MEMORY & I/O SYSTEMS

Chapter 6

Digital Design and Computer Architecture, 2nd Edition

David Money Harris and Sarah L. Harris

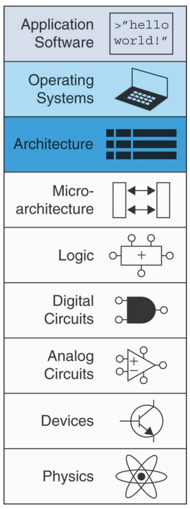
Adapted by
Pedro.guerra@upm.es

ELSEVIER

MEMORY & I/O SYSTEMS

Chapter 6 : Topics

- **Introduction**
- **Assembly Language**
- **Machine Language**
- **Programming**
- **Addressing Modes**
- **Lights, Camera, Action: Compiling, Assembling, & Loading**
- **Odds and Ends**



ELSEVIER

Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:**
- how to implement an architecture in hardware (covered in Chapter 7)

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **MIPS** architecture: RISC
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

Once you've learned one architecture, it's easy to learn others

MEMORY & I/O SYSTEMS

Ejercicio

Código máquina de las siguientes instrucciones

```
ori   $t0, $0, 0x00FF
la    $s0, PORTA
beq   $t0, $a0, 0x04
j     0x4440
```


ELSEVIER

MEMORY & I/O SYSTEMS

Solución

ori $t0, $0, 0x00FF

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

op: 13 (001101)
rs: $0 (00000)
rt: $t0 (01000)
imm: 0x00FF (0000000011111111)

001101	00000	01000	0000000011111111

Bin:	0011	0100	0000	1000	0000	0000	1111	1111
Hex:	3	4	0	8	0	0	F	F

ELSEVIER

MEMORY & I/O SYSTEMS

Solución (la: lui)

la	$s0, $0, PORTA	PORTA=0xBF886020
lui	$s0, PORTA_H	PORTA_H=0xBF88
ori	$s0, $s0, PORTA_L	PORTA_L=0x6020

op: 15 (001111)
rs: $0 (00000)
rt: $s0 (10000)
imm: 0xBF88 (1011111110001000)

001111	00000	10000	1011111110001000

Bin:	0011	1100	0001	0000	1011	1111	1000	1000
Hex:	3	C	1	0	B	F	8	8

ELSEVIER

MEMORY & I/O SYSTEMS

Solución (la: ori)

la	$s0, $0, PORTA	PORTA=0xBF886020
lui	$s0, PORTA_H	PORTA_H=0xBF88
ori	$s0, $s0, PORTA_L	PORTA_L=0x6020

op: 13 (001101)
rs: $s0 (10000)
rt: $s0 (10000)
imm: 0x6020 (0110000000100000)

001101	10000	10000	0110000000100000

Bin:	0011	0110	0001	0000	0110	0000	0010	0000
Hex:	3	6	1	0	6	0	2	0

ELSEVIER

MEMORY & I/O SYSTEMS

Solución

beq $t0, $a0, 0x04

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

op: 4 (000100)
rs: $t0 (01000)
rt: $a0 (00010)
imm: 0x0004 (0000000000000100)

000100	01000	00010	0000000000000100

Bin:	0001	0001	0000	0010	0000	0000	0000	0100
Hex:	1	1	0	2	0	0	0	4

ELSEVIER

MEMORY & I/O SYSTEMS

Solución

j 0x44440

J-Type

op	addr
6 bits	26 bits

op: 2 (000010)
addr: 0x1110 (00000000000001000100010000)
addr se calcula: despreciando los 2 bits de la derecha
despreciando los 4 bits de la izquierda

000010	00000000000001000100010000

Bin:	0000	1000	0000	0000	0001	0001	0001	0000
Hex:	0	8	0	0	1	1	1	0

ELSEVIER

- **El hecho de que la estructura simple de un procesador RISC conduzca a una notable reducción de la superficie del circuito integrado, se aprovecha con frecuencia para ubicar en el mismo, funciones adicionales:**

Architecture Design Principles

RISC underlying design principles:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Architecture Design Principles

P1: Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- easier to encode and handle in hardware

P2: Make the common case fast

- MIPS includes only simple, commonly used instructions
- HWto decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions

P3: Smaller is Faster

- MIPS includes only a small number of registers

Operands

- Operand location: physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

- MIPS has 32 32-bit registers
  - Registers are faster than memory
  - MIPS called "32-bit architecture" because it operates on 32-bit data

MEMORY & I/O SYSTEMS

MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

ELSEVIER

MEMORY & I/O SYSTEMS

Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw $s0, 5($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

Any register may be used as base address

ELSEVIER

MEMORY & I/O SYSTEMS

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `$s3`
  - address = (`$0` + 1) = 1
  - `$s3` = 0xF2F1AC07 after load

Assembly code

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

 Chapter 8 <17> ELSEVIER

MEMORY & I/O SYSTEMS

Writing Word-Addressable Memory

- **Example:** Write (store) the value in `$t4` into memory address 7
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

Assembly code

```
sw $t4, 0x7($0)  # write the value in $t4
                 # to memory word 7
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

 Chapter 8 <18> ELSEVIER

MEMORY & I/O SYSTEMS

Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address increments by 4

Word Address	Data				
⋮		⋮			⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

width = 4 bytes

ELSEVIER

MEMORY & I/O SYSTEMS

Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- **MIPS is byte-addressed, not word-addressed**

ELSEVIER

MEMORY & I/O SYSTEMS

Big-Endian & Little-Endian Memory

- How to number bytes within a word?
- **Little-endian:** byte numbers start at the little (least significant) end
- **Big-endian:** byte numbers start at the big (most significant) end
- **Word address** is the **same** for big- or little-endian

Big-Endian Byte Address				Word Address	Little-Endian Byte Address			
⋮				⋮	⋮			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

MEMORY & I/O SYSTEMS

Big-Endian & Little-Endian Example

- Suppose `$t0` initially contains 0x23456789
- After following code runs on big-endian system, what value is `$s0`?
- In a little-endian system?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

Ejercicio: práctica de endianness

- Utilizando estímulos externos podemos modificar los bits RA0 a RA7 (LSB). Cargamos el byte 0 de la palabra y lo copiamos en el puerto B.
  - Si vemos el cambio: Little-endian
  - Si no vemos el cambio: Big-endian

```
la    $t0, TRISB
sw    $0,   0($t0)

la    $t0, PORTA
lb    $t1,  0($t0)

la    $t0, LATB
sw    $t1, 0($t0)
```


Ejercicio

```
la $t0, TMR1
sw $0, 0($t0)

la      $t0, PR1
ori     $t1, $0, 258
sw      $t1, 0($t0)
```

```
TMR1 = 0xBF800610
.global TMR1
PR1 = 0xBF800620
.global PR1
```

¿Qué valor tienen los siguientes bytes de memoria en un entorno Big-endian?
Memoria: 0xBF800620, 0xBF800621, 0xBF800622, 0xBF800623
Memoria: 0xBF800610, 0xBF800611, 0xBF800612, 0xBF800613

¿Qué valor hay los registros $t0 y $t1 tras la ejecución?

MEMORY & I/O SYSTEMS

Solución

	Big-endian	Little-endian
0xBF800610	0	0
0xBF800611	0	0
0xBF800612	0	0
0xBF800613	0	0
0xBF800620	0	2
0xBF800621	0	1
0xBF800622	1	0
0xBF800623	2	0
$t0	0xBF800620	0xBF800620
$t1	258	258

MEMORY & I/O SYSTEMS

Solución

```
while (1) {
   IFS0 = IFS0 & (~T1IF);
   while (! IFS0 & T1IF);
   LATAINV = 0x01;
}
```

