

Cap 2. Lenguaje del computador . Parte 2:

Arquitectura de conjunto de instrucciones (Instruction Set Architecture – ISA) - Patterson y Hennessy

Ing. Hernán Quito
Universidad de Cuenca

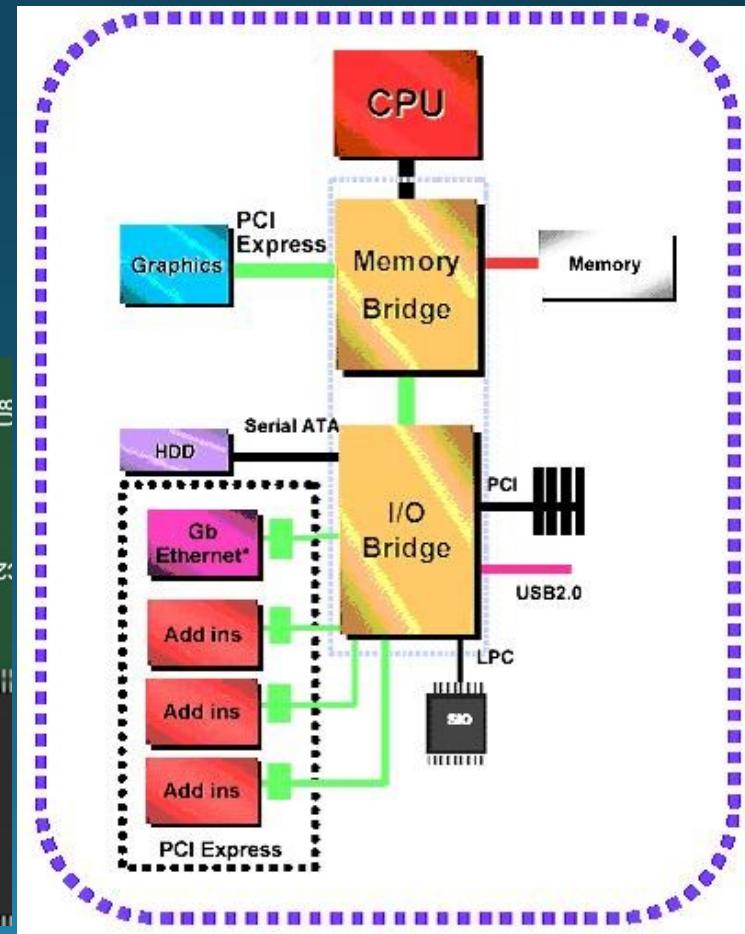
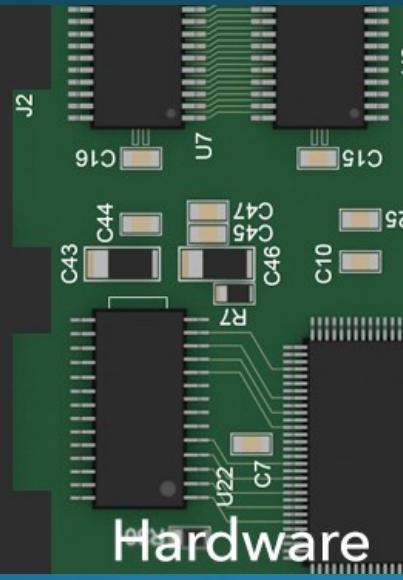
```
deep.route('/about')
def about():
    return render_template('about.html')

deep.route('/feeds/')
deep.route('/feeds/<int:uid>')
deep.route('/feeds/uid/<task>', methods=['GET'])
def feeds(uid=None, task=None):
    feed = None
    try:
        feed = Feed.query.filter_by(id=uid).first()
        if task == 'list':
            route('/feeds/{}'.format(feed.id))
        elif task == 'intel':
            route('/feeds/{}<int:uid>'.format(feed.id))
        else:
            feed = toDict(db.session.query(Feed).filter_by(id=uid).first())
            if task == 'update':
                print request.form
                update_feed(feed, request.form)
                return render_template('update.html', feed=feed)
            else:
                feed['last_update'] = feed['last_update'].isoformat()
                print feed
                return render_template('about.html', feed=feed)
    except:
        pass
    finally:
        db.session.close()

deep.route('/feeds/uid')
def feeds(uid):
    feed = toDict(db.session.query(Feed).filter_by(id=uid).first())
    if task == 'update':
        print request.form
        update_feed(feed, request.form)
        return render_template('update.html', feed=feed)
    else:
        feed['last_update'] = feed['last_update'].isoformat()
        print feed
        return render_template('about.html', feed=feed)
```

Instruction
Set
Architecture

Software



Software

Compilador

Ensamblador

*Application software
programa en C:*

```
swap (int v[ ], int k)
{int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

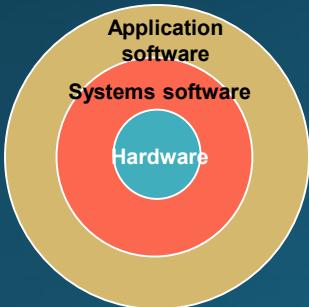
*Salida de compilador MIPS,
Programa en lenguaje ensamblador*

