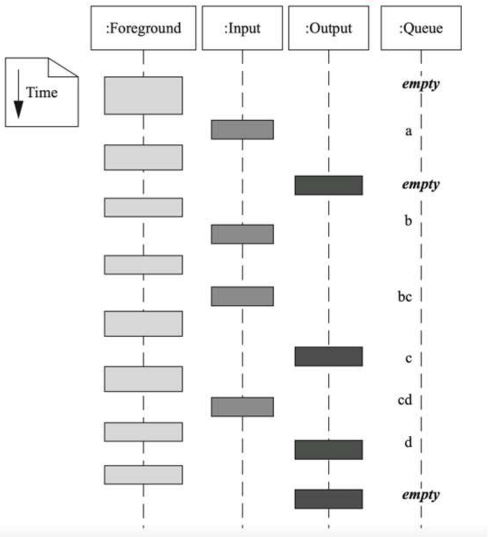
Just when the first character is added in the buffer

Copying Characters from I to O with Basic Interrupts

- Assume two interrupt handling: 1 for the input device + 1 for the output device

```
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
void output_handler() {
     if (!empty_buffer()) { /* start a new character */
          poke(OUT_DATA,remove_char()); /* send character */
          poke(OUT_STATUS,1); /* turn device on */
     }
}
```

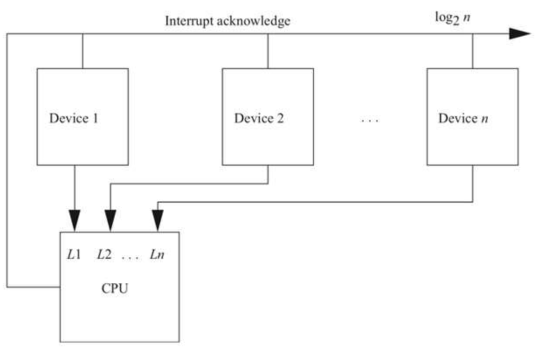
Copying Characters from I to O with Basic Interrupts



Interrupts: Priorities and Vectors

- Most embedded systems have more than one I/O device
- There is a need for a mechanism for allowing multiple devices to interrupt
- Two approaches can be used to handle multiple I/O devices
  - **Interrupt Priorities:**
    - Allow the CPU to recognize some interrupts as more important than others
  - **Interrupt vectors:**
    - Allow the interrupting device to specify its handle

Interrupt Priorities



- CPU provides several different interrupt request signals (L1, L2, …, Ln)
- Typically, the lower-numbered interrupt lines are given higher priority
- Most CPUs use a set of signals that provide the priority number of the winning interrupt in binary form
- A device knows that its interrupt request was accepted by seeing its own priority number on the interrupt acknowledge lines

Interrupt Priorities

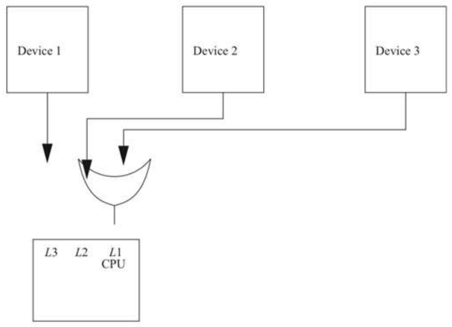
- Masking:
  - Mechanism used to ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is being handled
  - CPU stores in an internal register the priority level of that interrupt, when it acknowledge an interrupt
  - Priority of subsequent interrupts are checked against the priority register
    - New request is acknowledged only if it has higher priority than the currently pending interrupt
  - Priority register is reset when the interrupt handler exits

Interrupt Priorities

- Nonmaskable interrupt (NMI)
  - The highest-priority interrupt
  - The NMI cannot be turned off
  - Usually reserved for interrupts caused by power failures
    - A simple circuit can be used to detect a dangerously low power supply, and the NMI interrupt handler can be used to save critical state in nonvolatile memory, turn off I/O devices to eliminate spurious device operation during power- down, and so on

Interrupt Priorities

- Most CPUs provide a relatively small number of interrupt priority levels, such as eight
- When several devices naturally assume the same priority:

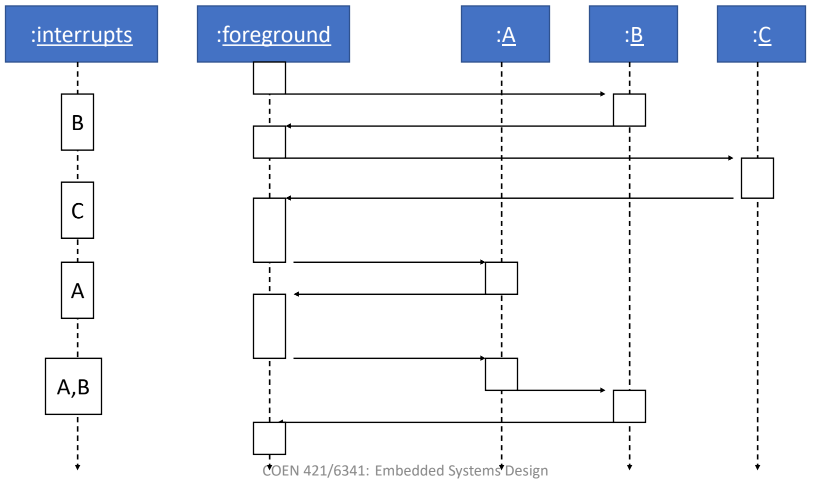


- The CPU will call the interrupt handler associated with this priority
- That handler does not know which of the devices actually requested the interrupt
- The handler uses software polling to check the status of each device:
  - e.g., it would read the status registers of 1, 2, and 3 to see which of them is ready and requesting service