```
endless:
    la    $t0, IFS0CLR
    la    $t1, T1IF
    sw    $t1, 0($t0)

    la    $t0, IFS0
loop:
    lw    $t2, 0($t0)
    and   $t3, $t2, $t1
    beq   $t3, $0, loop

    la    $t0, LATAINV
    addi  $t1, $0, 1
    sw    $t1, 0($t0)
    j     endless
```

MEMORY & I/O SYSTEMS

Operation types

Good design demands good compromises

- Multiple instruction formats allow flexibility
  - `add`, `sub`: use 3 register operands
  - `lw`, `sw`: use 2 register operands and a constant
- Number of instruction formats kept small

Instruction	Operands	Example
R-type	3	Arithmetic
I-type	2	Arithmetic/LD ST
J-type	1	JUMP

MEMORY & I/O SYSTEMS

Review: Instruction Formats

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

MEMORY & I/O SYSTEMS

Branching

000010 (2)	j	jump	PC = JTA
000011 (3)	jal	jump and link	$ra = PC+4, PC = JTA
000100 (4)	beq	branch if equal	if ([rs]==[rt]) PC = BTA
000101 (5)	bne	branch if not equal	if ([rs]!=[rt]) PC = BTA
000110 (6)	blez	branch if less than or equal to zero	if ([rs] ≤ 0) PC = BTA
000111 (7)	bgtz	branch if greater than zero	if ([rs] > 0) PC = BTA

- [reg]: contents of the register
- imm: 16-bit immediate field of the I-type instruction
- addr: 26-bit address field of the J-type instruction
- SignImm: sign-extended immediate
  = {{16{imm[15]}}, imm}
- ZeroImm: zero-extended immediate
  = {16'b0, imm}
- Address: [rs] + SignImm
- [Address]: contents of memory location Address
- BTA: branch target address[1]
  = PC + 4 + (SignImm << 2)
- JTA: jump target address
  = {(PC + 4)[31:28], addr, 2'b0}

ELSEVIER

MEMORY & I/O SYSTEMS

Power of the Stored Program

A computer is a simple automata that follows the following steps.

•**Fetches** an instruction from address in PC (a register)
•**Decodes** the instruction type
•**Execute**
  •Type R: Perform operation
  •Type I: Perform operation / Memory Access / Update PC
  •Type J : Updates PC
• **Store**. Update registers and Go to next instruction (given by PC)

ELSEVIER

MEMORY & I/O SYSTEMS

The Stored Program

Assembly Code	Machine Code
`lw   $t2, 32($0)`	0x8C0A0020
`add  $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub  $t0, $t3, $t5`	0x016D4022

Stored Program

Address	Instructions	
⋮	⋮	
0040000C	0 1 6 D 4 0 2 2	
00400008	2 2 6 8 F F F 4	
00400004	0 2 3 2 8 0 2 0	
00400000	8 C 0 A 0 0 2 0	← PC
⋮	⋮	

Main Memory

Program Counter (PC): keeps track of current instruction

MEMORY & I/O SYSTEMS

Ejercicio

Implementemos en ensamblador el siguiente código C:

```
while (1) {
    IFS0 = IFS0 & (~T1IF);
    while (! IFS0 & T1IF);
    LATAINV = 0x01;
}
```

MEMORY & I/O SYSTEMS

Solución

```
while (1) {
  IFS0 = IFS0 & (~T1IF);
  while (! IFS0 & T1IF);
  LATAINV = 0x01;
}
```

```
endless:
   la    $t0, IFS0CLR
   la    $t1, T1IF
   sw    $t1, 0($t0)

   la    $t0, IFS0
loop:
   lw    $t2, 0($t0)
   and   $t3, $t2, $t1
   beq   $t3, $0, loop

   la    $t0, LATAINV
   addi  $t1, $0, 1
   sw    $t1, 0($t0)
   j     endless
```

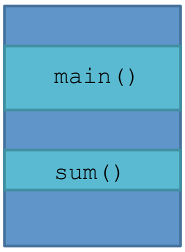
 Chapter 8 <33> ELSEVIER

MEMORY & I/O SYSTEMS

Function Calls

C Code

```
void
{
  int y;
  y = sum(42, 7);
  ...
}
```




```
int sum(int a, int b)
{
  return (a + b);
}
```

 Chapter 8 <34> ELSEVIER

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

MIPS Function Conventions

- **Call Function:** jump and link (`jal`)
- **Return** from function: jump register (`jr`)
- **Arguments**: $a0 - $a3
- **Return value**: $v0

Function Calls

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

MIPS assembly code

```
0x00400200 main: jal  simple
0x00400204       add  $s0, $s1, $s2
...

0x00401020 simple: jr $ra
```

`jal`: jumps to `simple`
`$ra` = PC + 4 = 0x00400204

`jr $ra`: jumps to address in `$ra` (0x00400204)

Ejercicio

Implementemos en ensamblador el siguiente código C:

```
if (PORTAbits.RA0) {
        do_porta(2);
} else {
        do_porta(5);
}
```

MEMORY & I/O SYSTEMS

Solución

```
if (PORTA & 0x01) {

    do_porta(2);

} else {

    do_porta(5);
}
```

```
        la    $t0, PORTA
        lw    $t1, 0($t0)
        ori   $t0, $0, 1
        and   $t2, $t1, $t0
        ori   $a0, $0, 2
        bne   $t2, $0, do_if
        ori   $a0, $0, 5
do_if:
        jal   do_porta
```


ELSEVIER

MEMORY & I/O SYSTEMS

Ejercicio: strcmp

Implementamos en ensamblador la función estándar

    int strcmp (const char* s1, const char* s2);

La función compara lexicográficamente las cadenas s1 y s2, ambas terminadas por el carácter '\0'.
La comparación se realiza letra a letra, como el orden en el diccionario, hasta que se encuentra una diferencia.
Un carácter es mayor que otro si el código ASCII que lo representa es mayor que el del otro. Si son iguales, el valor de retorno es 0. Si son iguales, pero uno termina y el otro no, el más corto es menor.
La función devuelve un número negativo si s1 es menor que s2, 0 si son iguales, mayor que 0 si s2 es menor que s1.

a0: s1
a1: s2

ELSEVIER

Ejercicio: strcmp

int strcmp (const char* s1, const char* s2);
a0: s1 a1: s2

```
strcmp:
        cargo dato de s1
        cargo dato de s2
        resto s1 - s2 en registro de retorno
        salto a fin si no son iguales
        salto a fin si el dato de s1 es 0
        s1 apunta al siguiente dato
        s2 apunta al siguiente dato
        salto a strcmp
fin:
        return (valor de retorno con el resultado de la resta)
```


MEMORY & I/O SYSTEMS

Solución: strcmp en ASM

int strcmp (const char* s1, const char* s2);
a0: s1 a1: s2

```
strcmp:
        lb      $t0, $a0
        lb      $t1, $a1
        sub     $v0, $t0, $t1
        bne     $v0, $0, fin
        beq     $t0, $0, fin
        addi    $a0, $a0, 1
        addi    $a1, $a1, 1
        j       strcmp
fin:
        jr      $ra
```

Solución: strcmp en C

```
int strcmp (const char* s1, const char* s2);
a0: s1                    a1: s2

int strcmp (const char* s1, const char* s2) {
        char    tmp = *s1 - *s2;
        while ((tmp == 0) && *s1) {
                s1++;
                s2++;
                tmp = *s1 - *s2;
        }
        return tmp;
}
```


MEMORY & I/O SYSTEMS

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to top of the stack

Address	Data	
7FFFFFFC	12345678	← $sp
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	← $sp
7FFFFFF0		
⋮	⋮	

How Functions use the Stack

- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # make space on stack
                      # to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
  add  $t1, $a2, $a3  # $t1 = h + i
  sub  $s0, $t0, $t1  # result = (f + g) - (h + i)
  add  $v0, $s0, $0   # put return value in $v0
  lw   $t1, 0($sp)    # restore $t1 from stack
  lw   $t0, 4($sp)    # restore $t0 from stack
  lw   $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12   # deallocate stack space
  jr   $ra            # return to caller
```