```
swap:
    multi    $2, $5, 4
    add     $2, $4, $2
    lw      $15, 0 ($2)
    lw      $16, 4 ($2)
    sw      $16, 0 ($2)
    sw      $15, 4 ($2)
    jr      $31
```

*Código de máquina binario
MIPS*

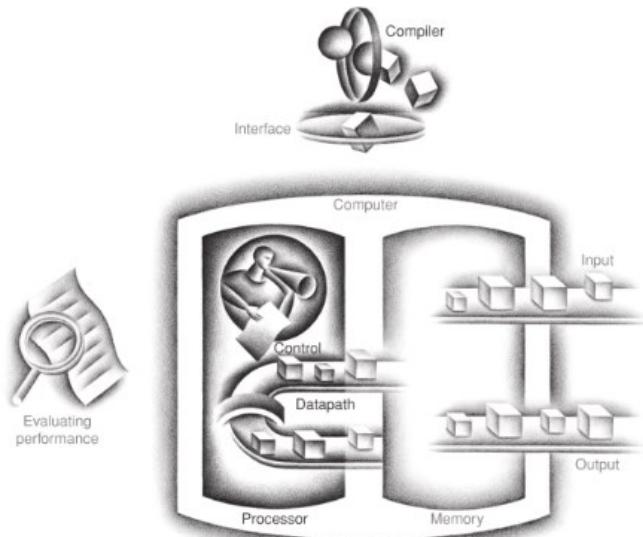
```
0000000010100010000000000011000
0000000000011000001100000100001
10001100011000100000000000000000
10001100111100100000000000000000
10101100111100100000000000000000
10101100011000100000000000000000
00000011111000000000000000000000
```

The Five Classic Components of a Computer



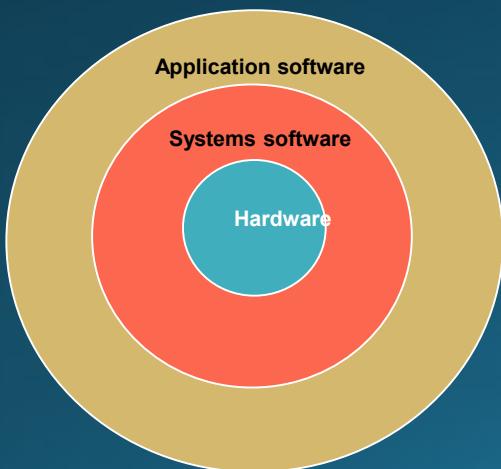
Conjunto de instrucciones
El vocabulario de los comandos entendidos por una arquitectura dada.

MIPS (Microprocessor without Interlocked Pipeline Stages)



Instruction Set Architecture (ISA)

- Un conjunto de instrucciones en lenguaje ensamblador (ISA) proporciona un enlace entre el software y el hardware.
 - Dado un conjunto de instrucciones, los programadores de software y los ingenieros de hardware trabajan de forma más o menos independiente.
- ISA está diseñado para extraer el máximo rendimiento de la tecnología de hardware disponible.



Software



Hardware

ISA

- ISA Define
 - Registros
 - Modos de transferencia de datos (instrucciones)
 - entre registros, memoria y E / S.
 - *Debe haber suficientes instrucciones para traducir de manera eficiente cualquier programa para el procesamiento de la máquina*
 - Formato del conjunto de instrucciones:
 - representación binaria utilizada por el hardware
 - Instrucciones de longitud variable frente a longitud fija

Tipos de ISA

- Conjunto de instrucciones complejas (CISC)
 - Muchas instrucciones (varios cientos)
 - Una instrucción tarda muchos ciclos en ejecutarse.
 - Ejemplo: Intel Pentium
- Conjunto de instrucciones reducido (RISC)
 - Pequeño conjunto de instrucciones (normalmente 32)
 - Instrucciones simples, cada una se ejecuta en un ciclo de reloj
 - - ¿REALMENTE? Bueno, casi.
 - Uso efectivo de la canalización.
 - Ejemplo: ARM

Lectura:

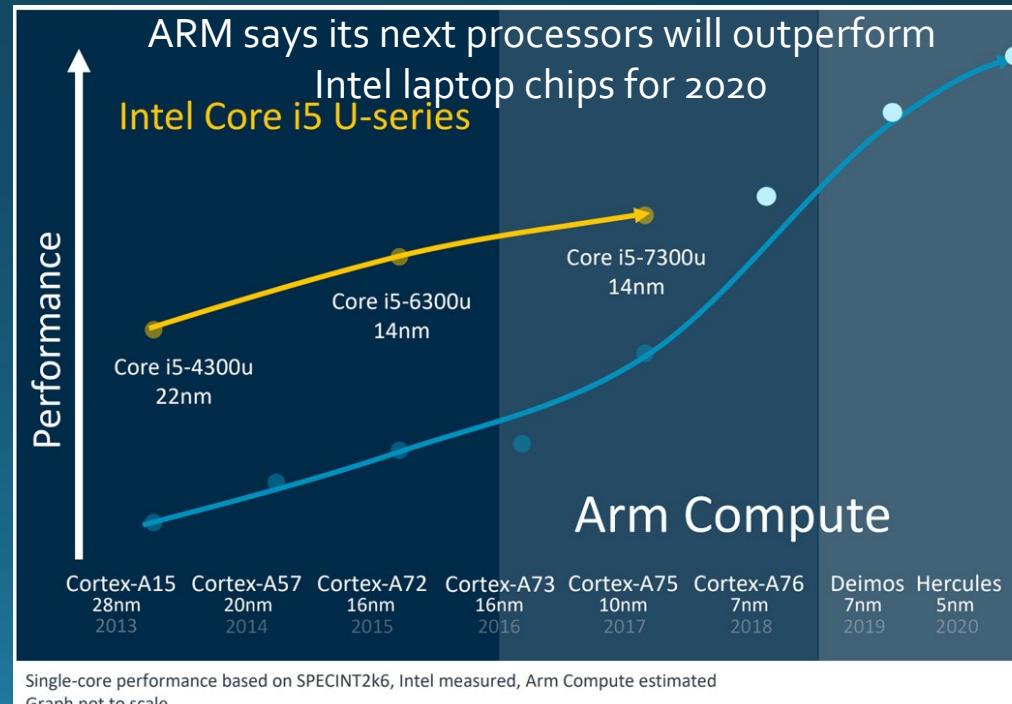
Brad Smith, "ARM and Intel Battle over the Mobile Chip's Future," Computer, vol. 41, no. 5, pp. 15-18, May 2008.

Compara 3Ps:

Performance

Power consumption

Price



Segmentacion de cauce de instrucciones RIS

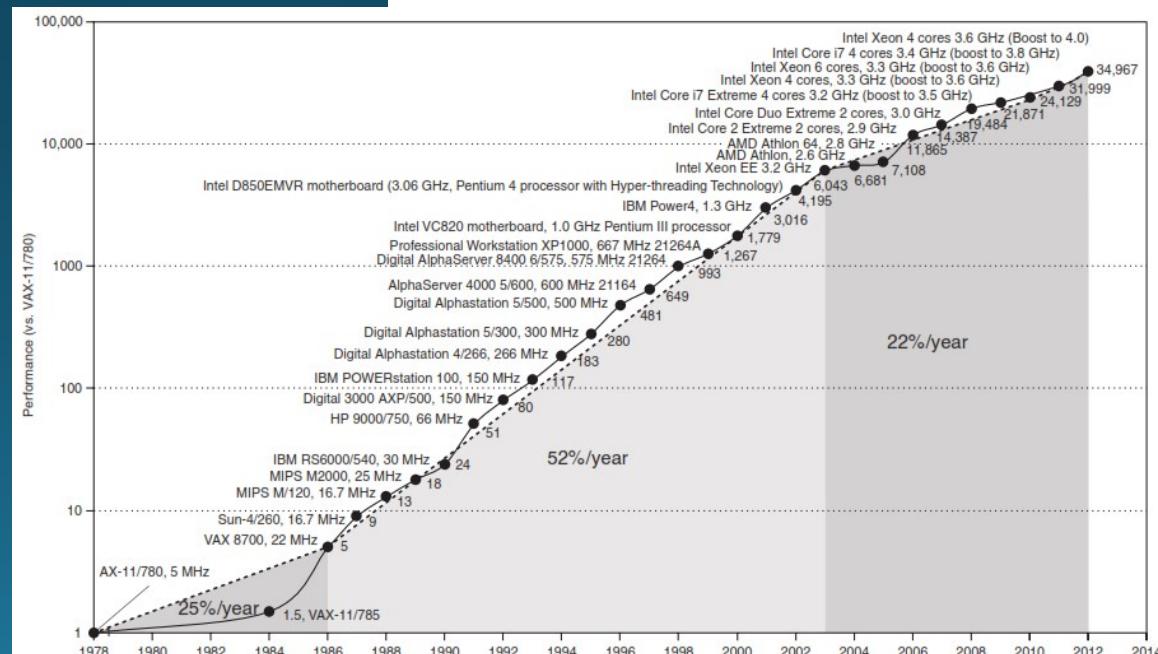
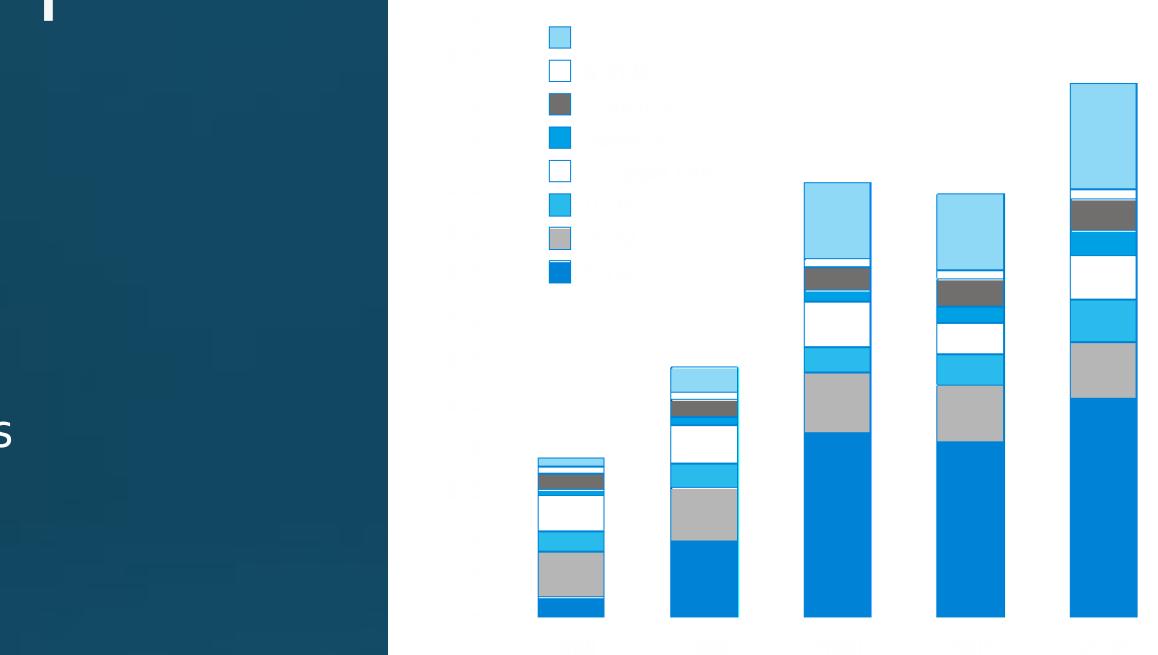
(Pipelining of RISC Instructions)



Aunque una instrucción toma cinco ciclos de reloj, una instrucción puede completarse en cada ciclo.

Crecimiento de procesadores

- Lenguaje de máquina
- Trabajaremos con la arquitectura del conjunto de instrucciones MIPS
 - Similar a otras arquitecturas desarrolladas desde los años 80.
 - Casi 100 millones de procesadores MIPS fabricados en 2002.
 - Usado por NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...
 - Tendencia actual RISC-V
 - U. Berkeley 2010



Conjunto de instrucciones MIPS (RISC)

- Las instrucciones ejecutan funciones simples.
- Mantiene la regularidad del formato:
 - cada instrucción es una palabra, contiene opcode y argumentos.
- Minimiza los accesos a la memoria:
 - siempre que sea posible, utiliza registros como argumentos.
- Tres tipos de instrucciones :
 - Register (R) – solo registros como argumentos.
 - Immediate (I) – Los argumentos son registros y números (constantes o direcciones de memoria)..
 - Jump (J) – el argumento es una dirección.



Instrucciones aritméticas MIPS

- Todas las instrucciones tienen 3 operandos.
- El orden del operando es fijo (destino primero)

Ejemplo:

código C: **a = b + c;**

código MIPS : **add a, b, c**

"El número natural de operandos para una operación como la suma es tres ... requiriendo que cada instrucción tenga exactamente tres operandos conforme a la filosofía de mantener el hardware simple"

Instrucciones aritméticas MIPS (Cont)

- Principio de diseño: la simplicidad favorece la regularidad.
 - Por supuesto esto complica algunas cosas....

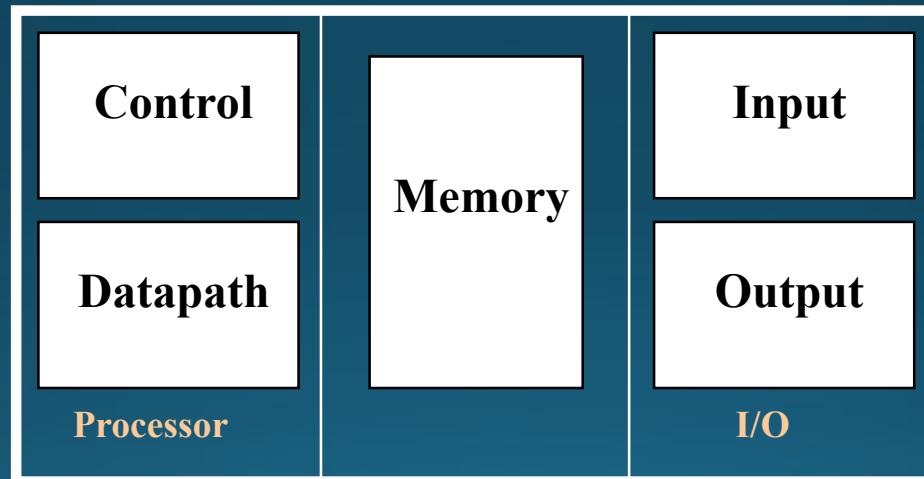
Código C : **a = b + c + d;**

Código MIPS : **add a, b, c**
 add a, a, d

- Los operandos deben ser registros
 - ¿por qué?: Cuello de botella de von Neumann.
- 32 registros provistos
- Cada registro contiene 32 bits.

Registros vs. Memoria

- Los operandos de las instrucciones aritméticas deben ser registros.
 - 32 registros provistos
- El compilador asocia las variables con los registros.
 - ¿Qué pasa con los programas con muchas variables?
 - Debe usar la memoria.

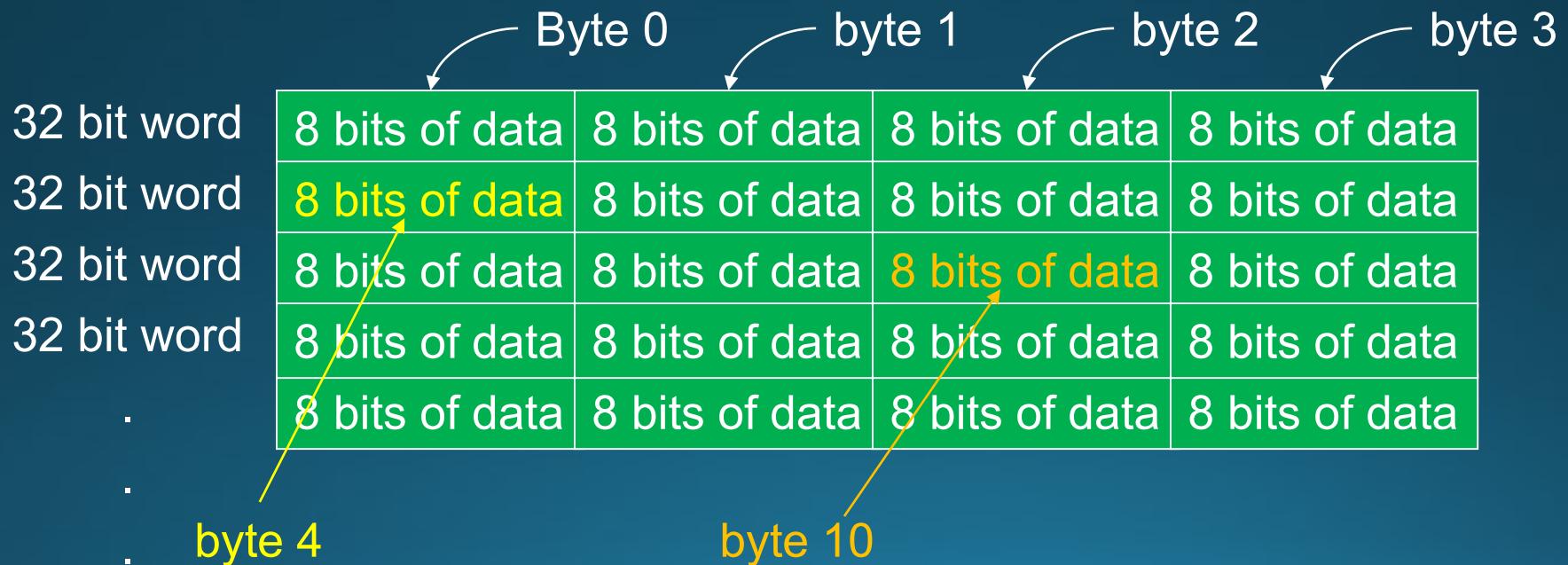


Datapath

Conjunto de componentes del procesador que realiza operaciones aritméticas (ALUs, multiplicadores, registros, bus interno).

Organización de la memoria

- Visto como una matriz grande, de una sola dimensión, con una dirección.
- Una dirección de memoria es un índice en la matriz.
- "Direccionamiento de bytes" significa que el índice apunta a un byte de memoria.



Organización de la memoria

- Los bytes están bien, pero la mayoría de los elementos de datos usan "palabras" más grandes
- Para MIPS, una palabra contiene 32 bits o 4 bytes.

word addresses

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
.	32 bits of data

...

Los registros contienen 32 bits de datos.

Usa direcciones de 32 bit

- 2^{32} bytes con direcciones de 0 a $2^{32} - 1$
- 2^{30} palabras con direcciones 0, 4, 8, ... $2^{32} - 4$
- Las palabras están alineadas

Instrucciones

- instrucciones de carga y almacenamiento.
- Ejemplo:

Código C : **A[12] = h + A[8];**

Código MIPS : **lw \$t0, 32(\$s3) #addr of A in reg s3
add \$t0, \$s2, \$t0 #h in reg s2
sw \$t0, 48(\$s3)**

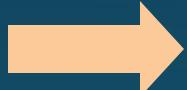
- Puede referirse a registros por nombre (ej. \$ s2, \$ t2) en lugar del número
- La instrucción de almacenamiento tiene destino al final
- ¡Recuerde que los operandos aritméticos son registros, no memoria!

No se puede escribir : **add 48(\$s3), \$s2, 32(\$s3)**

Primer ejemplo

- ¿Podemos imaginarnos el código de la subrutina?