45

Interrupt Priorities

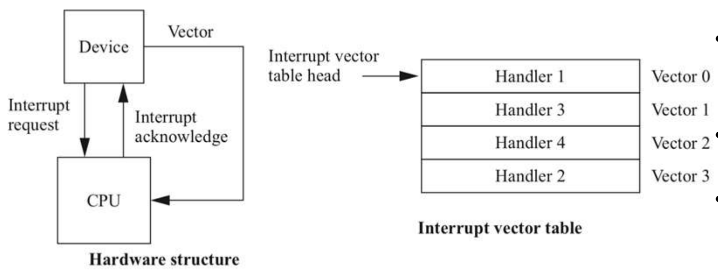
A has priority 1 (highest priority), B has priority 2, and C has priority 3



46

Interrupt Vectors

- Provides the ability to define the interrupt handler that should service a request from a device



- Interrupt vector lines run from the devices to the CPU
- After a device’s request is acknowledged, it sends its interrupt vector over those lines to the CPU
- The CPU then uses the vector number as an index in a table stored in memory
- The location referenced in the interrupt vector table by the vector number gives the address of the handler

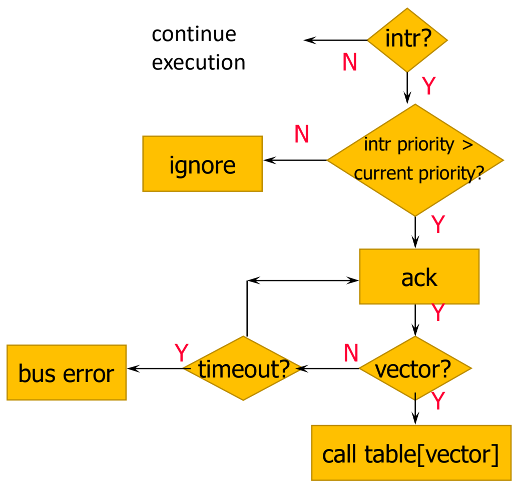
Interrupts

- Most modern CPUs implement both prioritized and vectored interrupts
- Priorities determine which device is serviced first, and vectors determine what routine is used to service the interrupt

The Interrupt Handling Process

- Device requests an interrupt
- The CPU checks for pending interrupts at the beginning of an instruction
  - It will acknowledge the highest-priority pending interrupt (which has a higher priority than that given in the interrupt priority register)
- The device receives the acknowledgment and sends the CPU its interrupt vector
- The CPU looks up the device handler address in the interrupt vector table using the vector as an index
- The device driver (interrupt handler routine) may save additional CPU state
- It then performs the required operations on the device
- It then restores any saved state and executes the interrupt return instruction
- The interrupt return instruction restores the PC and other automatically saved states to return execution to the code that was interrupted

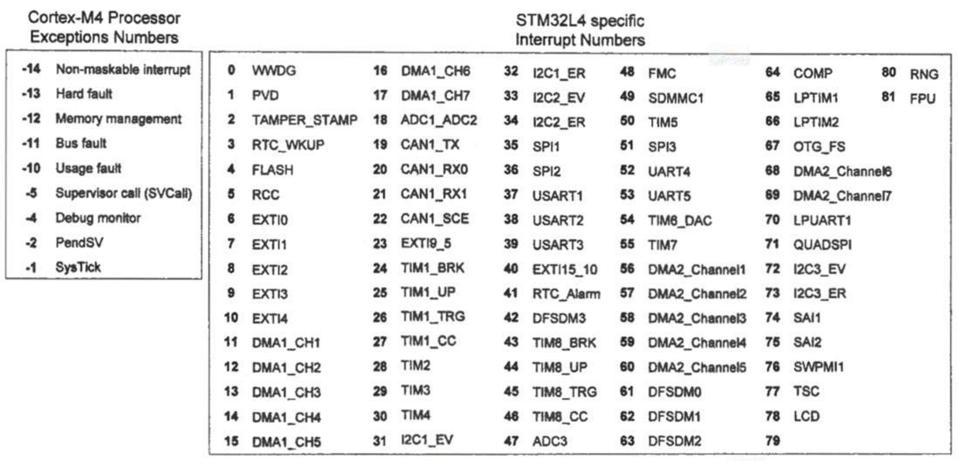
The Interrupt Handling Process



Cortex-M Interrupts

- 256 types of interrupts
- Each interrupt is identified by a unique number [-15, 240] (except reset interrupt)
- Interrupt numbers are defined by ARM and chip manufacturers collectively
- Interrupt numbers are divided into two groups
  - System interrupts (or system exceptions) [-15, -1]
    - Those exceptions (interrupts) come from the processor core
    - Defined by ARM
  - Peripheral interrupts [0, 240]
    - Defined by chip manufacturers
    - Total number of peripheral interrupts supported varies among chips

Cortex-M Interrupts



Cortex-M4 Processor Exceptions Numbers

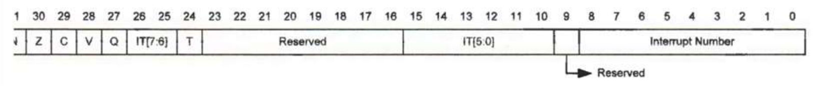
-14	Non-maskable interrupt
-13	Hard fault
-12	Memory management
-11	Bus fault
-10	Usage fault
-5	Supervisor call (SVCall)
-4	Debug monitor
-2	PendSV
-1	SysTick