MEMORY & I/O SYSTEMS

ELSEVIER

Multiple Function Calls

```
proc1:
  addi $sp, $sp, -4    # make space on stack
  sw   $ra, 0($sp)     # save $ra on stack
  jal  proc2
  ...
  lw   $ra, 0($sp)     # restore $s0 from stack
  addi $sp, $sp, 4     # deallocate stack space
  jr  $ra              # return to caller
```

MEMORY & I/O SYSTEMS

ELSEVIER

MEMORY & I/O SYSTEMS

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to
                      # store one register
  sw   $s0, 0($sp)    # save $s0 on stack
                      # no need to save $t0 or $t1
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr   $ra            # return to caller
```

  ELSEVIER

MEMORY & I/O SYSTEMS

Registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
$s0-$s7	**$t0-$t9**
$ra	**$a0-$a3**
$sp	**$v0-$v1**
stack above $sp	**stack below $sp**

  ELSEVIER

MEMORY & I/O SYSTEMS

Ejercicio: factorial

```
factorial:
        bne     $a0, $0, fact_recursive
        addi    $v0, $0, 1
        jr      $ra

fact_recursive:
        addiu   $sp, $sp, -XX
        sw      R1, O1($sp)
        add     $s0, $a0, $0
        sw      R2, O2($sp)
        addi    $a0, $a0, -1
        jal     factorial
        lw      R2, O2($sp)
        mul     $v0, $v0, $a0
        lw      R1, O1($sp)
        addiu   $sp, $sp, XX
        jr      $ra
```

a. ¿Cómo se llama a la función para conocer el factorial de 10?
b. Según el programa, ¿cuál es el factorial de 0?
c. Las instrucciones en negrita gestionan la pila y los registros que se almacenan en ella. Ponga valores válidos a las etiquetas:

XX, O1, O2, R1, R2

ELSEVIER

MEMORY & I/O SYSTEMS

Solución: factorial

```
factorial:
        bne     $a0, $0, fact_recursive
        addi    $v0, $0, 1
        jr      $ra

fact_recursive:
        addiu   $sp, $sp, -8
        sw      $s0, 0($sp)
        add     $s0, $a0, $0
        sw      $ra, 4($sp)
        addi    $a0, $a0, -1
        jal     factorial
        lw      $ra, 4($sp)
        mul     $v0, $v0, $a0
        lw      $s0, 0($sp)
        addiu   $sp, $sp, 8
        jr      $ra
```

a. ¿Cómo se llama a la función para conocer el factorial de 10?

```
ori     $a0, $0, 10
jal     factorial
```

b. Según el programa, ¿cuál es el factorial de 0?

1 (primeras 3 líneas de factorial

c. Las instrucciones en negrita gestionan la pila y los registros que se almacenan en ella. Ponga valores válidos a las etiquetas:

XX: 8 O1: 0 (o 4) O2: 4 (o 0)
R1: $s0 R2: $ra

ELSEVIER

Function Call Summary

- **Caller**
  - Put arguments in $a0-$a3
  - Save any needed registers ($ra, maybe $t0-t9)
  - jal callee
  - Restore registers
  - Look for result in $v0
- **Callee**
  - Save registers that might be disturbed ($s0-$s7)
  - Perform function
  - Put result in $v0
  - Restore registers
  - jr $ra

MEMORY & I/O SYSTEMS

Addressing Modes

How do we address the operands?

- Register Only
- Immediate
- Base Addressing
- PC-Relative
- Pseudo Direct

Addressing Modes

Register Only

- Operands found in registers
  - **Example:** `add $s0, $t2, $t3`
  - **Example:** `sub $t8, $s1, $0`

Immediate

- 16-bit immediate used as an operand
  - **Example:** `addi $s4, $t5, -73`
  - **Example:** `ori  $t3, $t7, 0xFF`

Addressing Modes

Base Addressing

- Address of operand is:

  `base address + sign-extended immediate`

  - **Example:** `lw  $s4, 72($0)`
    - address = `$0 + 72`
  - **Example:** `sw  $t2, -25($t1)`
    - address = `$t1 - 25`

MEMORY & I/O SYSTEMS

Addressing Modes

PC-Relative Addressing

```
0x10                beq   $t0, $0, else
0x14                addi  $v0, $0, 1
0x18                addi  $sp, $sp, i
0x1C                jr          $ra
0x20        else:   addi  $a0, $a0, -1
0x24                jal   factorial
```

Assembly Code

```
beq $t0, $0, else
(beq $t0, $0, 3)
```

Field Values

op	rs	rt	imm
4	8	0	3
6 bits	5 bits	5 bits	5 bits 5 bits 6 bits

ELSEVIER

MEMORY & I/O SYSTEMS

Addressing Modes

Pseudo-direct Addressing

```
0x0040005C          jal   sum
...
0x004000A0   sum:   add   $v0, $a0, $a1
```

JTA 0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)

26-bit addr 0000 0000 0100 0000 0000 0000 1010 0000 (0x0100028)

0 1 0 0 0 2 8

Field Values

op	imm
3	0x0100028
6 bits	26 bits

Machine Code

op	addr	
000011	00 0001 0000 0000 0000 0010 1000	(0x0C100028)
6 bits	26 bits	

ELSEVIER

MEMORY & I/O SYSTEMS

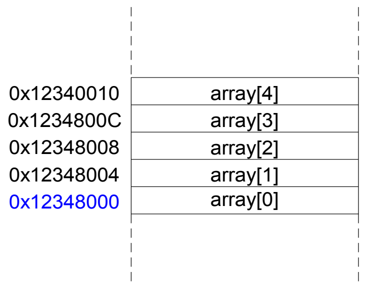
Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

MEMORY & I/O SYSTEMS

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
    int array[5];
    array[0] = array[0] * 2;
    array[1] = array[1] * 2;
```


MEMORY & I/O SYSTEMS

Accessing Arrays

```
// C Code
    int array[5];
    array[0] = array[0] * 2;
    array[1] = array[1] * 2;

# MIPS assembly code
# array base address = $s0
  lui  $s0, 0x1234          # 0x1234 in upper half of $S0
  ori  $s0, $s0, 0x8000     # 0x8000 in lower half of $s0

  lw   $t1, 0($s0)          # $t1 = array[0]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 0($s0)          # array[0] = $t1

  lw   $t1, 4($s0)          # $t1 = array[1]
  sll  $t1, $t1, 1          # $t1 = $t1 * 2
  sw   $t1, 4($s0)          # array[1] = $t1
```

MEMORY & I/O SYSTEMS

Arrays using For Loops

```
// C Code
  int array[1000];
  int i;

  for (i=0; i < 1000; i = i + 1)
      array[i] = array[i] * 8;

# MIPS assembly code
# $s0 = array base address, $s1 = i
```


MEMORY & I/O SYSTEMS

Arrays Using For Loops

```
# MIPS assembly code
# $s0 = array base address, $s1 = i
# initialization code
  lui  $s0, 0x23B8        # $s0 = 0x23B80000
  ori  $s0, $s0, 0xF000   # $s0 = 0x23B8F000
  addi $s1, $0, 0         # i = 0
  addi $t2, $0, 1000      # $t2 = 1000

loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if not then done
  sll  $t0, $s1, 2        # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0      # address of array[i]
  lw   $t1, 0($t0)        # $t1 = array[i]
  sll  $t1, $t1, 3        # $t1 = array[i] * 8
  sw   $t1, 0($t0)        # array[i] = array[i] * 8
  addi $s1, $s1, 1        # i = i + 1
  j    loop               # repeat
done:
```