```
swap(int v[], int k);  
{ int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```



swap:

```
sll $2, $5, 2  
add $2, $4, $2  
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)  
jr $31
```

- Inicialmente, k está en reg 5;
- la dirección base de v está en el registro 4;
- return addr está en el registro 31

Que sucede?

-
-
- **call swap**
-  *return address*
-
-

- Cuando el programa llega al "intercambio de llamadas":
 - Salta para intercambiar rutina.
 - Los registros 4 y 5 contienen los argumentos (convención de registro)
 - El registro 31 contiene la dirección de retorno (convención de registro)
 - Intercambia dos palabras en la memoria
 - Vuelve a la dirección de retorno para continuar con el resto del programa.

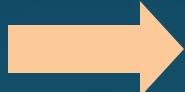
Memoria y registros



Nuestro primer ejemplo

- Ahora a descifrar el código:

```
swap(int v[], int k);  
{ int temp;  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```



swap:

```
sll $2, $5, 2  
add $2, $4, $2  
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)  
jr $31
```

Hasta ahora hemos aprendido

- MIPS
 - carga palabras pero direcciona bytes
 - aritmética solo en registros

- Instrucción

- Significado

add \$s1, \$s2, \$s3

\$s1 = \$s2 + \$s3

sub \$s1, \$s2, \$s3

\$s1 = \$s2 - \$s3

lw \$s1, 100(\$s2)

\$s1 = Memory[\$s2+100]

sw \$s1, 100(\$s2)

Memory[\$s2+100] = \$s1

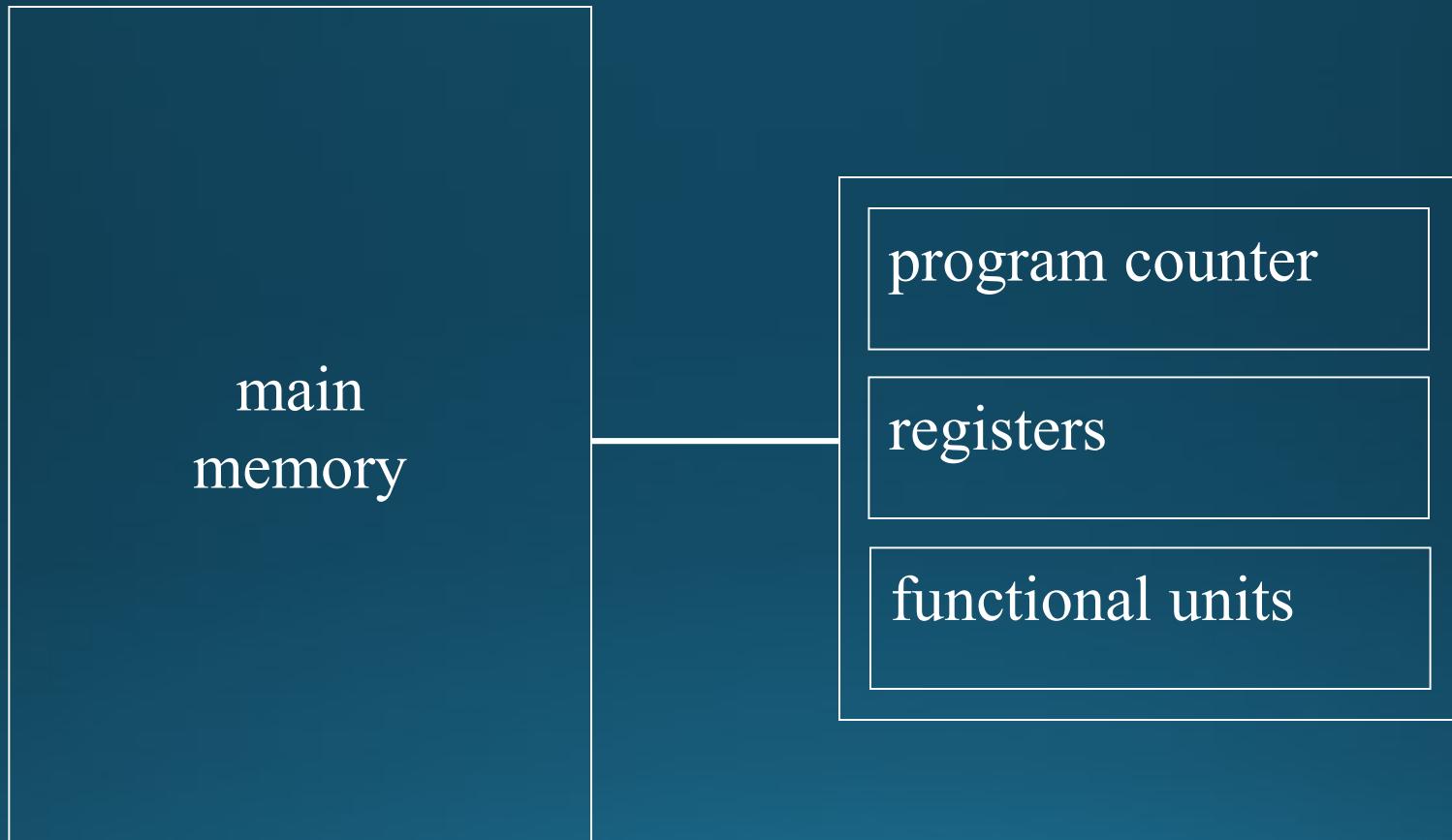
Tarea – 3 personas:

- 1. instalar emulador MIPS
 - <http://spimsimulator.sourceforge.net/>
- 2. resumir los artículos + 1 ensayo sobre la evolución de la tecnología ARM y el impacto en nuestra sociedad
 - ARM and Intel Battle over the Mobile Chip's Future + RISC & Reward
 - Future_of_microprocessors
 - Arm Ecosystem Reduces SoC Design Cost and Time to Market

Conversion a lenguaje de máquina

- //seleccionar un lenguaje
- $A = B + C;$ //por ejemplo, c
- Convertir a un lenguaje de programación mucho más primitivo.
 - Asumir que A,B,C están asociados con \$to,\$t1,\$t2
- add \$to, \$t1, \$t2 #por ejemplo, MIPS

Una organización muy sencilla.

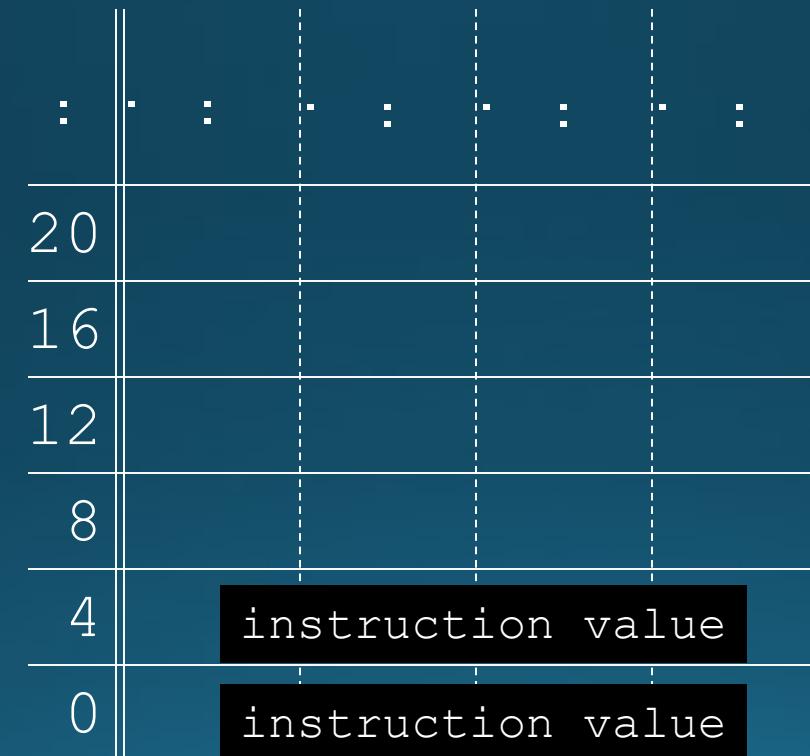


Instrucciones en memoria principal

- Las instrucciones se almacenan en la memoria principal.
 - Es simplemente una serie contigua de palabras
 - Byte direccionable: cada byte en la memoria tiene un número (una dirección)
- Program counter (PC) apunta a la siguiente instrucción
 - Todas las instrucciones MIPS tienen una longitud de 4 bytes, por lo que las direcciones de instrucciones siempre son múltiplos de 4
- Las direcciones de los programas tienen una longitud de 32 bits.
 - $2^{32} = 4,294,967,296 = 4 \text{ GigaBytes (GB)}$

Instrucciones en memoria

instruction
addresses



Ciclo Fetch/Execute

- Funcionamiento de una computadora :