STM32L4 specific Interrupt Numbers

0	WWDG	16	DMA1_CH6	32	I2C1_ER	48	FMC	64	COMP	80	RNG
1	PVD	17	DMA1_CH7	33	I2C2_EV	49	SDMMC1	65	LPTIM1	81	FPU
2	TAMPER_STAMP	18	ADC1_ADC2	34	I2C2_ER	50	TIM5	66	LPTIM2		
3	RTC_WKUP	19	CAN1_TX	35	SPI1	51	SPI3	67	OTG_FS		
4	FLASH	20	CAN1_RX0	36	SPI2	52	UART4	68	DMA2_Channel6		
5	RCC	21	CAN1_RX1	37	USART1	53	UART5	69	DMA2_Channel7		
6	EXTI0	22	CAN1_SCE	38	USART2	54	TIM6_DAC	70	LPUART1		
7	EXTI1	23	EXTI9_5	39	USART3	55	TIM7	71	QUADSPI		
8	EXTI2	24	TIM1_BRK	40	EXTI15_10	56	DMA2_Channel1	72	I2C3_EV		
9	EXTI3	25	TIM1_UP	41	RTC_Alarm	57	DMA2_Channel2	73	I2C3_ER		
10	EXTI4	26	TIM1_TRG	42	DFSDM3	58	DMA2_Channel3	74	SAI1		
11	DMA1_CH1	27	TIM1_CC	43	TIM8_BRK	59	DMA2_Channel4	75	SAI2		
12	DMA1_CH2	28	TIM2	44	TIM8_UP	60	DMA2_Channel5	76	SWPMI1		
13	DMA1_CH3	29	TIM3	45	TIM8_TRG	61	DFSDM0	77	TSC		
14	DMA1_CH4	30	TIM4	46	TIM8_CC	62	DFSDM1	78	LCD		
15	DMA1_CH5	31	I2C1_EV	47	ADC3	63	DFSDM2	79			

Cortex-M Interrupts

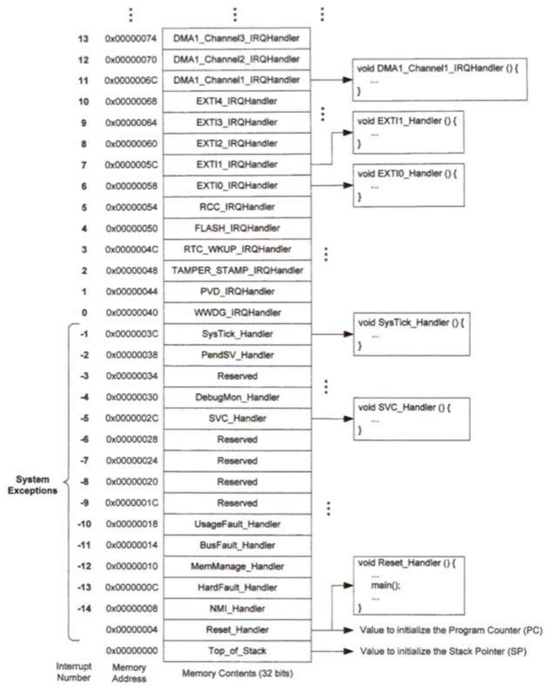
- The numbering scheme allows software to distinguish system exceptions and peripheral interrupts
- Interrupt number is stored in the PSR when an interrupt is processed



- ARM Cortex-M does not store interrupt numbers in two's complement
- How can we represent the negative numbers?
  - *Interrupt number in PSR adds a positive offset of 15*

Cortex-M Interrupt Vector Table

- Interrupt Vector Table:
  - Special array used by Cortex-M to stores the starting memory address of every ISR
- Address of ISR = IVT[i+15]
- How can we find the memory address of an interrupt handler?



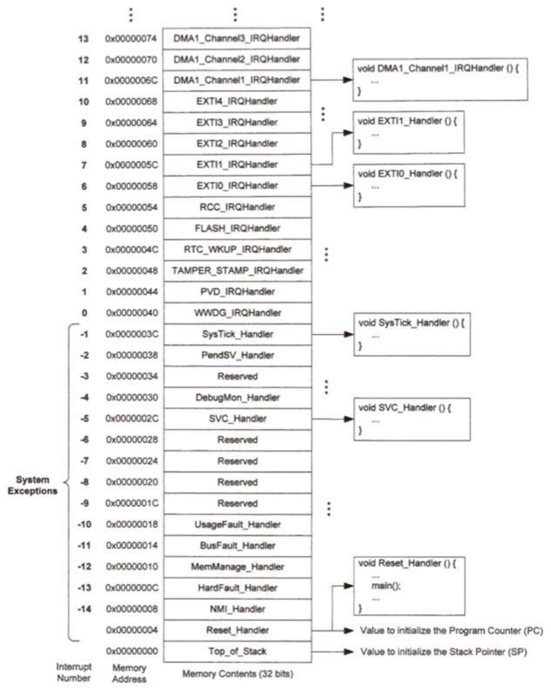
Interrupt Number	Memory Address	Memory Contents (32 bits)
⋮	⋮	⋮
13	0x00000074	DMA1_Channel3_IRQHandler
12	0x00000070	DMA1_Channel2_IRQHandler
11	0x0000006C	DMA1_Channel1_IRQHandler
10	0x00000068	EXTI4_IRQHandler
9	0x00000064	EXTI3_IRQHandler
8	0x00000060	EXTI2_IRQHandler
7	0x0000005C	EXTI1_IRQHandler
6	0x00000058	EXTI0_IRQHandler
5	0x00000054	RCC_IRQHandler
4	0x00000050	FLASH_IRQHandler
3	0x0000004C	RTC_WKUP_IRQHandler
2	0x00000048	TAMPER_STAMP_IRQHandler
1	0x00000044	PVD_IRQHandler
0	0x00000040	WWDG_IRQHandler
-1	0x0000003C	SysTick_Handler
-2	0x00000038	PendSV_Handler
-3	0x00000034	Reserved
-4	0x00000030	DebugMon_Handler
-5	0x0000002C	SVC_Handler
-6	0x00000028	Reserved
-7	0x00000024	Reserved
-8	0x00000020	Reserved
-9	0x0000001C	Reserved
-10	0x00000018	UsageFault_Handler
-11	0x00000014	BusFault_Handler
-12	0x00000010	MemManage_Handler
-13	0x0000000C	HardFault_Handler
-14	0x00000008	NMI_Handler
	0x00000004	Reset_Handler
	0x00000000	Top_of_Stack