Concepto: puntero

```
int a, b;
int* address;

a = 3;
address = &a;
b = *address;
```

```
int a, b;
int* address;
int my_array[5];

my_array[0] = 0;
my_array[1] = 1;
address = my_array;
a = *address;
b = *(address+1);
```


Punteros

```
# MIPS assembly code
 li  $t0, 0x12348000          #0x12348000 address of a
 li  $t1, 0x12348004          #0x12348004 address of address
 li  $t2, 0x12348004          #0x12348008 address of b
 addi  $t3, $0, 3             #Load constant
 sw   $t3, 0($t0)             # a = 3
 sw   $t0, 0($t1)             # address = &a
 lw   $t0, 0($t1)
 lw   $t3, 0($t0)
 sw   $t3, 0($t2)             # b = *address
```

MEMORY & I/O SYSTEMS

Ejercicio: Punteros

Sustituye los valores A, B, C

```
int buffer[4];
int i;
for (i = 0; A; i++) {
   buffer[i] = i+'0';
   printf("buffer es %d\n", B);
}

printf("La primera dirección de memoria del array es %d\n", C);
```

A:
a) sizeof(buffer)
b) 3. El último se reserva para el caracter '\0'
c) sizeof(int)*4
d) sizeof(buffer)/sizeof(int)

B:
a) *(buffer+i)
b) &buffer[i]
c) buffer+*i
d) &buffer+i

C:
a) &buffer
b) buffer
c) *buffer
d) buffer+i

ELSEVIER

MEMORY & I/O SYSTEMS

Solución: Punteros

Sustituye los valores A, B, C

```
int buffer[4];
int i;
for (i = 0; A; i++) {
   buffer[i] = i+'0';
   printf("Siguiente elemento: %d\n", B);
}

printf("La primera dirección de memoria del array es %d\n", C);
```

A:
a) sizeof(buffer)
b) 3. El último se reserva para el caracter '\0'
c) sizeof(int)*4
d) sizeof(buffer)/sizeof(int)

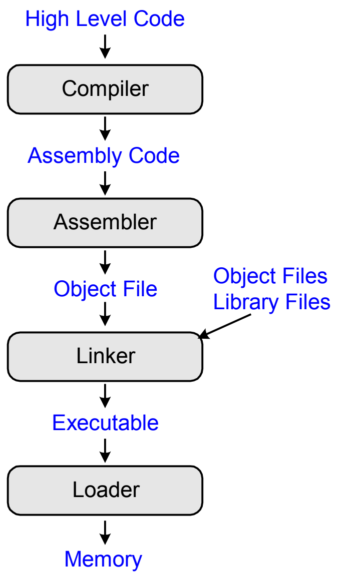
B:
a) *(buffer+i)
b) &buffer[i]
c) buffer+*i
d) &buffer+i

C:
a) &buffer
b) buffer
c) *buffer
d) buffer+i

ELSEVIER

MEMORY & I/O SYSTEMS

How to Compile & Run a Program



MEMORY & I/O SYSTEMS

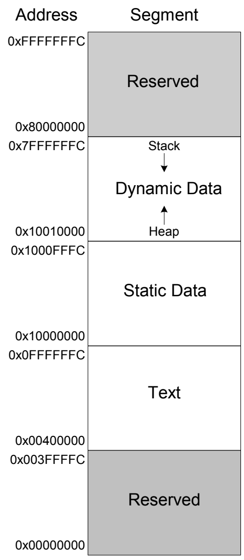
What is Stored in Memory?

- Instructions (also called *text*)
- Data
  - Global/static: allocated before program begins
  - Dynamic: allocated within program

- How big is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

MEMORY & I/O SYSTEMS

MIPS Memory Map



MEMORY & I/O SYSTEMS

Example Program: C Code

```
int f, g, y;  // global variables


int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);

  return y;
}


int sum(int a, int b) {
  return (a + b);
}
```

MEMORY & I/O SYSTEMS

Example Program: MIPS Assembly

```
int f, g, y;   // global

int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}

int sum(int a, int b) {
  return (a + b);
}
```

```
.data
f:
g:
y:
.text
main:
  addi $sp, $sp, -4    # stack frame
  sw   $ra, 0($sp)     # store $ra
  addi $a0, $0, 2      # $a0 = 2
  sw   $a0, f          # f = 2
  addi $a1, $0, 3      # $a1 = 3
  sw   $a1, g          # g = 3
  jal  sum             # call sum
  sw   $v0, y          # y = sum()
  lw   $ra, 0($sp)     # restore $ra
  addi $sp, $sp, 4     # restore $sp
  jr   $ra             # return to OS
sum:
  add  $v0, $a0, $a1   # $v0 = a + b
  jr   $ra             # return
```


ELSEVIER

MEMORY & I/O SYSTEMS

Example Program: Symbol Table

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

ELSEVIER

MEMORY & I/O SYSTEMS

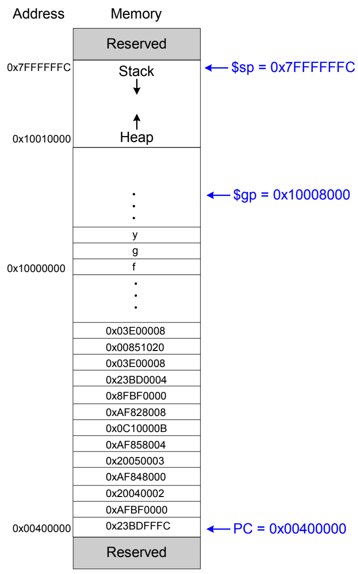
Example Program: Executable

Executable file header	Text Size	Data Size	
	0x34 (52 bytes)	0xC (12 bytes)	
Text segment	**Address**	**Instruction**	
	0x00400000	0x23BDFFFC	addi $sp, $sp, -4
	0x00400004	0xAFBF0000	sw $ra, 0 ($sp)
	0x00400008	0x20040002	addi $a0, $0, 2
	0x0040000C	0xAF848000	sw $a0, 0x8000 ($gp)
	0x00400010	0x20050003	addi $a1, $0, 3
	0x00400014	0xAF858004	sw $a1, 0x8004 ($gp)
	0x00400018	0x0C10000B	jal 0x0040002C
	0x0040001C	0xAF828008	sw $v0, 0x8008 ($gp)
	0x00400020	0x8FBF0000	lw $ra, 0 ($sp)
	0x00400024	0x23BD0004	addi $sp, $sp, -4
	0x00400028	0x03E00008	jr $ra
	0x0040002C	0x00851020	add $v0, $a0, $a1
	0x00400030	0x03E00008	jr $ra
Data segment	**Address**	**Data**	
	0x10000000	f	
	0x10000004	g	
	0x10000008	y	

ELSEVIER

MEMORY & I/O SYSTEMS

Example Program: In Memory



ELSEVIER

MEMORY & I/O SYSTEMS

Chapter 8

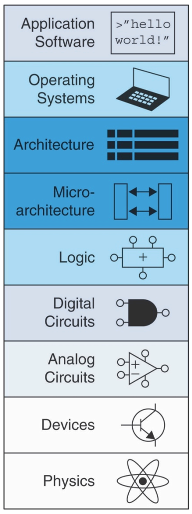
***Digital Design and Computer Architecture*, 2nd Edition**

David Money Harris and Sarah L. Harris

MEMORY & I/O SYSTEMS

Chapter 8 :: Topics

- **Introduction**
- Memory System Performance Analysis
- Caches
- Virtual Memory
- **Memory-Mapped I/O**
- **Summary**



MEMORY & I/O SYSTEMS

New Concepts

- Memory Hierarchy
- Embedded System: microcontroller
  - Peripherals
  - Memory Mapping
- Interrupts