```
while (procesador no esté detenido) {  
    obtener instrucciones en la ubicación de la memoria (PC)  
    PC = PC + 4 (incremento para apuntar a la siguiente instrucción)  
    ejecutar la instrucción obtenida  
}
```

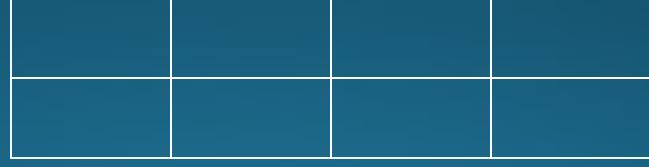
- Las instrucciones se ejecutan secuencialmente, a menos que un salto o rama cambie la PC para hacer que la próxima instrucción se obtenga de otra parte.

Algunas unidades de almacenamiento comunes

Tenga en cuenta que un byte es de 8 bits en casi todas las máquinas.

La definición de palabra es menos uniforme (4 y 8 bytes son comunes hoy en día).

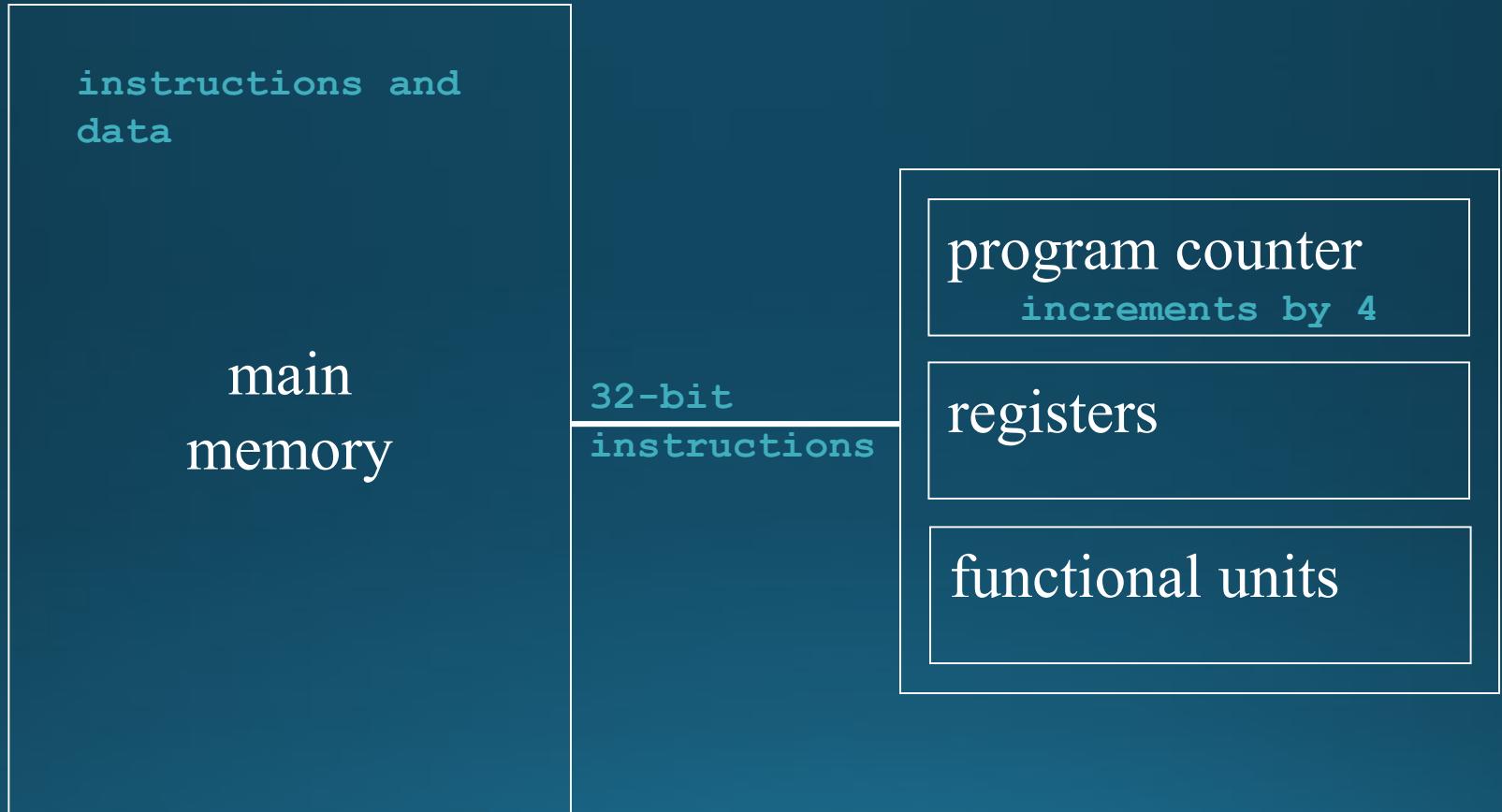
nibble = 4 bits (medio byte!)

<i>unit</i>	<i># bits</i>	
byte	8	
half-word	16	
word	32	
double word	64	

Alineación

- Un objeto en la memoria está "alineado" cuando su dirección es un múltiplo de su tamaño
- Byte: siempre alineado
- Media palabra: la dirección es múltiplo de 2
- Palabra: la dirección es múltiplo de 4
- Doble palabra: dirección es múltiplo de 8
- La alineación simplifica la carga / almacenamiento de hardware

Organización del sistema hasta ahora



Registros MIPS

- Ancho de 32 bits
 - 32 bits es 4 bytes
 - igual que una palabra en la memoria
 - Valores con signo desde -2^{31} a $+2^{31}-1$
 - Valores sin signo desde 0 a $2^{32}-1$
- Fácil de acceder y manipular
 - Acceso rápido * y * más útil que la memoria
 - 32 registros (no relacionado al ancho de 32 bits)
 - Relacionado con $2^5 = 32$
 - En el chip, acceso muy rápido.

Registros

- 32 registros de propósito general
- ¿Cuántos bits se necesitan para identificar un registro?
 - 5 bits $2^5 = 32$
- 32 registros es una selección de compromiso.
 - más requeriría más bits para identificar
 - menos sería más difícil de usar eficientemente

Registros: números y nombres

<i>number</i>	<i>name</i>	<i>usage</i>
0	zero	always returns 0
1	at	reserved for use as assembler temporary
2-3	v0 , v1	values returned by procedures
4-7	a0-a3	first few procedure arguments
8-15, 24, 25	t0-t9	temps - can use without saving
16-23	s0-s7	temps - must save before using
26, 27	k0 , k1	reserved for kernel use - may change at any time
28	gp	global pointer
29	sp	stack pointer
30	fp or s8	frame pointer
31	ra	return address from procedure

¿Cómo se utilizan los registros?

- Muchas instrucciones usan 3 registros
 - 2 registros fuente 1 registro de destino
- Ejemplo
 - **add \$t1, \$a0, \$t0**
 - **add \$t1,\$zero,\$a0**

instrucciones R-Type : 3 registros

- Add \$to, \$t₁, \$t₂
- 32 bits disponibles en la instruccion
- 15 bits para tres de los registros de 5-bit
- Los 17 bits restantes estan disponibles para especificar la instruccion
 - 6-bit op code
 - 5-bit registro ($2^5=32$)
 - 6-bit codigo de funcion

Campos R-Type

op code	source 1	source 2	dest	shamt	function
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instrucciones típicas
 - aritmeticas: **add, sub, mult, div**
 - logicas: **and, or, sll, srl**
 - Comparacion: **slt** (set on less than)
 - Saltos a traves de registros: **jr**

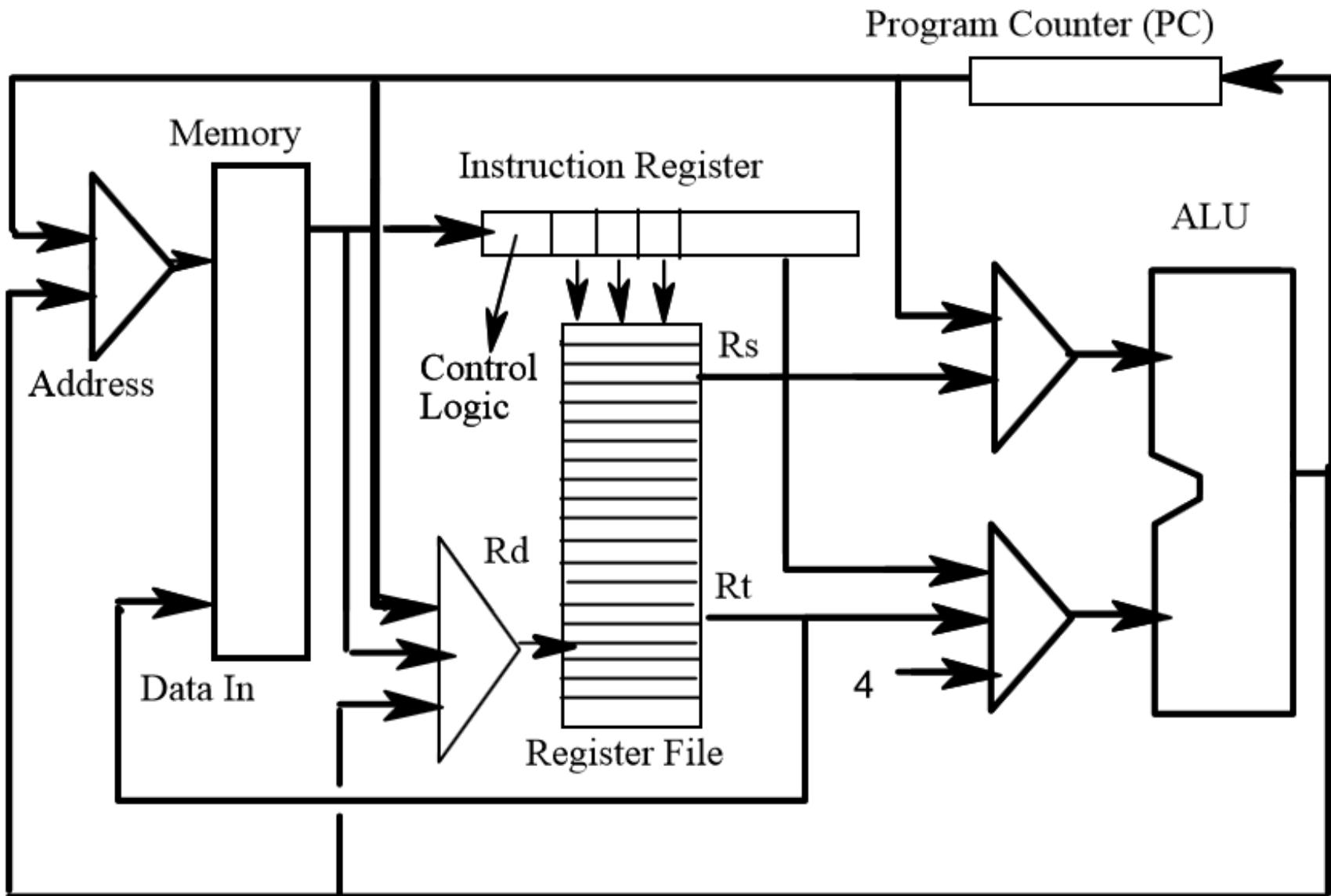


Figure 1.1 MIPS Simplified Datapath Diagram

Integer Instruction Set

Name	Syntax	Space/Time
Add:	add Rd, Rs, Rt	1/1
Add Immediate:	addi Rt, Rs, Imm	1/1
Add Immediate Unsigned:	addiu Rt, Rs, Imm	1/1
Add Unsigned:	addu Rd, Rs, Rt	1/1
And:	and Rd, Rs, Rt	1/1
And Immediate:	andi Rt, Rs, Imm	1/1
Branch if Equal:	beq Rs, Rt, Label	1/1
Branch if Greater Than or Equal to Zero:	bgez Rs, Label	1/1
Branch if Greater Than or Equal to Zero and Link:	bgezal Rs, Label	1/1
Branch if Greater Than Zero:	bgtz Rs, Label	1/1
Branch if Less Than or Equal to Zero:	blez Rs, Label	1/1
Branch if Less Than Zero and Link:	bltzal Rs, Label	1/1
Branch if Less Than Zero:	bltz Rs, Label	1/1
Branch if Not Equal:	bne Rs, Rt, Label	1/1
Divide:	div Rs, Rt	1/38
Divide Unsigned:	divu Rs, Rt	1/38
Jump:	j Label	1/1
Jump and Link:	jal Label	1/1
Jump and Link Register:	jalr Rd, Rs	1/1
Jump Register:	jr Rs	1/1
Load Byte:	lb Rt, offset(Rs)	1/1
Load Byte Unsigned:	lbu Rt, offset(Rs)	1/1
Load Halfword:	lh Rt, offset(Rs)	1/1
Load Halfword Unsigned:	lhu Rt, offset(Rs)	1/1
Load Upper Immediate:	lui Rt, Imm	1/1
Load Word:	lw Rt, offset(Rs)	1/1
Load Word Left:	lwl Rt, offset(Rs)	1/1
Load Word Right:	lwr Rt, offset(Rs)	1/1
Move From High:	mfhi Rd	1/1



Int Regs [16]