System Exceptions
void DMA1_Channel1_IRQHandler () { ... }
void EXTI1_Handler () { ... }
void EXTI0_Handler () { ... }
void SysTick_Handler () { ... }
void SVC_Handler () { ... }
void Reset_Handler () { ... main(); ... }
Value to initialize the Program Counter (PC)
Value to initialize the Stack Pointer (SP)

Cortex-M Interrupt Vector Table

- How can we find the memory address of an interrupt handler?
  - Example: SysTick_Handler?
    - Interrupt vector table is stored at the memory address:
      - 0x00000004
    - SysTick interrupt is -1

    - A = 0x00000004 + 4 X (-1+15)
    - A = 0x0000003C



Interrupt Number	Memory Address	Memory Contents (32 bits)
⋮	⋮	⋮
13	0x00000074	DMA1_Channel3_IRQHandler
12	0x00000070	DMA1_Channel2_IRQHandler
11	0x0000006C	DMA1_Channel1_IRQHandler
10	0x00000068	EXTI4_IRQHandler
9	0x00000064	EXTI3_IRQHandler
8	0x00000060	EXTI2_IRQHandler
7	0x0000005C	EXTI1_IRQHandler
6	0x00000058	EXTI0_IRQHandler
5	0x00000054	RCC_IRQHandler
4	0x00000050	FLASH_IRQHandler
3	0x0000004C	RTC_WKUP_IRQHandler
2	0x00000048	TAMPER_STAMP_IRQHandler
1	0x00000044	PVD_IRQHandler
0	0x00000040	WWDG_IRQHandler
-1	0x0000003C	SysTick_Handler
-2	0x00000038	PendSV_Handler
-3	0x00000034	Reserved
-4	0x00000030	DebugMon_Handler
-5	0x0000002C	SVC_Handler
-6	0x00000028	Reserved
-7	0x00000024	Reserved
-8	0x00000020	Reserved
-9	0x0000001C	Reserved
-10	0x00000018	UsageFault_Handler
-11	0x00000014	BusFault_Handler
-12	0x00000010	MemManage_Handler
-13	0x0000000C	HardFault_Handler
-14	0x00000008	NMI_Handler
	0x00000004	Reset_Handler
	0x00000000	Top_of_Stack

```
void DMA1_Channel1_IRQHandler () {
    ...
}
void EXTI1_Handler () {
    ...
}
void EXTI0_Handler () {
    ...
}
void SysTick_Handler () {
    ...
}
void SVC_Handler () {
    ...
}
void Reset_Handler () {
    ...
    main();
    ...
}
```

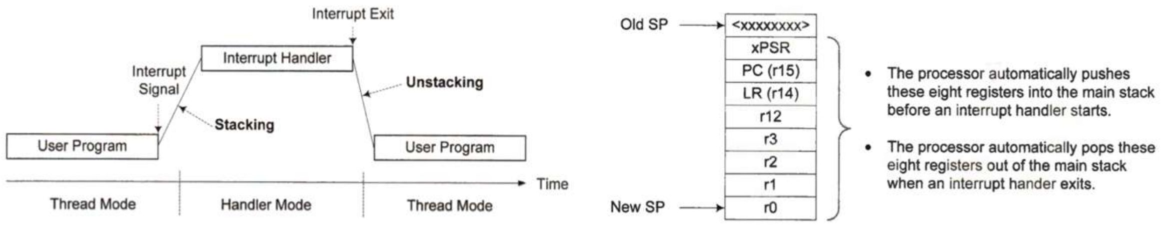
System Exceptions
Value to initialize the Program Counter (PC)
Value to initialize the Stack Pointer (SP)

Cortex-M Interrupts

- Enable specific interrupts and set their priority levels
- All interrupts are served based on their priority levels
- Priority levels -> given by interrupt number
  - A higher value of the interrupt priority number represents a lower priority
- A current interrupt being served is stopped if a higher priority interrupt occurs
- Processor resumes the low-priority interrupt when the handler of the higher priority interrupt completes

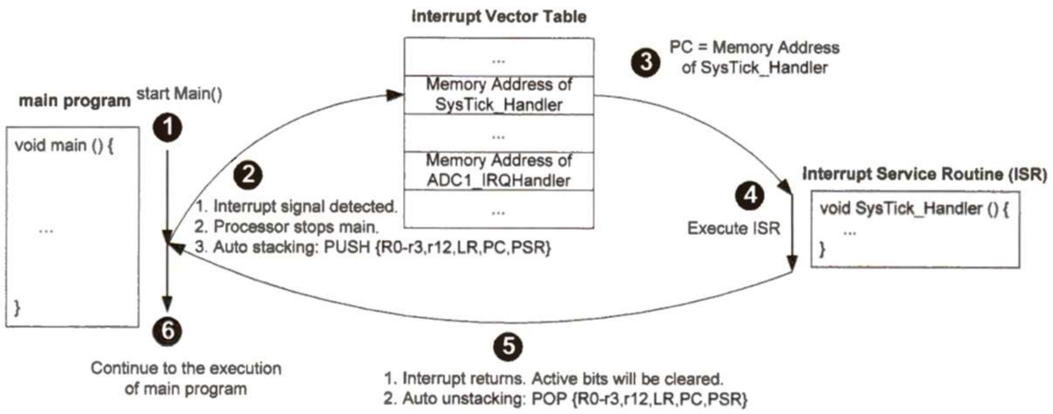
Cortex-M Interrupts

- Interrupt stacking:
  - Before executing the interrupt handler, the current running environment must be preserved
  - 8 registers are preserved (r0, r1, r2, r3, r12, LR, PSR, and PC)
- Interrupt Unstacking
  - Recovers the environment that existed at the time instant immediately before the interrupt handler started to run
  - Pops the values of the r0, r1, r2, r3, r12, LR, PSR, and PC off the stack



- The processor automatically pushes these eight registers into the main stack before an interrupt handler starts.
- The processor automatically pops these eight registers out of the main stack when an interrupt hander exits.