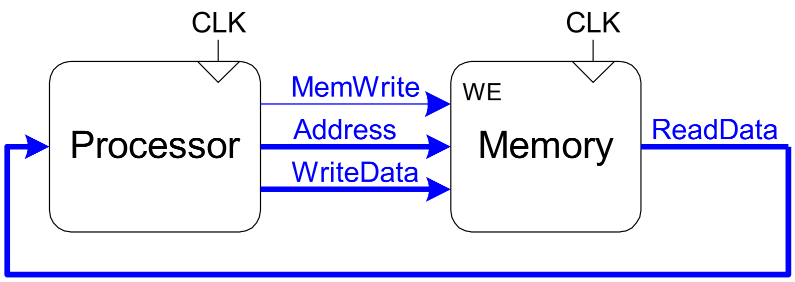
ELSEVIER

MEMORY & I/O SYSTEMS

Memory Hierarchy

- Computer performance depends on:
  - Processor performance
  - Memory system performance

Memory Interface



ELSEVIER

uC: Memory-Mapped I/O

- Processor accesses I/O devices just like memory (like keyboards, monitors, printers)
- Each I/O device assigned one or more address
- When that address is detected, data read/written to I/O device instead of memory
- A portion of the address space dedicated to I/O devices

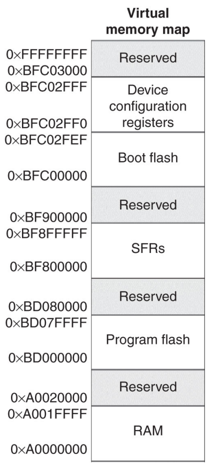


Figure 8.30 PIC32 memory map
(? 2012 Microchip Technology Inc.; reprinted with permission.)

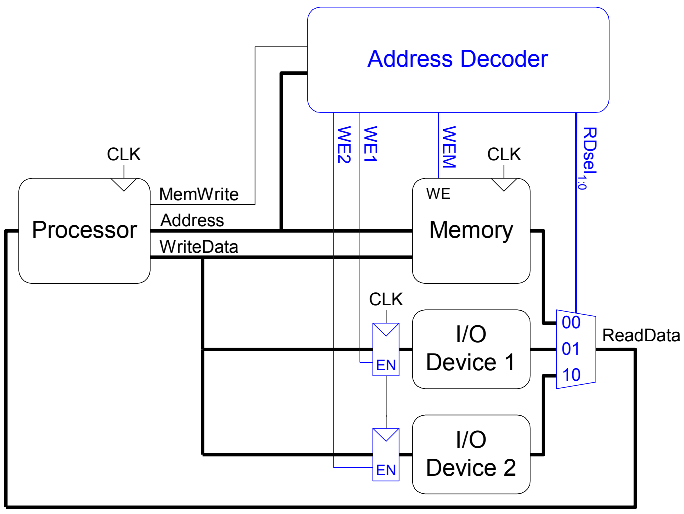
Memory-Mapped I/O Hardware

MEMORY & I/O SYSTEMS

- **Address Decoder:**
  – Looks at address to determine which device/ memory communicates with the processor
- **I/O Registers:**
  – Hold values written to the I/O devices
- **ReadData Multiplexer:**
  – Selects between memory and I/O devices as source of data sent to the processor

Memory-Mapped I/O Hardware

MEMORY & I/O SYSTEMS



I/O Peripherals

- Types
  - GPIO: Leds, switch, configurations
  - Serial: SPI, UART, bluetooth
  - Data: DAC ADC
- SW Interface: memory mapped registers
  - Configuration
  - Use

MEMORY & I/O SYSTEMS

Embedded I/O Systems

- Example microcontroller: PIC32
  - microcontroller
  - 32-bit MIPS processor
  - low-level peripherals include:
    - GPIO
    - serial ports
    - timers
    - A/D converters

Digital I/O: GPIO

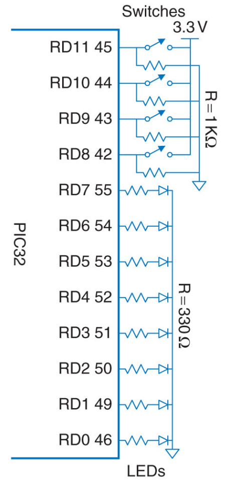


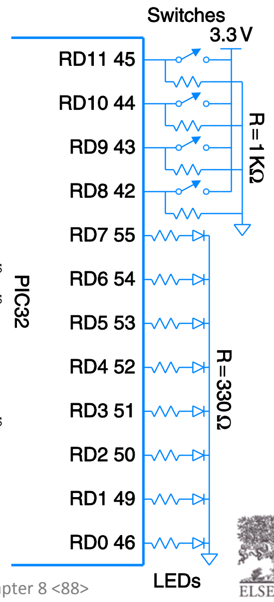
Figure 8.35 LEDs and switches connected to 12-bit GPIO port D

Digital I/O: GPIO

C Program to red 4 switches and turn on the corresponding LEDs

```
// C Code
#include <p3xxxx.h>

int main(void) {
  int switches;
  TRISD = 0xFF00;       // RD[7:0] outputs
                        // RD[11:8] inputs
  while (1) {
    // read & mask switches, RD[11:8]
    switches = (PORTD >> 8) & 0xF;
    PORTD = switches;   // display on LEDs
  }
}
```



MEMORY & I/O SYSTEMS

Ejercicio

¿Cuál es el valor de los siguientes registros después de ejecutar las siguientes instrucciones?
Registros: $t0, $t1
SFR: PORTA, ANSELA, LATB, PORTB

No se ha modificado nada más que lo que se ve.
El valor del puerto A en binario es 00110110
El valor del puerto B en binario es 01010001

```
ANSELA = 0xBF886000
.global ANSELA
PORTA = 0xBF886020
.global PORTA
LATB = 0xBF886130
.global LATB
```

```
la      $t0, ANSELA
sw      $t0, 0($t0)

la      $t0, PORTA
lw      $t1, 0($t0)

la      $t0, LATB
sw      $t1, 0($t0)
```

$t0:	LATB (0xBF886130)
$t1:	00110110 (0x0036)
PORTA:	00110110 (0x0036)
ANSELA:	0 (0x00)
LATB:	00110110 (0x0036)
PORTB:	01010001 (0x0051)

ELSEVIER

MEMORY & I/O SYSTEMS

Solución

```
while (1) {
   IFS0 = IFS0 & (~T1IF);
   while (! IFS0 & T1IF);
   LATAINV = 0x01;
}
```

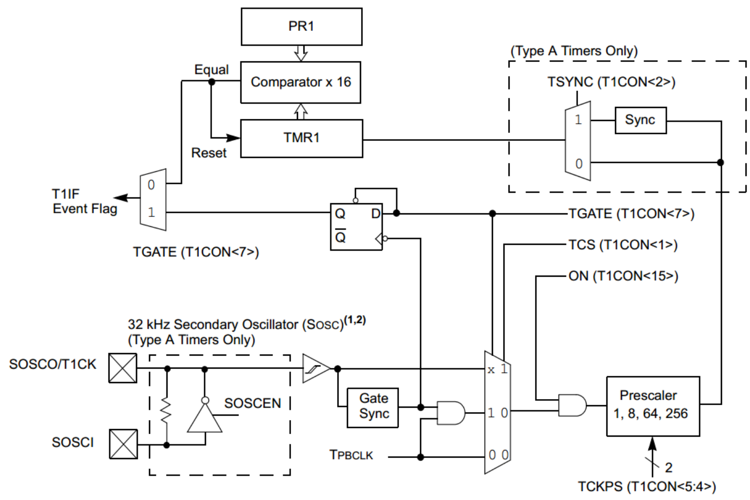
```
endless:
   la     $t0, IFS0CLR
   la     $t1, T1IF
   sw     $t1, 0($t0)

   la     $t0, IFS0
loop:
   lw     $t2, 0($t0)
   and    $t3, $t2, $t1
   beq    $t3, $0, loop

   la     $t0, LATAINV
   addi   $t1, $0, 1
   sw     $t1, 0($t0)
   j      endless
```


ELSEVIER

Figure 14-1: Type A Timer Block Diagram



Note 1: For information on enabling the 32 kHz Secondary Oscillator (Sosc), refer to **Section 6. "Oscillators"** (DS61112).