```
PC      = 0
EPC     = 0
Cause   = 0
BadVAddr = 0
Status  = 3000ff10
```

```
HI      = 0
LO      = 0
```

```
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 5
R5 [a1] = 7fffff114
R6 [a2] = 7fffff12c
R7 [a3] = 0
```

1.asm - Notepad2

File Edit View Settings ?

```
1 .text
2 .globl main
3 main:
4 ori $t0, $0, 0x2
5 ori $t1, $0, 0x3
6 addu $t2, $t0, $t1
7
8 li $v0, 10
9 syscall
```

Data Text

```
[00400000] 8fa40000 lw $4, 0($29)
[00400004] 27a50004 addiu $5, $29, 4
[00400008] 24a60004 addiu $6, $5, 4
[0040000c] 00041080 sll $2, $4, 2
[00400010] 00c23021 addu $6, $6, $2
[00400014] 0c100009 jal 0x00400024 [main]
[00400018] 00000000 nop
[0040001c] 3402000a ori $2, $0, 10
[00400020] 0000000c syscall
[00400024] 34080002 ori $8, $0, 2
[00400028] 34090003 ori $9, $0, 3
[0040002c] 01095021 addu $10, $8, $9
[00400030] 3402000a ori $2, $0, 10
[00400034] 0000000c syscall
```

```
[80000180] 0001d821 addu $27, $0, $1
[80000184] 3c019000 lui $1, -28672
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672
[80000190] ac240204 sw $4, 516($1)
[80000194] 401a6800 mfc0 $26, $13
[80000198] 001a2082 srl $4, $26, 2
[8000019c] 3084001f andi $4, $4, 31
[800001a0] 34020004 ori $2, $0, 4
[800001a4] 3c049000 lui $4, -28672 [_m1_]
[800001a8] 0000000c syscall
[800001ac] 34020001 ori $2, $0, 1
[800001b0] 001a2082 srl $4, $26, 2
```

User Text Segment [004000]