Cortex-M Interrupts


Interrupt Vector Table
...
Memory Address of SysTick_Handler
...
Memory Address of ADC1_IRQHandler
...
main program
start Main()
void main () {
...
}
1
2
1. Interrupt signal detected.
2. Processor stops main.
3. Auto stacking: PUSH {R0-r3,r12,LR,PC,PSR}
3
PC = Memory Address of SysTick_Handler
4
Execute ISR
Interrupt Service Routine (ISR)
void SysTick_Handler () {
...
}
5
1. Interrupt returns. Active bits will be cleared.
2. Auto unstacking: POP {R0-r3,r12,LR,PC,PSR}
6
Continue to the execution of main program

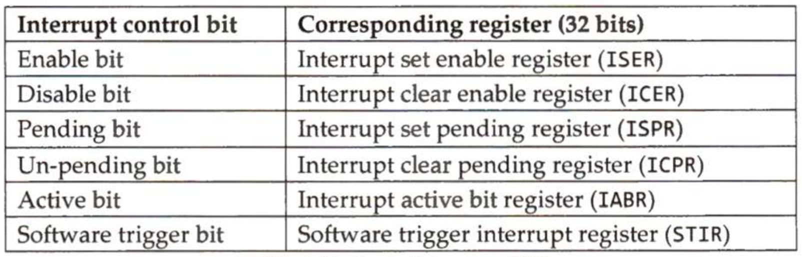
Cortex-M Interrupts

- The interrupt service routine exits by running “BX LR”
- LR in an interrupt service routine indicates whether the processor uses the main stack or the process stack in the push and pop operations
  - LR shall be fixed to a special value to indicate whether the processor should unstack data out of the main stack (MSP) or process stack (PSP)

59

Nested Vectored Interrupt Controller

- Three key functions:
  - Enable and disable interrupts
  - Configure the preemption priority and sub-priority of a specific interrupt
  - Set and clear the handling bit of a specific interrupt
- Each interrupt has six control bits

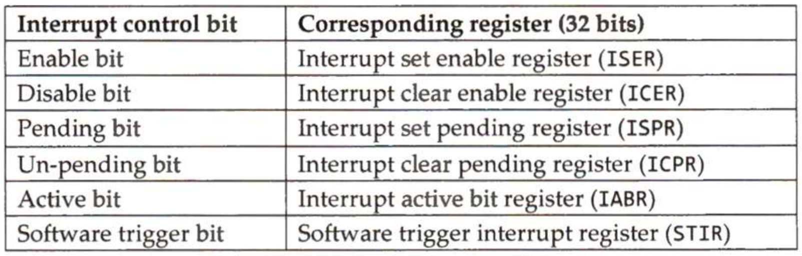


Interrupt control bit	Corresponding register (32 bits)
Enable bit	Interrupt set enable register (ISER)
Disable bit	Interrupt clear enable register (ICER)
Pending bit	Interrupt set pending register (ISPR)
Un-pending bit	Interrupt clear pending register (ICPR)
Active bit	Interrupt active bit register (IABR)
Software trigger bit	Software trigger interrupt register (STIR)

60

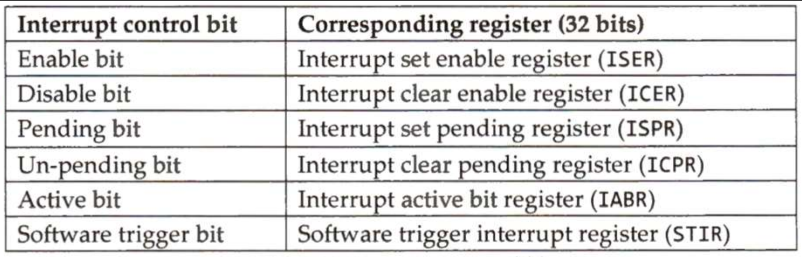
Nested Vectored Interrupt Controller

- Three key functions:
  - Enable and disable interrupts
  - Configure the preemption priority and sub-priority of a specific interrupt
  - Set and clear the handling bit of a specific interrupt
- Each interrupt has six control bits and 8 registers for each control bit



Interrupt control bit	Corresponding register (32 bits)
Enable bit	Interrupt set enable register (ISER)
Disable bit	Interrupt clear enable register (ICER)
Pending bit	Interrupt set pending register (ISPR)
Un-pending bit	Interrupt clear pending register (ICPR)
Active bit	Interrupt active bit register (IABR)
Software trigger bit	Software trigger interrupt register (STIR)

Nested Vect



Interrupt control bit	Corresponding register (32 bits)
Enable bit	Interrupt set enable register (ISER)
Disable bit	Interrupt clear enable register (ICER)
Pending bit	Interrupt set pending register (ISPR)
Un-pending bit	Interrupt clear pending register (ICPR)
Active bit	Interrupt active bit register (IABR)
Software trigger bit	Software trigger interrupt register (STIR)