2: The default state of the SOSCEN bit (OSCCON<1>) during a device Reset is controlled by the FSOSCEN bit (DEVCFG1<5>).

MEMORY & I/O SYSTEMS

Each Timer module is a 16-bit timer that consists of the following Registers:
- T1CON: Type A Timer Control Register
- TxCON: Type B Timer Control Register
- TMRx: Timer Register
- PRx: Period Register

Each Timer module also has the following associated bits for interrupt control:
- TxIE: Interrupt Enable Control bit in IEC0 interrupt register
- TxIF: Interrupt Flag Status bit in IFS0 interrupt register
- TxIP<2:0>: Interrupt Priority Control bits in IPC1, IPC2, IPC3, IPC4, and IPC5 interrupt registers
- TxIS<1:0>: Interrupt Subpriority Control bits in IPC1, IPC2, IPC3, IPC4, and IPC5 interrupt registers

MEMORY & I/O SYSTEMS

Ejercicio: timer de práctica

Queremos configurar el timer para que marque un período de 100 us y 100ms.
La frecuencia del sistema es 8 MHz.
El timer es de 16-bits.
Los posibles valores de prescaler son 1:1, 1:8, 1:64 y 1:256.
El timer termina cuando TMR1 supera el valor de PR1

Solución:
$F_{TMR} = F_{SYSTEM} / (PRESCALER \times (PR1+1))$

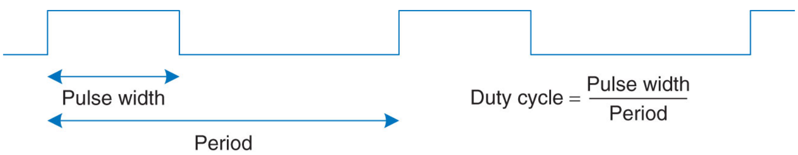
100 us -> 10KHz: PR1 = 8000000 / (10000 x 1) - 1 = round (800) – 1 = 799
Prescaler: 1:1, PR1: 799

~~100 ms -> 10Hz: PR1 = 8000000 / (10 x 1) - 1 = round (800000) – 1 = 799999~~
~~100 ms -> 10Hz: PR1 = 8000000 / (10 x 8) - 1 = round (100000) – 1 = 99999~~
100 ms -> 10Hz: PR1 = 8000000 / (10 x 64) - 1 = round (12500) – 1 =12499
Prescaler: 1:64, PR1: 12499

ELSEVIER

MEMORY & I/O SYSTEMS

PWM



Dos registros con comparador:

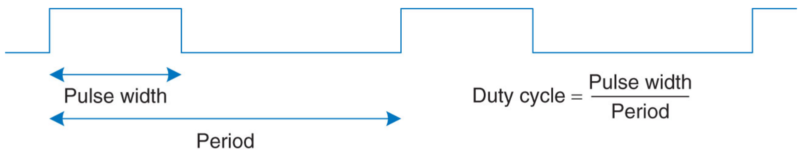
Periodo: PRx (PR2)
Duty cycle: OCxR (OC2R)

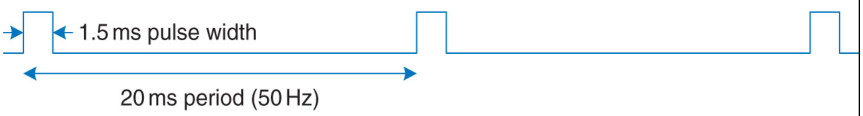
TMRx llega a OCxR: cambia pin de salida de 1 a 0
TMRx sobrepasa a PRx: reinicia cuenta (TMRx = 0) y pone pin a 1

ELSEVIER

MEMORY & I/O SYSTEMS

Ejercicio: PWM para servo



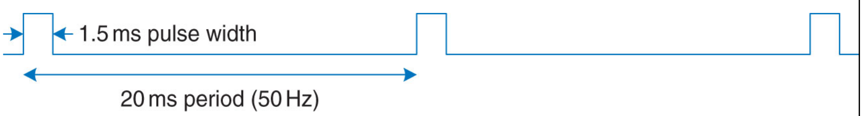


El servo se coloca en el ángulo 0º con un pulso de 0.5ms y en el ángulo 180º con 2.5ms.

¿Qué valor de PR2 y OC2R hay que poner para colocar el servo en 90º? La frecuencia del sistema es 8MHz, y los prescaler disponibles son 1:1, 1:8, 1:64 y 1:256

MEMORY & I/O SYSTEMS

Solución: PWM para servo



PASO 1: Período

$F_{TMR} = 1/T_{TMR} = F_{SYSTEM}$ / (PRESCALER x (PR2 + 1))

~~PR2 = 8000000 / (50 x 1) – 1 = 160000~~

PR2 = 8000000 / (50 x 8) – 1 = 20000 – 1 = 19999

PASO 2: Duty cycle

$1/T_{HIGH} = F_{SYSTEM}$ / (PRESCALER x OC2R)

$T_{90º}$ = 1.5ms

OC2R = 8000000 x 0.0015 / 8 = 1500

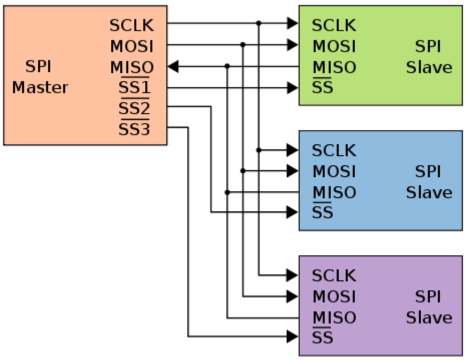
Prescaler= 1:8 PR2 = 19999 OC2R = 1500

Serial I/O: Serial

- Example serial protocols
  - **SPI:** Synchronous Serial Peripheral Interface
  - **UART:** Universal Asynchronous Receiver/ Transmitter
  - Also: I^2C, USB, Ethernet, etc.

MEMORY & I/O SYSTEMS

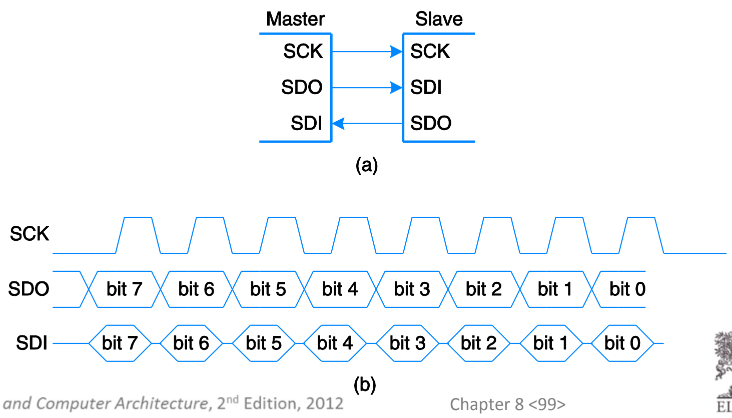
SPI connection



MEMORY & I/O SYSTEMS

SPI: Serial Peripheral Interface

- Synchronous interface
- Master initiates communication to slave by sending pulses on SCK
- Master sends SDO (Serial Data Out) to slave, msb first
- Slave may send data (SDI) to master, msb first