```
; 183: lw $a0 0($sp) # argc
; 184: addiu $a1 $sp 4 # argv
; 185: addiu $a2 $a1 4 # envp
; 186: sll $v0 $a0 2
; 187: addu $a2 $a2 $v0
; 188: jal main
; 189: nop
; 191: li $v0 10
; 192: syscall # syscall 10 (e)
; 4: ori $t0, $0, 0x2
; 5: ori $t1, $0, 0x3
; 6: addu $t2, $t0, $t1
; 8: li $v0, 10
; 9: syscall
```

Kernel Text Segment [80000]

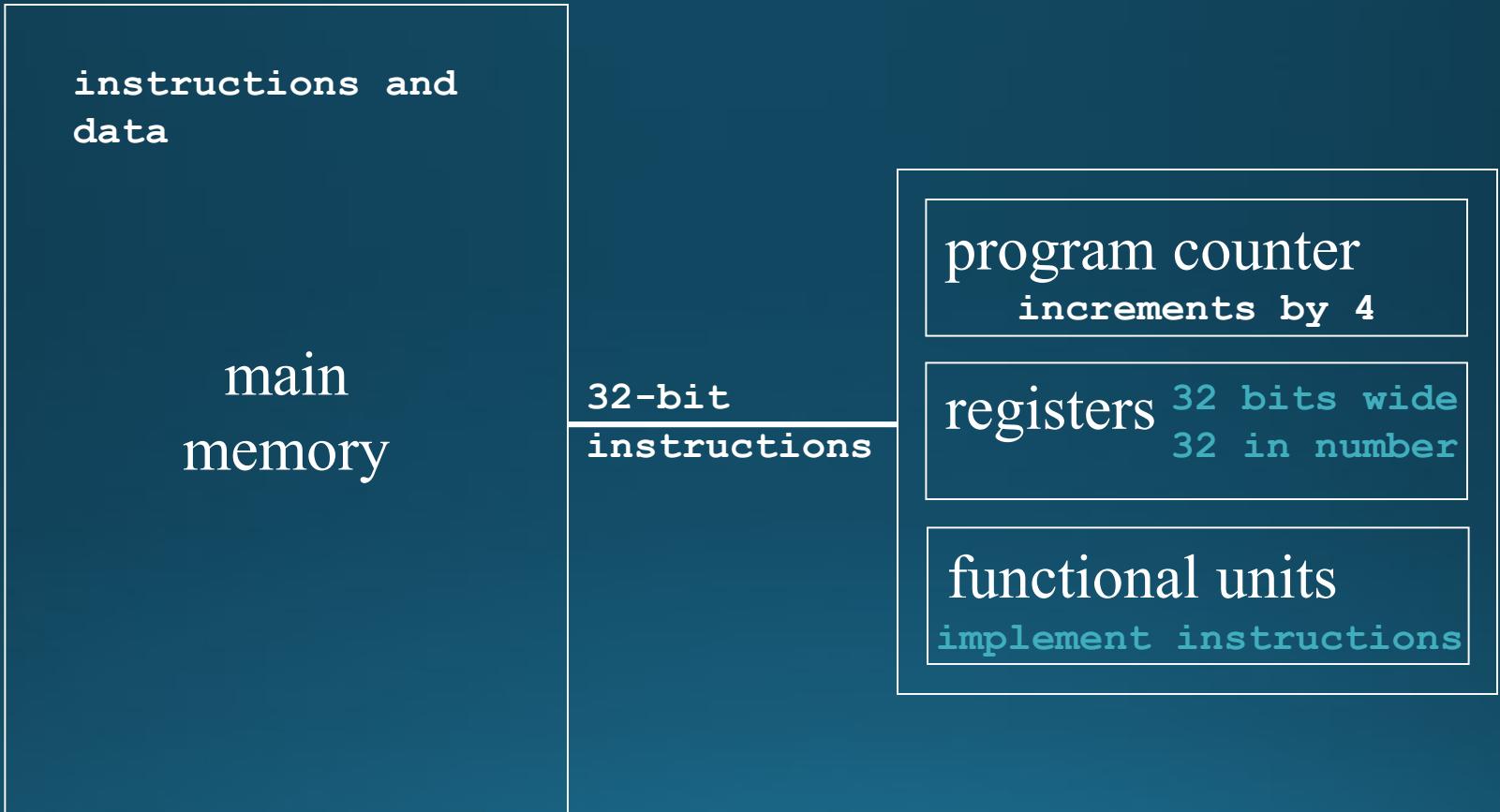
```
; 90: move $k1 $at # Save $at
; 92: sw $v0 $1 # Not re-entrant
; 93: sw $a0 $s2 # But we need
; 95: mfc0 $k0 $13 # Cause reg
; 96: srl $a0 $k0 2 # Extract
; 97: andi $a0 $a0 0x1f
; 101: li $v0 4 # syscall 4 (p)
; 102: la $a0 _m1_
; 103: syscall
; 105: li $v0 1 # syscall 1 (p)
; 106: srl $a0 $k0 2 # Extract
```

Los bits son solo bits

- Los bits significan lo que el diseñador dice que significa cuando se define la ISA
- ¿Cuántas instrucciones de 3 registros posibles hay?
 - $2^{17} = 131,072$
 - incluye todos los valores de op code, shamt, funciones
- A medida que se desarrolla el ISA a lo largo de los años, la codificación tiende a volverse menos lógica.

op code	source 1	source 2	dest	shamt	function
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Nuevamente la Organización del sistema



Transferencia de la memoria al registro

- Instrucciones Load
 - **word:** **lw rt, address**
 - **half word:** **lh rt, address**
 lhu rt, address
 - **byte:** **lb rt, address**
 lbu rt, address
- carga con signo => el bit de signo se extiende a los bits superiores del registro de destino
- carga sin signo=> 0 en los bits superiores de registro

Transferencia del registro a la memoria

- Instrucciones de almacenamiento

- **word:** **sw rt, address**

- **half word:** **sh rt, address**

- **byte:** **sb rt, address**

El término “address”

- Hay un modo de direccionamiento básico :
offset + valor del registro base
- El Offset es de 16 bits (± 32 KB)
- Cargar la palabra apuntada por so, agregar t1, almacenar

lw \$t0 , 0 (\$s0)

add \$t0 , \$t0 , \$t1

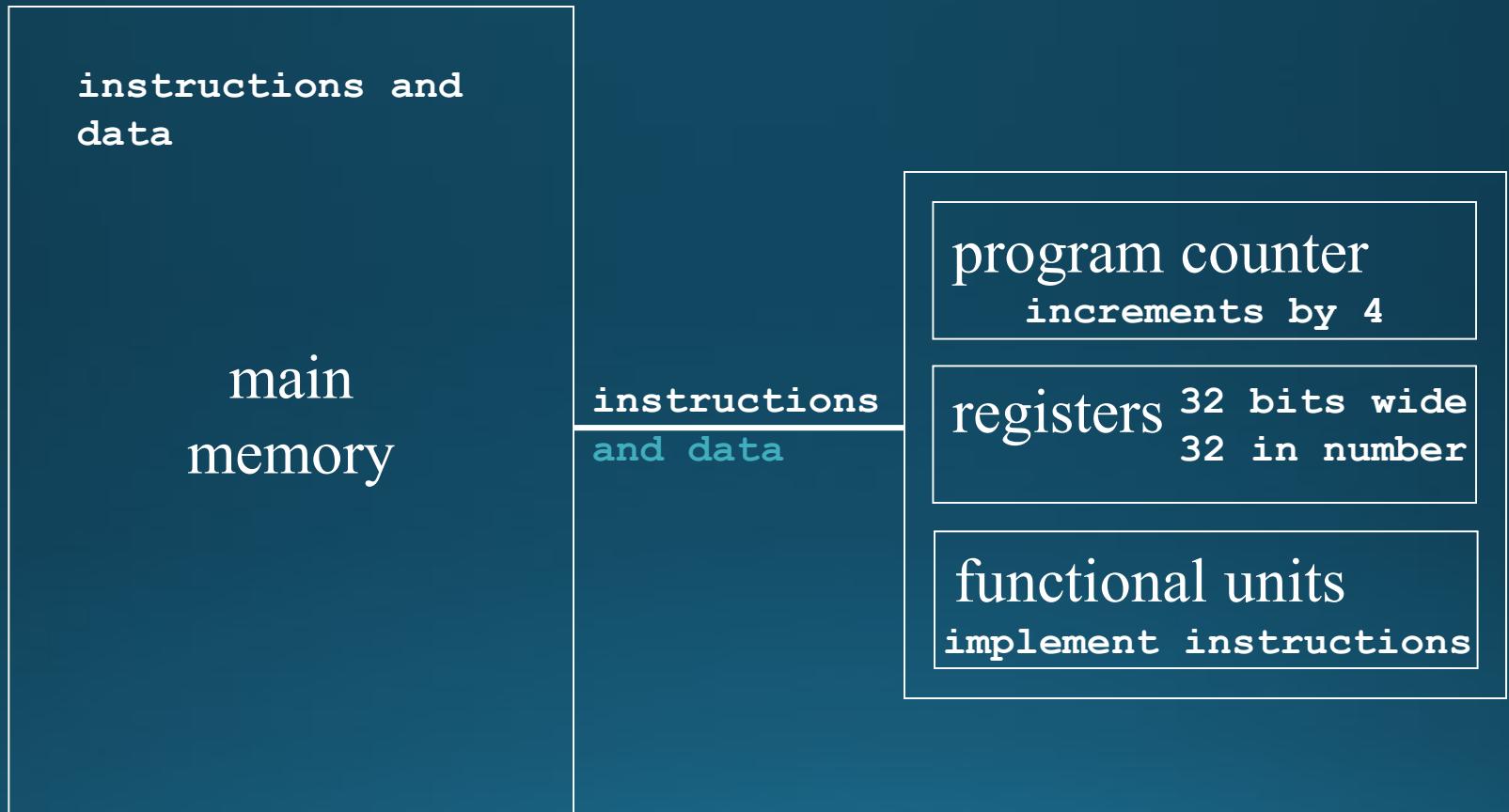
sw \$t0 , 0 (\$s0)

Campos I-Type

op code	base reg	src/dest	offset or immediate value
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>16 bits</i>

- El contenido del registro base y el valor de offset (compensación) se suman para generar la dirección para la referencia de memoria
- También puede usar los 16 bits para especificar un valor inmediato, en lugar de una dirección

Instrucciones y flujo de datos



El ojo del espectador

- Los patrones de bits no tienen un significado inherente
- Una palabra de 32 bits puede ser vista como
 - un entero con signo (± 2 mil millones)
 - un entero sin signo o un puntero de dirección (0 a 4B)
 - un número de punto flotante de precisión simple
 - cuatro caracteres de 1 byte
 - una instrucción
 - Cuatro caracteres ASCII

R-Type vs I-Type

op code	source 1	source 2	dest	shamt	function
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>6 bits</i>

op code	base reg	src/dest	offset or immediate value
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>16 bits</i>

R-Type vs I-Type

op code	source 1	source 2	dest	shamt	function
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>6 bits</i>

2 vs 3 registros, no existe shift, solo opcode

op code	base reg	src/dest	offset or immediate value
<i>6 bits</i>	<i>5 bits</i>	<i>5 bits</i>	<i>16 bits</i>

¿Qué pasa si probamos un offset o un relleno inmediato en shamt?

¿Qué pasa si intentamos indexar una matriz (inmediata) usando una función?

¿Si intentamos saltar o ramificar usando la función?

¿Qué pasa con la compensación de tipo I?

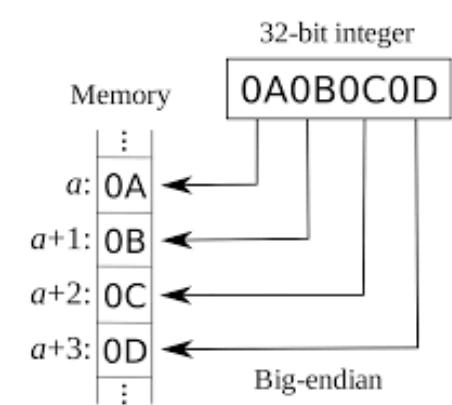
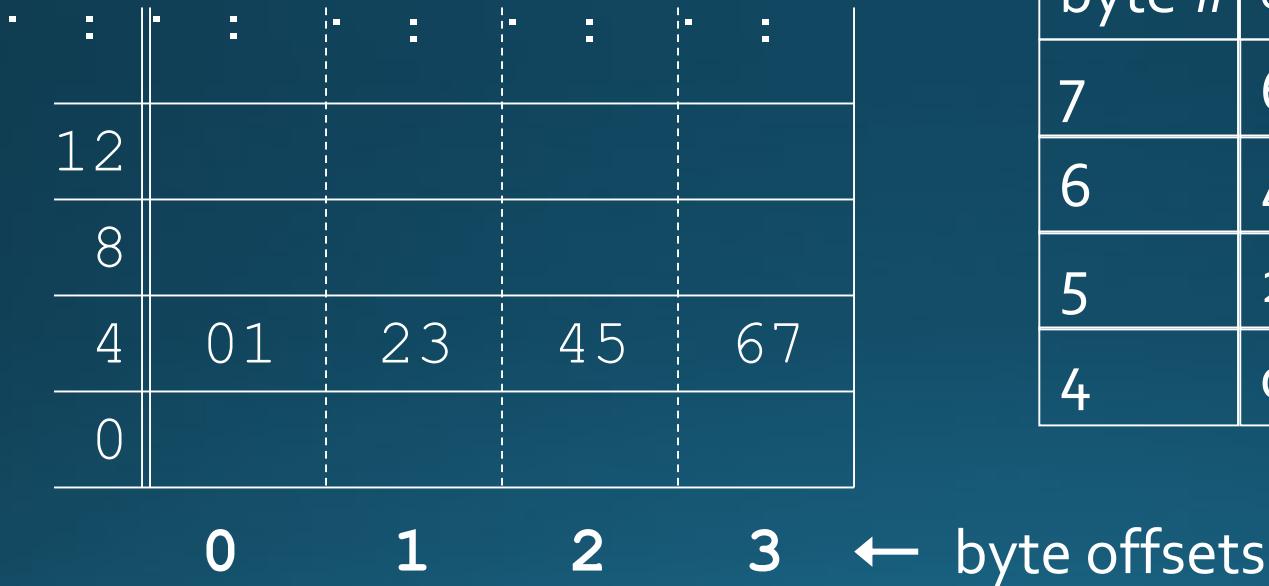
¿Qué pasaría si tuviéramos un nuevo tipo de instrucción para Jump, y el primer bit del código de operación lo indica, el resto es la dirección de salto (o offset), a qué distancia podemos saltar?

Big-endian, little-endian

- Una palabra de 32 bits en la memoria tiene 4 bytes de longitud
- pero ¿qué byte es qué dirección?
- Considere el numero de 32-bit `ox01234567`
 - Cuatro bytes: `01, 23, 45, 67`
 - Los bits más significativos son `ox01`
 - bits menos significativos son `ox67`

Datos en la memoria- big endian

Big endian (Motorola) - los bits **más significativos** están en el byte o (CERO) de la palabra
Representación más natural

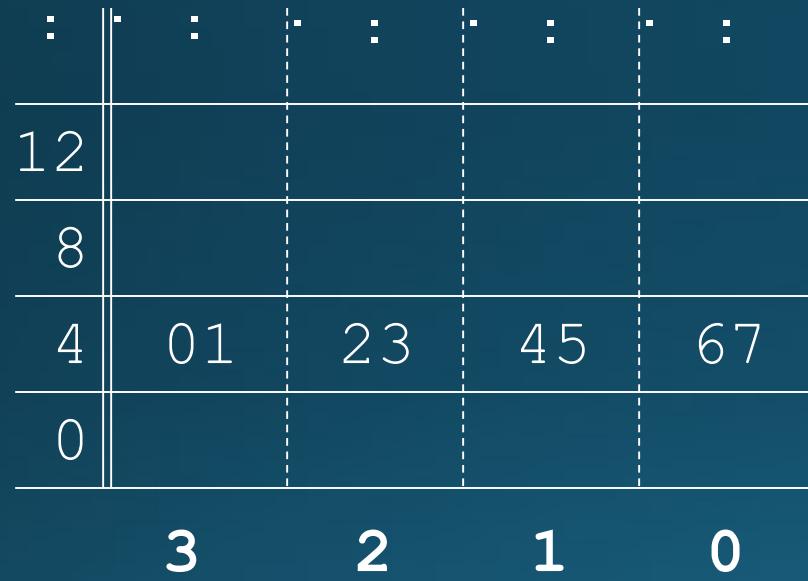


Datos en la memoria- little endian

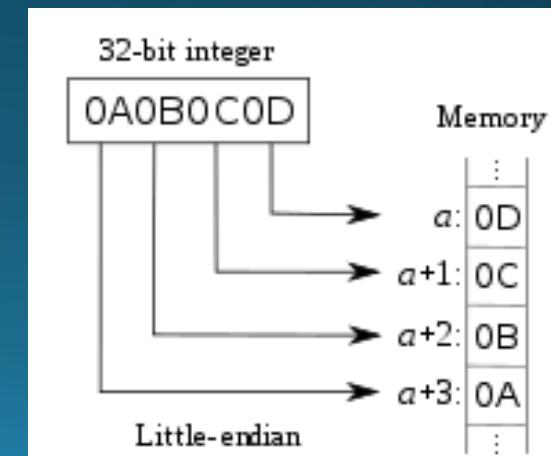
Little endian (Intel)- los bits **menos significativos** están en el byte o (CERO) de la palabra

29-04-2019

- más intuitivo el acceso a datos



byte #	contents
7	01
6	23
5	45
4	67



Ejemplo Immediate

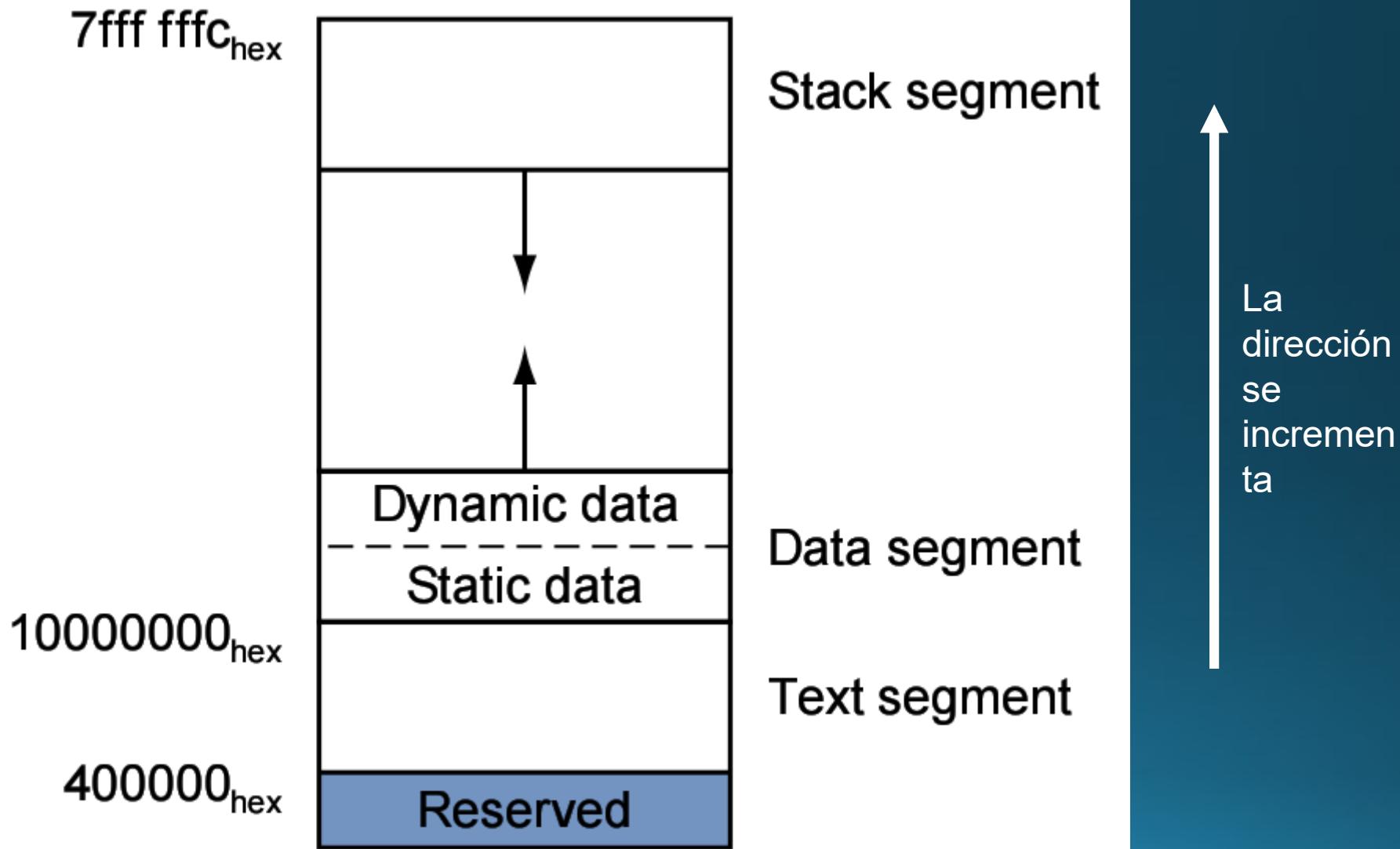
```
.text
.globl main
main:
    li      $t0, 0x2          # $8 ← 0x2
    li      $t1, 0x3          # $9 ← 0x3
    addu   $t2, $t0, $t1      # $10 ← ADD($t0, $t1)
```

\$0	zero	Permanently 0	\$24, \$25	\$t8, \$t9	Temporary
\$2, \$3	\$v0, \$v1	Value returned by a subroutine			
\$4-\$7	\$a0-\$a3	Subroutine Arguments	\$29	\$sp	Stack Pointer
\$8-\$15	\$t0-\$t7	Temporary	\$30	\$fp	Frame Pointer
			\$31	\$ra	Return Address

Ejemplo Or immediate