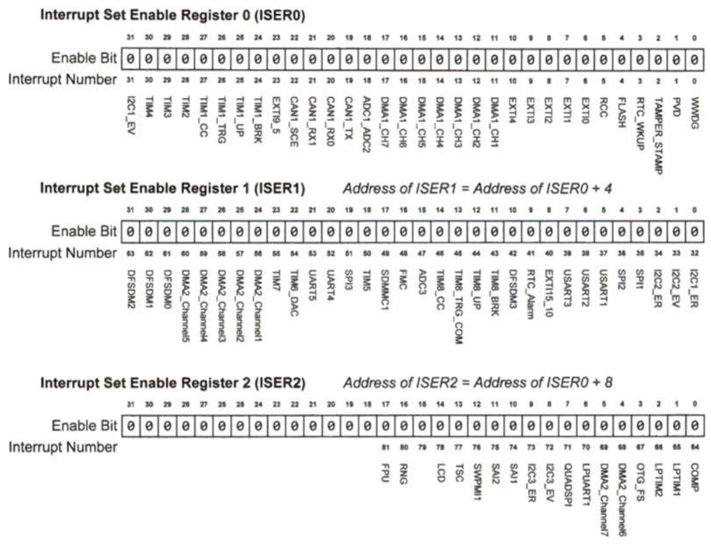
- Interrupt control bit
  - Enable interrupt
    - Write an enable bit to 1 in ISER register
  - Disable interrupt
    - Write a disable bit to 1 in ICER register
  - Pending interrupt
    - If the microcontroller cannot process an interrupt immediately the corresponding pending bit is set in ISPR
    - If the pending interrupt is handled, the corresponding interrupt is removed from the pending list by seting the un-pending corresponding bit (ICPR)
  - Active interrupt
    - When an interrupt is disabled but it is pending bit has already been set, this interrupt instance remains active and it is serviced before it is disabled

Nested Vectored Interrupt Controller

- Each interrupt has six control bits and 8 registers for each control bit
  - e.g., Interrupt Set-Enable Registers
    - Bit 1: enable the corresponding interrupt
    - Bit 0: does not turn off the corresponding interrupt

    - To enable the interrupt, it is needed to know what register and bit to set
      - Register:
        - i = floor (interrupt number/32)
        - j = interrupt number % 32

Nested Vectored Interrupt Controller



Interrupt Set Enable Register 0 (ISER0)

Bit	Enable Bit	Interrupt Number	Interrupt
31	0	31	I2C1_EV
30	0	30	TIM4
29	0	29	TIM3
28	0	28	TIM2
27	0	27	TIM1_CC
26	0	26	TIM1_TRG
25	0	25	TIM1_UP
24	0	24	TIM1_BRK
23	0	23	EXTI9_5
22	0	22	CAN1_SCE
21	0	21	CAN1_RX1
20	0	20	CAN1_RX0
19	0	19	CAN1_TX
18	0	18	ADC1_ADC2
17	0	17	DMA1_CH7
16	0	16	DMA1_CH6
15	0	15	DMA1_CH5
14	0	14	DMA1_CH4
13	0	13	DMA1_CH3
12	0	12	DMA1_CH2
11	0	11	DMA1_CH1
10	0	10	EXTI4
9	0	9	EXTI3
8	0	8	EXTI2
7	0	7	EXTI1
6	0	6	EXTI0
5	0	5	RCC
4	0	4	FLASH
3	0	3	RTC_WKUP
2	0	2	TAMPER_STAMP
1	0	1	PVD
0	0	0	WWDG

Interrupt Set Enable Register 1 (ISER1) *Address of ISER1 = Address of ISER0 + 4*

Bit	Enable Bit	Interrupt Number	Interrupt
31	0	63	DFSDM2
30	0	62	DFSDM1
29	0	61	DFSDM0
28	0	60	DMA2_Channel5
27	0	59	DMA2_Channel4
26	0	58	DMA2_Channel3
25	0	57	DMA2_Channel2
24	0	56	DMA2_Channel1
23	0	55	TIM7
22	0	54	TIM6_DAC
21	0	53	UART5
20	0	52	UART4
19	0	51	SPI3
18	0	50	TIM5
17	0	49	SDMMC1
16	0	48	FMC
15	0	47	ADC3
14	0	46	TIM8_CC
13	0	45	TIM8_TRG_COM
12	0	44	TIM8_UP
11	0	43	TIM8_BRK
10	0	42	DFSDM3
9	0	41	RTC_Alarm
8	0	40	EXTI15_10
7	0	39	USART3
6	0	38	USART2
5	0	37	USART1
4	0	36	SPI2
3	0	35	SPI1
2	0	34	I2C2_ER
1	0	33	I2C2_EV
0	0	32	I2C1_ER

Interrupt Set Enable Register 2 (ISER2) *Address of ISER2 = Address of ISER0 + 8*

Bit	Enable Bit	Interrupt Number	Interrupt
31	0		
30	0		
29	0		
28	0		
27	0		
26	0		
25	0		
24	0		
23	0		
22	0		
21	0		
20	0		
19	0		
18	0		
17	0	81	FPU
16	0	80	RNG
15	0	79	
14	0	78	LCD
13	0	77	TSC
12	0	76	SWPMI1
11	0	75	SAI2
10	0	74	SAI1
9	0	73	I2C3_ER
8	0	72	I2C3_EV
7	0	71	QUADSPI
6	0	70	LPUART1
5	0	69	DMA2_Channel7
4	0	68	DMA2_Channel6
3	0	67	OTG_FS
2	0	66	LPTIM2
1	0	65	LPTIM1
0	0	64	COMP

Nested Vectored Interrupt Controller

- Setting the proper bit in register ISER to enable interrupt IRQn

```
WordOffset = IRQn >> 5;                          // Word Offset = IRQn/32
BitOffset = IRQn & 0x1F;                         // Bit Offset = IRQn mod 32
NVIC->ISER[WordOffset]  =  1 << BitOffset;       // Enable interrupt
```

- E.g., IRQn = 50 (Timer5 interrupt)

```
NVIC->ISER[1] = 1 << 18;                         // Enable Timer 5 interrupt
```

- IRQn = j + 32 x i
  - i = floor (interrupt number/32) => floor(50/32) = 1
  - j = interrupt number % 32 => 50 % 32 = 18