ELSEVIER

MEMORY & I/O SYSTEMS

UART: Universal Asynchronous Rx/Tx

Asynchronous interface
Master (DTE) initiates communication to slave by data through TX
Slave (DCE) may reply through RX

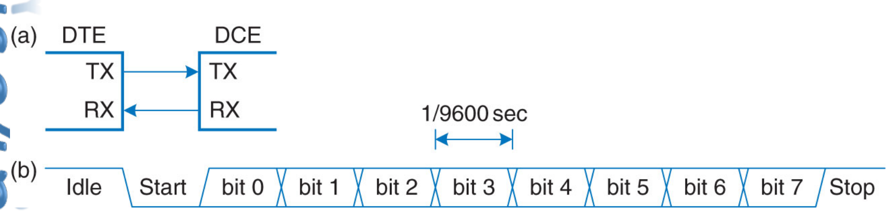


Figure 8.40 Asynchronous serial link

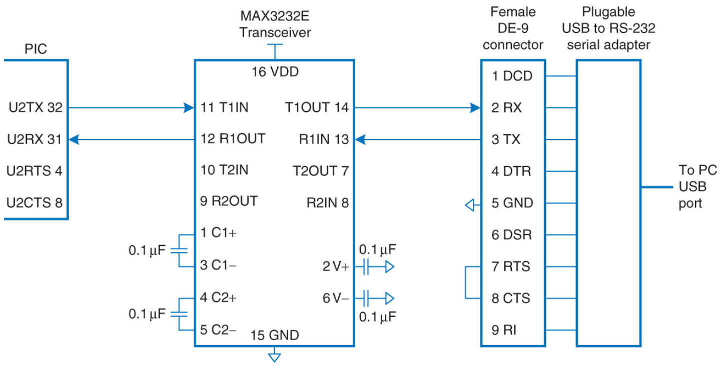
Concepts
-Idle-Start transtion
-Start-stop length
-Parity bit
-RS232 & DB9

ELSEVIER

MEMORY & I/O SYSTEMS

UART: Communication with PC

Connection from PIC to PC, to show data on terminal
With 115.2kbaud and 8 bits



Voltage Level Adjustment

Figure 8.43 PIC32 to PC serial link

ELSEVIER

MEMORY & I/O SYSTEMS

Ejercicio: Frecuencia de UART

Se desea habilitar el módulo UART 2 para la transmisión a 57600 baudios.
La frecuencia del sistema es 80 MHz
El prescaler seleccionado para la UART es 1:16
¿Qué valor hay que poner en el registro BRG2, que controla el período de la frecuencia de transmisión?
¿Cuál es la frecuencia de transmisión real que se obtiene?

Solución:
$F_{UART} = F_{SYSTEM}$ / (PRESCALER x (BRG2+1))

BRG2 = 80000000 / (57600 x 16) - 1 = round (86,8) – 1 = 86

$F_{UART\ real}$ = 80000000 / (16 x 87) = 57471 baudios

ELSEVIER

MEMORY & I/O SYSTEMS

Analog I/O

- Needed to interface with outside world (real)
- **Analog input:** Analog-to-digital (A/D) conversion
  - Often included in microcontroller
  - *N*-bit: converts analog input from V_{ref-}-V_{ref+} to 0-2^{N-1}
- **Analog output:**
  - Digital-to-analog (D/A) conversion
    - Typically need external chip (e.g., AD558 or LTC1257)
    - *N*-bit: converts digital signal from 0-2^{N-1} to V_{ref-}-V_{ref+}
  - Pulse-width modulation

MEMORY & I/O SYSTEMS

Analog I/O

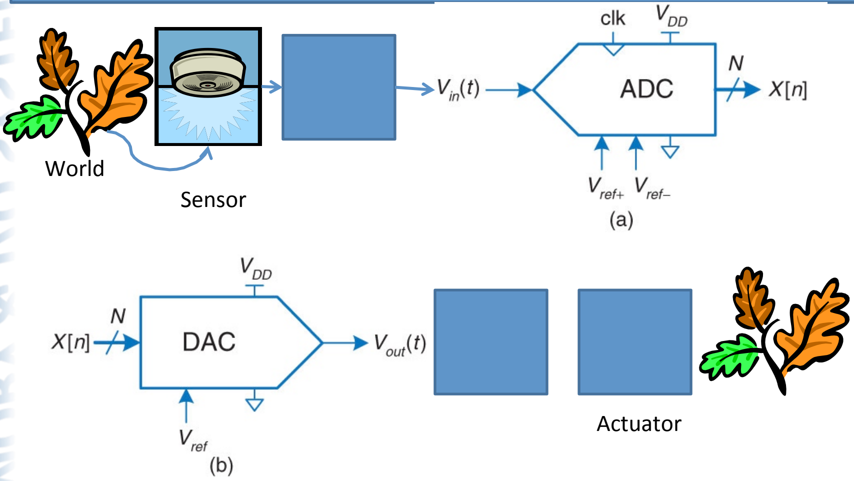


Figure 8.45 ADC and DAC symbols

MEMORY & I/O SYSTEMS

Analog I/O

$$x[n] = 2^N \frac{V_{in}(t) - V_{ref^-}}{V_{ref^+} - V_{ref^-}}$$

$$t = n \cdot T_s$$

N Bit resolution
Fs=1/Ts: Sampling Rate

Digital signal : sampling is related to analog signal Nyquist cryterion (link with DSP)

Many microcontrollers have buil-in ADCs of moderate performance
The PIC has a 10bit ADC with maximum speed of 1MHz
The ADC can connect to 16 analog pins with a multiplexer
Multiple registers for control

MEMORY & I/O SYSTEMS

Ejercicio ADC

$$x[n] = 2^N \frac{V_{in}(t) - V_{ref^-}}{V_{ref^+} - V_{ref^-}}$$

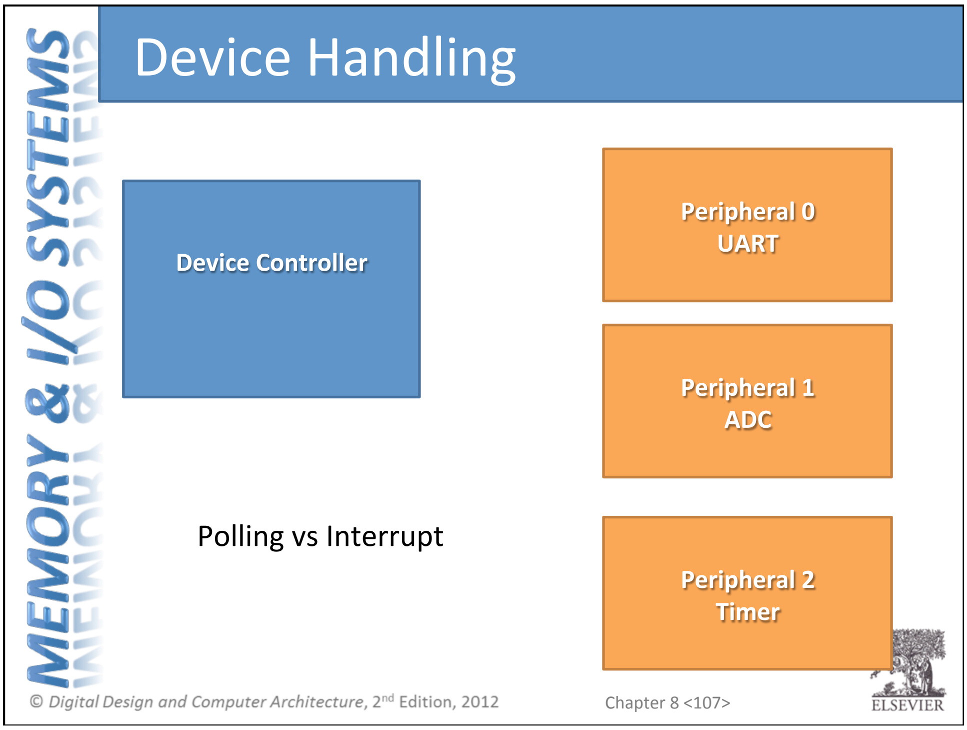
Tenemos un ADC de 12 bits, con V_{ref+} de 2.5V y V_{ref-} de 0V