```
.text
.globl main
main:
    ori    $t0, $0, 0x2  # $8 ← OR($0, 0x2)
    ori    $t1, $0, 0x3  # $9 ← OR($0, 0x3)
    addu   $t2, $t0, $t1 # $10 ← ADD($t0, $t1)
```

Distribución de la memoria



Cómo usar la memoria

1. Cargar desde la memoria al registro

lw, lb, ld, ... (lw \$to, address)

2. Realizar el cálculo en los registros

add, ori, beq, jal,... (add \$t2, \$to, \$t1)

3. Almacenar desde el registro a la memoria

sw, sb, sd,... (sw \$t2, address)

Modos de direccionamiento

Format	Address Computation
(register)	contents of register
imm	immediate
<u>imm (register)</u>	<u>immediate + contents of register</u>
symbol	address of symbol
symbol \pm imm	address of symbol + or - immediate
symbol \pm imm (register)	address of symbol + or - (immediate + contents of register)

Modos de direccionamiento

Carga desde la memoria a \$to: lw \$to, dirección?

Imm+Register: (Único modo directo)

```
la $t1, label # carga la dirección de la etiqueta a $t1  
lw $to, 2($t1) # dirección: dirección de la etiqueta + 2
```

Immediate:

```
lw $to, ox000AE430 # dirección: dirección ox000AE430
```

Register:

```
la $t1, label # carga la dirección de la etiqueta a $t1  
lw $to, $t1 # dirección: dirección en $t1
```

Symbol:

```
lw $to, label # dirección: dirección de la etiqueta
```

Symbol±Imm:

```
lw $to, label+2 # dirección: dirección de la etiqueta + 2
```

Symbol±Imm+Register:

```
lw $to, label+2($t1) # dirección: dir de la etiqueta + 2 + $t1
```

```
.data
n:    .word  ox2
m:    .word  ox3
r:    .space 4

.text
.globl main
main:
la    $t5, n      # carga dirección de n a $t5
lw    $to, 0($t5)  # carga n a $to
la    $t5, m      # carga dirección de m a $t5
lw    $t1, 0($t5)  # carga m a $t1
addu $t2, $to, $t1  # $10 ← ADD($8, $9)
la    $t5, r      # carga dirección de r a $t5
sw    $t2, 0($t5)  # almacena $10 en r
```