Nested Vectored Interrupt Controller

- Assembly implementation of enabling or disabling a peripheral interrupt

```
; Input arguments:
;    r0: interrupt number of a peripheral interrupt
;    r1: 1 = Enable, 0 = Disable

Peripheral_Interrupt_Enable    PROC
   PUSH  {r4, lr}
   AND   r2, r0, #0x1F       ; Bit offset in a word
   MOV   r3, #1
   LSL   r3, r3, r2          ; r3 = 1 << (IRQn & 0x1F)
   LDR   r4, =NVIC_BASE

   CMP   r1, #0              ; Check whether enable or disable
   LDRNE r1, =NVIC_ISER0     ; Enable register base address
   LDREQ r1, =NVIC_ICER0     ; Disable register base address

   ADD   r1, r4, r1          ; r1 = addr. of NVIC->ISER0 or NVIC->ICER0
   LSR   r2, r0, #5          ; Memory offset (in words): IRQn >> 5
   LSL   r2, r2, #2          ; Calculate byte offset
   STR   r3, [r1, r2]        ; Enable/Disable interrupt
   POP   {r4, pc}
   ENDP
```

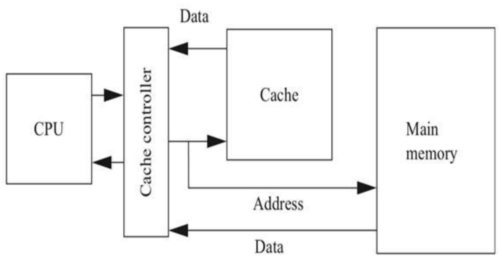
Nested Vectored Interrupt Controller

- Setting the proper bit in register **ICER** to enable interrupt IRQn

```
WordOffset = IRQn >> 5;                     // WordOffset = IRQn/32
BitOffset = IRQn & 0x1F;                    // BitOffset = IRQn mod 32
NVIC->ICER[WordOffset]  =  1 << BitOffset;  // Disable interrupt
```

Memory System Mechanism

- Caches
  - Small, fast memory that holds copies of some of the contents of main memory
  - Used to speed up reads and writes in memory systems
- Cache Controller:
  - Mediates between the CPU and the memory system comprised of the cache and main memory
  - Sends a memory request to the cache and main memor
- Cache Hit:
  - If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request
- Cache Miss:
  - If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU



Cache

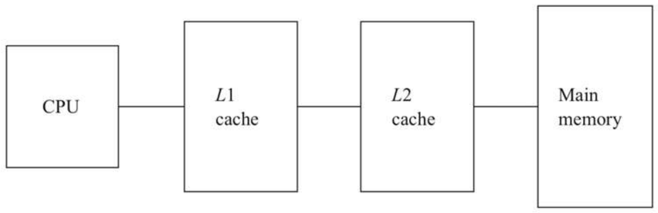
- We can classify cache misses into several types depending on the situation that generated them
  - Compulsory miss (also known as a cold miss)
    - Occurs the first time a location is used
  - Capacity miss
    - Caused by a too-large working set
  - Conflict miss
    - Happens when two locations map to the same location in the cache.

Memory System Performance

- t_{cache} : access time of the cache (a few nanoseconds)
- t_{main}: main memory access time (~50 to 75ns)
- $t_{cache} << t_{main}$
- h: hit ratio
- $t_{av} = ht_{cache} + (1-h)t_{main}$

Cache

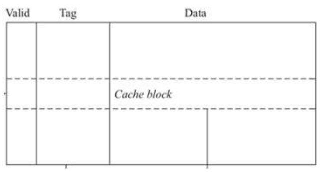
- Modern CPUs may use multiple levels of cache
- L1: First-level cache is closest to the CPU
- L2: Second-level cache feeds the first-level cache
- … and so on



$t_{av} = h_1 t_{L1} + (h_2 - h_1) t_{L2} + (1 - h_2) t_{main}$

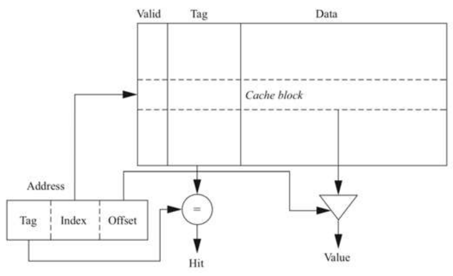
Direct-Mapped Cache

- The cache consists of blocks
- Each line includes:
  - A **tag** to show which memory location is represented by this block
  - A **data** field that holds the content of that memory location
  - A **valid tag** to show whether the contents of this cache block are valid



Direct-Mapped Cache

- Address: divided into 3 sections
  - **Index**: used to select which cache block to check
  - **Tag** is compared against the tag value in the block selected by the index
  - **Offset** is used to select the required value from the data field when the length of the data field is longer than the minimum addressable unit

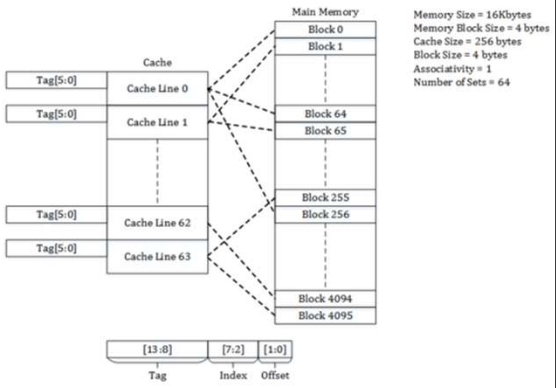


Direct-Mapped Cache

- Write operation: We have to update the main memory as well
  - Write-through
    - Every write changes both the cache and the corresponding main memory location
  - Write-back
    - Write only when we remove a location from the cache