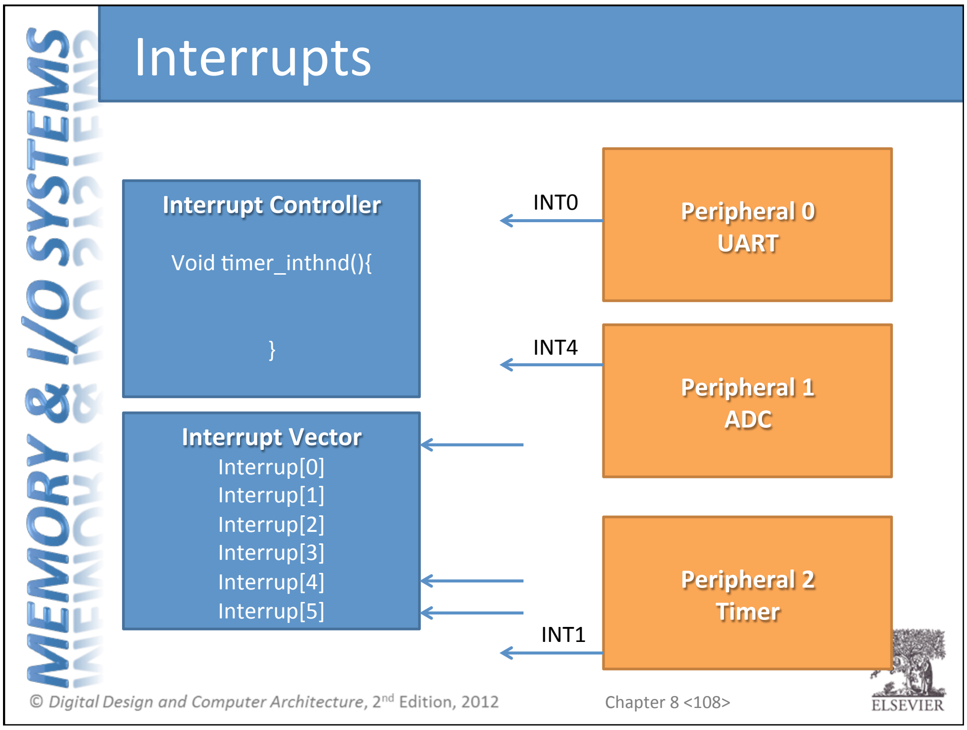
1. ¿Cuál es el valor que devuelve el ADC si la señal analógica es de 1V?

ADC = $2^N (V_{in} - V_{ref-})/(V_{ref+} - V_{ref-})$ = 4096 x 1 / 2.5 = 1638

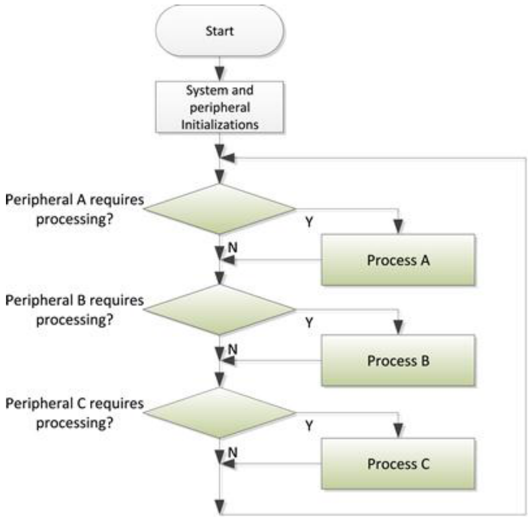
2. ¿Qué valor de tensión corresponde a un valor de 819?

ADC = $2^N (V_{in} - V_{ref-})/(V_{ref+} - V_{ref-})$; V_{in} = ADC x $(V_{ref+} - V_{ref-}) / 2^N + V_{ref-}$
V_{in} = 819 x 2.5 / 4096 = 0.5V


MEMORY & I/O SYSTEMS
Device Handling
Device Controller
Polling vs Interrupt
Peripheral 0
UART
Peripheral 1
ADC
Peripheral 2
Timer
© Digital Design and Computer Architecture, 2nd Edition, 2012
Chapter 8 <107>
ELSEVIER


MEMORY & I/O SYSTEMS
Interrupts
Interrupt Controller
Void timer_inthnd(){
}
Interrupt Vector
Interrup[0]
Interrup[1]
Interrup[2]
Interrup[3]
Interrup[4]
Interrup[5]
INT0
INT4
INT1
Peripheral 0
UART
Peripheral 1
ADC
Peripheral 2
Timer
© Digital Design and Computer Architecture, 2nd Edition, 2012
Chapter 8 <108>
ELSEVIER

Flujo de programa: polling

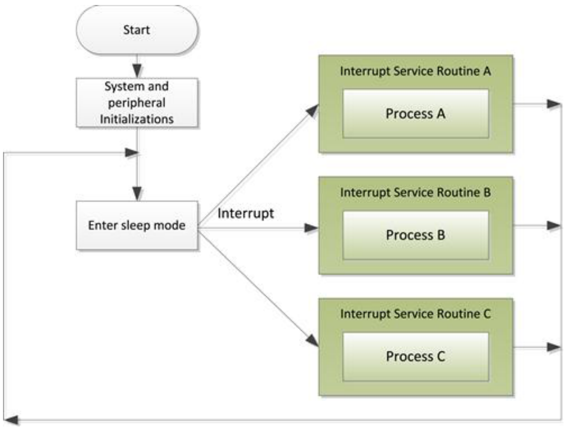


- Continuo
  - Consumo
- Bloqueo
  - Prioridades

Interrupciones

- Parar procesado secuencial
  - Guardar contexto (registros, pila, …)
- Habilitar en distintos niveles: individual y global
  - Normalmente el *flag* sigue activándose
- Limpieza de *flag* (evitar interrupción continua)
- Prioridades (de 0 a 7, más alto => más prior.)
- Vectores de *reset*: tabla con punteros a función
  - Si se comparte hay que comprobar por SW
- Tiempo preciso o liberar carga: interrupción

Flujo de programa: interrupt



- Continuo ✓
  - Consumo
- Bloqueo ✓
  - Prioridades
- Prioridad atómica
  - División en fases

ELSEVIER

MEMORY & I/O SYSTEMS

Interrupts

Advantage	Energy Efficient
Disadvantage	Flow is no sequential Debugging: testing

Ejercicio: ping-pong buffer para almacenar datos y transmitir por RF (entregado en clase)

ELSEVIER

Ejercicio

Implementar una función que implemente un ping-pong buffer para almacenamiento de 200 bytes que llegan por UART, 1000 bytes/s.

```
void add_data(char data);
```

Variables globales:

```
#define TX_BYTES 200
char buf1[TX_BYTES];
char buf2[TX_BYTES];
char* buf = buf1;
unsigned char rx_count = 0;
```

Función de transmisión por RF

```
void rf_tx(char* data, unsigned char len);
```


MEMORY & I/O SYSTEMS

Solución

```
void add_data(char data) {
   buf[rx_count] = data;
   rx_count++;
   if (rx_count >= TX_BYTES) {
      rx_count = 0;
      if (buf == buf1) {
         buf = buf2;
         rf_tx (buf1, TX_BYTES);
      } else {
         buf = buf1;
         rf_tx (buf2, TX_BYTES);
      }
}
```