```
.data  
n:      .word  ox2  
m:      .word  ox3  
r:      .space 4  
  
.text  
.globl main  
main:  
lw    $to, n      # carga n a $to  
lw    $t1, m      # carga m a $t1  
addu $t2, $to, $t1    # $10 ← ADD($8, $9)  
sw    $t2, r      # almacena $10 en r
```

Llamadas de sistema

Servicios del sistema

Servicio	Código de llamada	Argumentos	Resultado
print_int	1	\$a0 = entero	
print_float	2	\$f12 = real (32 bits)	
print_double	3	\$f12 = real (64 bits)	
print_string	4	\$a0 = cadena	
read_int	5		Entero (en \$v0)
read_float	6		Real 32 bits (en \$f0)
read_double	7		Real 64 bits (en \$f0)
read_string	8	\$a0=buffer, \$a1 = longitud	
sbrk	9	\$a0 = cantidad	Dirección (en \$v0)
exit	10		

Llamadas de sistema – print_str

```
.data  
str: .asciiz "Hola mundo"  
  
.text  
.globl main  
main:  
    li $v0, 4          # código de print_str  
    la $a0, str        # argumento  
    syscall            # ejecuta print_str
```

Llamadas de sistema- read_int

```
.data
num: .space 4

.text
.globl main
main:
    li $v0, 5          # código de read_int
    syscall            # ejecuta read_int
                      # valor de retorno se guarda en $v0
    la $t0, num         # carga la dirección de num en $t0
    sw $v0, 0($t0)      # guarda el número en num
```

Derivaciones

`x ← read_int`

`y ← read_int`

`if x == y`

`then print “Igual”`

`else print “No igual”`

Derivaciones

```
.text
.globl main
main:
    li $v0, 5
    syscall
    move $t0, $v0
    li $v0, 5
    syscall
    move $t1, $v0
    seq $t2, $t0, $t1
    beq $t2, 1, printEq
    beq $t2, $0, printNe
    printEq:
        la $ao, strEq
        j print
    printNe:
        la $ao, strNe
        j print
    print:
        li $v0, 4
        syscall
.data
strEq: .asciiz "Igual"
strNe: .asciiz "No igual"
```

Bucles

x ← read_int

counter ← 0

total ← 0

do

 counter ← counter + 1

 total ← total + counter

until counter == x

print total

Bucles

```
.text
.globl main
main:
    li $vo, 5
    syscall

    move $to, $vo

# $to is the original value

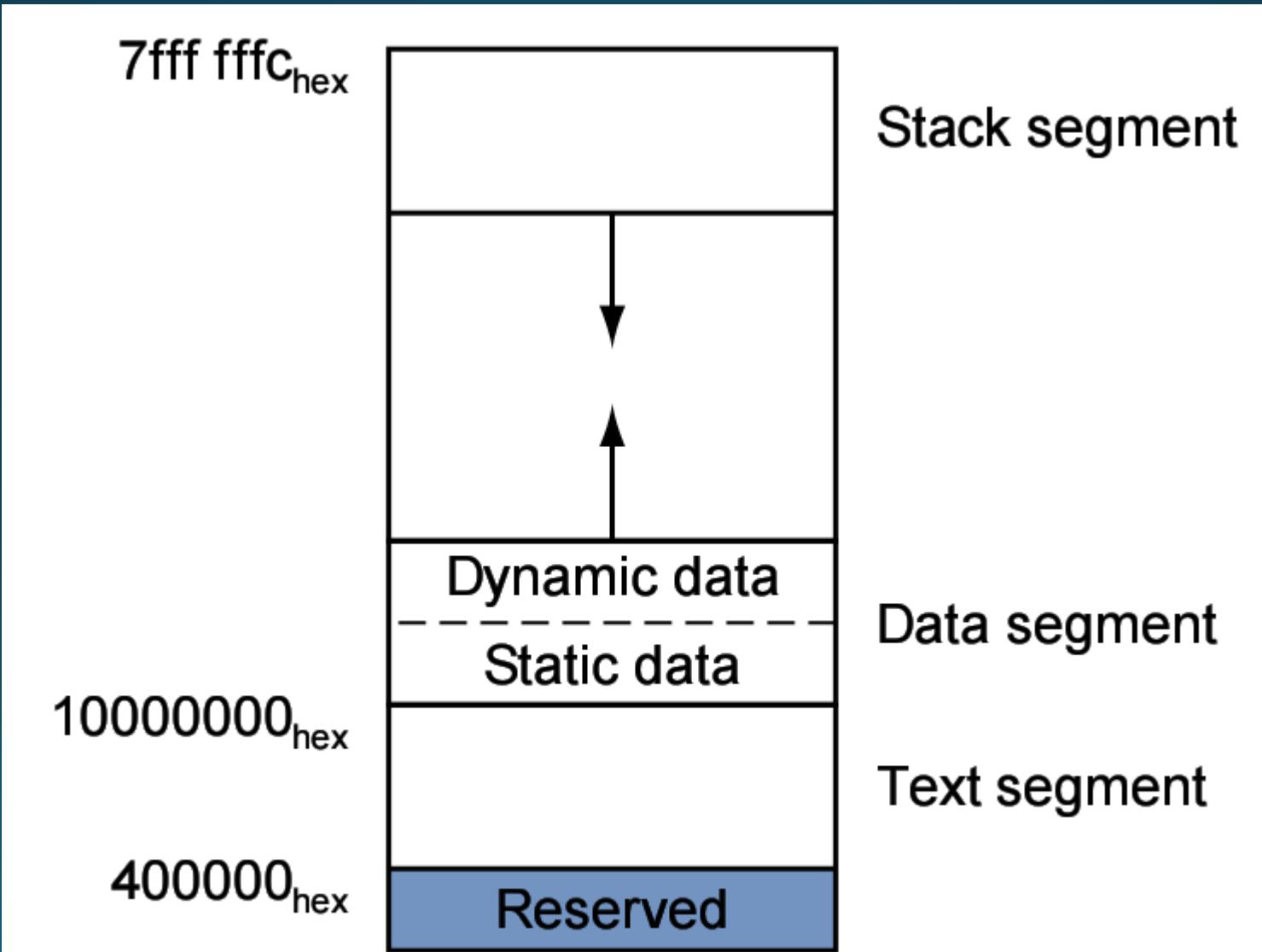
    li    $t1, 0 # counter
    li    $t2, 0 # sum
```

```
loop:
    addi $t1, $t1, 1
    add $t2, $t2, $t1

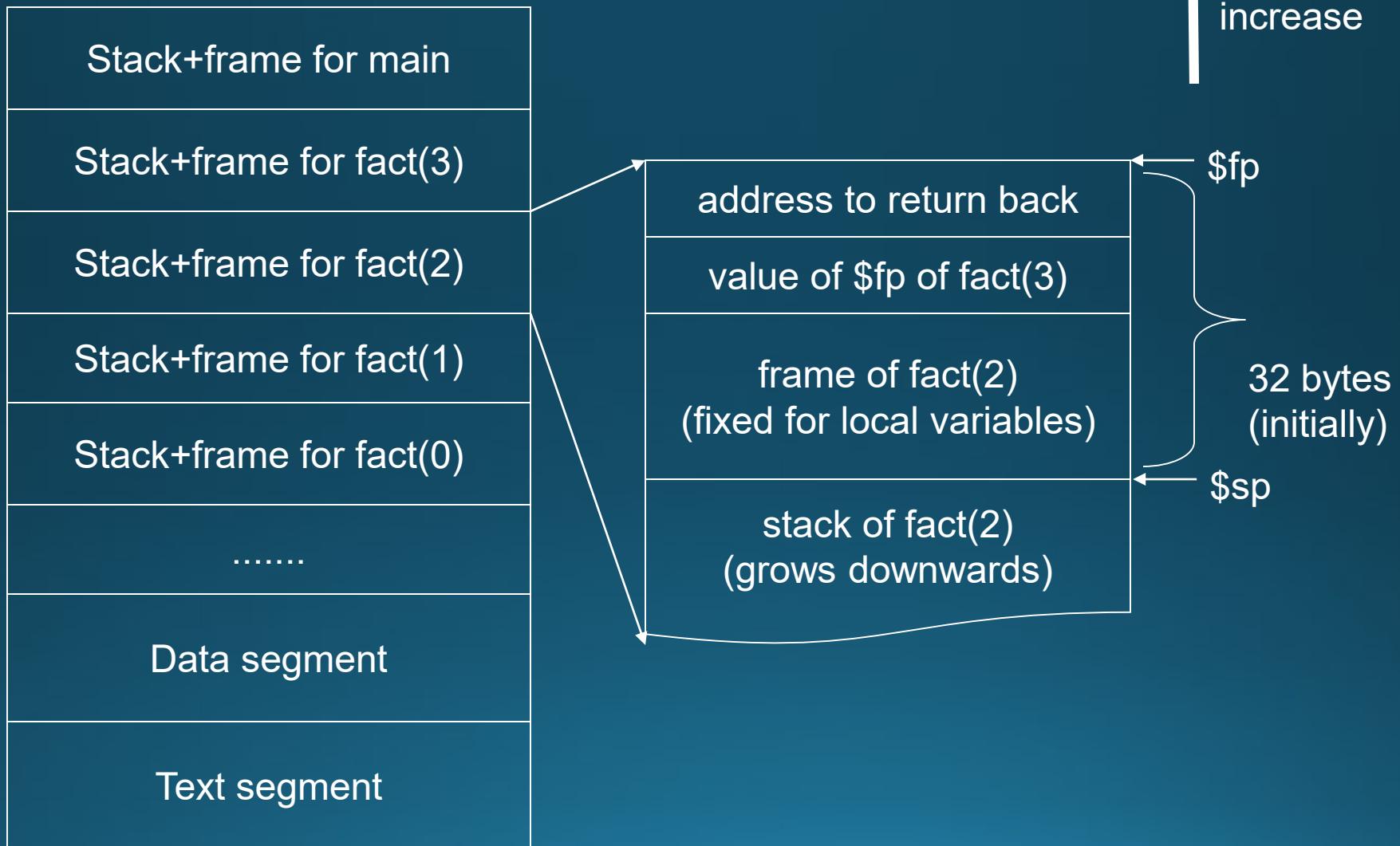
    beq $to, $t1, done
    j loop
```

```
done:
    li $vo, 1 # print_int
    move $ao, $t2
    syscall
```

Funciones



Factorial



Practicar

- Escribir un programa factorial sin funciones
- Usar solo derivaciones y bucles