Direct-Mapped Cache

- Example: Assume a main memory of 16Kb and cache of 256bytes organized in 4byte blocks
- Memory address: 14-bits
- Cache lines: 256/4 = 64:
  - Thus, 6 bits to address the lines
- 4 blocks in each cache line:
  - Thus, 2 bits to address the block
- 6 bits as tag

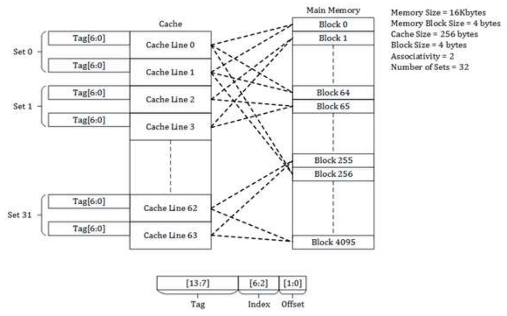


Set-Associative Cache

- Characterized by the number of banks or ways
- A set is formed by all the blocks (one for each bank) that share the same index
- Each set is implemented with a direct-mapped cache
- A cache request is broadcast to all banks (or ways) simultaneously
- If any of the ways has the location, the cache reports a hit
- The set-associative cache generally provides higher hit rates than the direct-mapped cache because conflicts between a small set of locations can be resolved within the cache

Set-Associative Cache

- Example: Assume a main memory of 16Kb and a 2-way set-associative of 256 bytes organized with a block size of 4
- Memory address: 14-bits long
- Cache:
  - 2-way set associative: means 2 cache lines in each set
  - Number of banks: 256/(4*2) = 32
- Addressing:
  - 2 bits to address the block
  - 5 bits to address the set
  - 7 bits for tag



Example

- Let's assume a memory where the address is given by 3 bits
- Data requests: 001, 010, 011, 100, 101, and 111

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

After 001 access:

Block	Tag	Data
00	—	—
01	0	1111
10	—	—
11	—	—

After 010 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	—	

After 011 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	0	0110

Example

- Let's assume a memory where the address is given by 3 bits
- Data requests: 001, 010, 011, 100, 101, and 111

After 100 access (notice that the tag bit for this entry is 1):

Block	Tag	Data
00	1	1000
01	0	1111
10	0	0000
11	0	0110

After 101 access (overwrites the 01 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	0	0110

After 111 access (overwrites the 11 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	1	0100

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

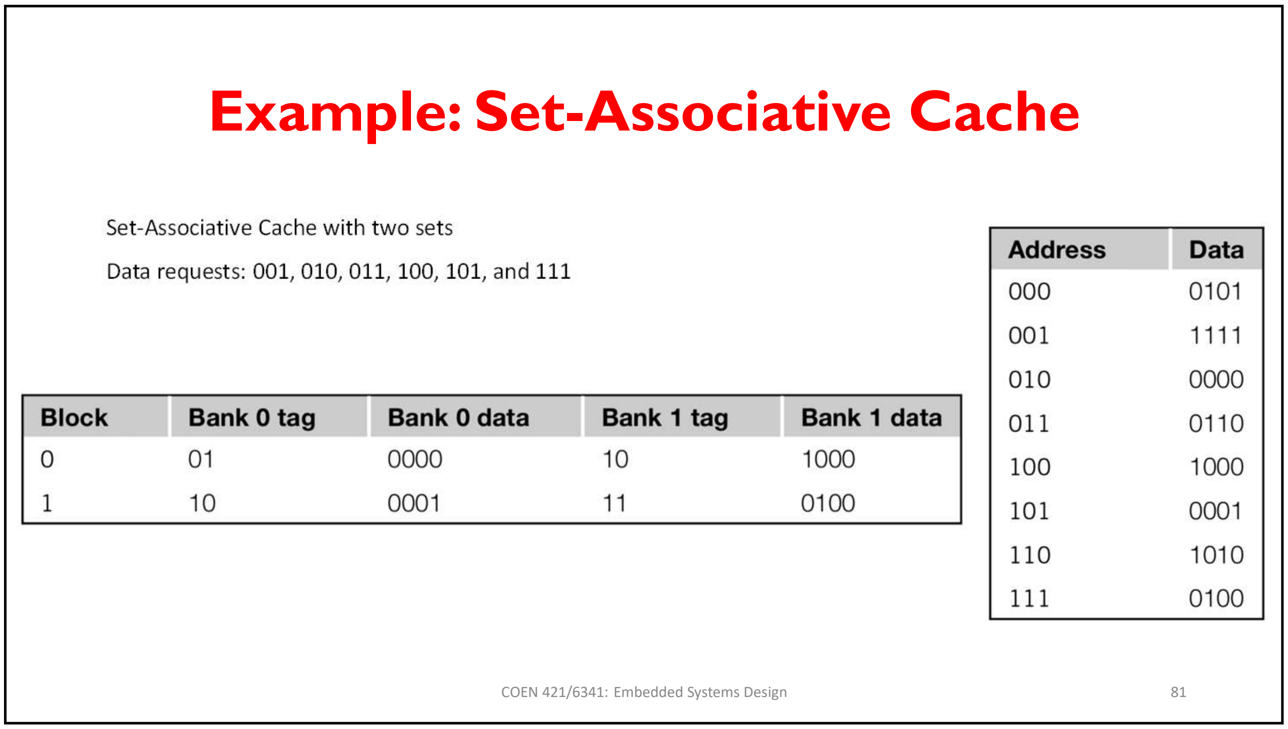
Example: Set-Associative Cache

To start, let's consider that each bank has the same size of the example of the original direct-mapped cache

Data requests: 001, 010, 011, 100, 101, and 111

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
00	1	1000	—	—
01	0	1111	1	0001
10	0	0000	—	
11	0	0110	1	0100

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100



Example: Set-Associative Cache

Set-Associative Cache with two sets

Data requests: 001, 010, 011, 100, 101, and 111

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
0	01	0000	10	1000
1	10	0001	11	0100

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

81