computer Architectur

Arquitectura de Computadores

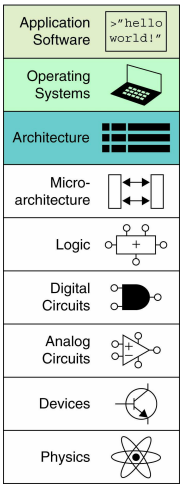
Lenguaje Ensamblador

Basado en texto: "***Digital Design and Computer Architecture*, 2nd Edition**", David Money Harris and Sarah L. Harris

computer Architectur

Tópicos

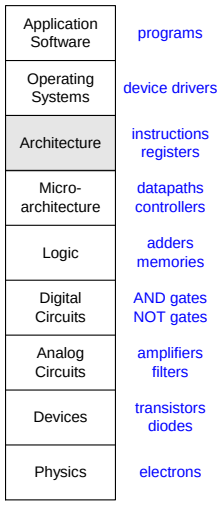
- **Introducción**
- **Lenguaje Ensamblador**
- **Lenguaje de Maquina**
- **Programación**
- **Modos de Direccionamiento**
- **Luz, Cámara y Acción: Compilar, Ensamblar, & Cargar**
- **Misceláneo**



computer Architectur

Introducción

- Subamos algunos niveles de abstracción
- **Arquitectura:** La vista del programador del computador
  - Esta definida por la ubicación de los operandos & instrucciones
- **Micro-arquitectura:** como implementar una arquitectura en hardware (se cubrirá en el próximo capitulo)



computer Architectur

Lenguaje Ensamblador

- **Instrucciones:** comandos de un lenguaje de un computador
  - **Lenguaje Ensamblador (Assembly):** instrucciones definidas en un formato comprensible por un ser humano
  - **Lenguaje de maquina:** formato comprensible para un computador (1's y 0's)
- Arquitectura **MIPS**:
  - Desarrollado por John Hennessy y sus colegas en Stanford en los 1980's.
  - Ha sido usado en sistemas comerciales, que incluyen a Silicon Graphics, Nintendo, y Cisco

John Hennessy

- Rector de Stanford University
- Profesor de Ingeniería Eléctrica y Ciencias de la Computación en Stanford desde 1977
- Có-invento el set reducido de instrucciones para un computador (RISC) con David Patterson
- Desarrollo la arquitectura MIPS en Stanford en 1984 y cofundo la empresa MIPS Computer Systems
- Al año 2004, mas de 300 millones de microprocesadores MIPS han sido vendidos

Principios de Diseño de una Arquitectura

Los principios de diseño definidos por Hennessy y Patterson son:

1. **La simplicidad favorece la regularidad**
2. **Haz que el caso común opere con rapidez**
3. **Mientras mas pequeño es mas rápido**
4. **Un buen diseño demanda buenos compromisos**

Instrucciones: Suma

Código C
a = b + c;

Código Ensamblador MIPS
add a, b, c

- **add:** mnemotecnia que indica la operación a realizar
- **b, c:** operandos de entrada (a los cuales se realizara la operación)
- **a:** operando destino (al cual se le escribirá el resultado)

Instrucciones: Resta

- Similar a la suma – solo cambio mnemotécnico

Código C
a = b - c;

Código Ensamblador MIPS
sub a, b, c

- **sub:** mnemotecnia
- **b, c:** operandos de origen
- **a:** operando destino

Principio de Diseño1

La simplicidad favorece la regularidad

- Formato de instrucción consistente
- El mismo numero de operandos (dos de origen y uno de destino)
- Facilita la codificación y el manejo en hardware

Múltiples Instrucciones

- Un código complejo es manejado por múltiples instrucciones MIPS

Código C

```
a = b + c - d;
```

Codigo Ensamblador MIPS

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Principio de Diseño 2

Haz que el caso común opere con rapidez

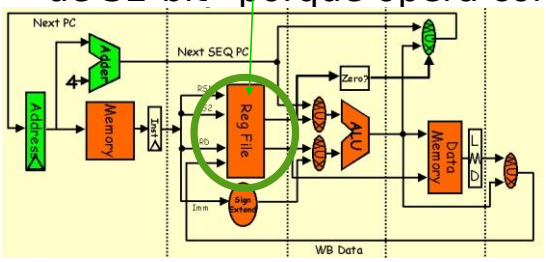
- MIPS incluye solo instrucciones simples y que son comúnmente usadas
- El hardware para decodificar y ejecutar las instrucciones debe ser simple, pequeño y rápido
- Las instrucciones mas complejas (que son las menos comunes) son realizadas con múltiples instrucciones simples
- MIPS es un ***reduced instruction set computer* (RISC)**, con un pequeño numero de instrucciones simples
- Otras arquitecturas, tales como x86 de Intel, son ***complex instruction set computers* (CISC)**

Operandos

- La ubicación de un operando: ubicación física en el computador
  - Registros
  - Memoria
  - Constantes (también llamadas *inmediatos*)

Operandos: Registros

- MIPS tiene 32 registros de 32-bit
- Los registros son mas rápidos que la memoria
- Se dice que MIPS es una “arquitectura de 32-bit” porque opera con datos de



Microprocesador monociclo MIPS

Principio de Diseño 3

Mientras mas pequeño es mas rápido

- MIPS incluye solo un pequeño numero de registros

Conjunto de Registros de MIPS

Nombre	Numero del Registro	Uso
$.	0	El valor constante 0
$ at	1	assembler temporal
$ v. -$ v/	2-3	Valores de retorno función
$ a. -$ a1	4-7	Argumentos función
$ t. -$ t5	8-15	temporales
$ s. -$ s5	16-23	Variables guardadas
$ t6 -$ t7	24-25	mas temporales
$ k. -$ k/	26-27	Temporales del SO
$ gp	28	Puntero global
$ sp	29	stack pointer
$ fp	30	frame pointer
$ ra	31	Dirección retorno función

Operandos: Registros

- Registros:
  - $ antes del nombre
  - Ejemplo: $0, “registro cero”, “dolar cero”
- Los registros son usados para propósitos específicos:
  - $0 siempre guarda el valor constante 0.
  - Los *registros de guarda*, $ s. -$ s5, se usan para almacenar variables
  - Los *registros temporales*, **$t0 - $t9**, se usan para guardar valores intermedios durante un gran computo
  - Discutiremos los otros después

Instrucciones con Registros

- Revisemos la instrucción add

Código C

```
a = b + c
```

Código ensamblador MIPS

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```


Operandos: Memoria

- Tenemos tan solo 32 registros para almacenar muchos datos
- Almacene la mayoría de los datos en memoria
- Si bien la memoria es grande, pero es lenta
- Mantenga las variables de mayor uso en registros

Memoria direccionable por palabras

- Cada palabra de datos de 32 bits tiene una dirección única

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

Nota: MIPS usa memoria direccionable por bytes. Tocaremos este punto próximamente

Leyendo una Memoria Direccionable por Palabras

- A la lectura de memoria se le llama ***load***
- **Mnemotecnia:** *load word* (lw)
- **Formato:**
  lw $s0, 5($t1)
- **Calculo dirección:**
  - sume a la dirección *base* ($t1) la compensación (*offset)* (5)
  - dirección = ($t1 + 5)
- **Resultado:**
  - $s0 guarda el valor leído de la dirección ($t1 + 5)

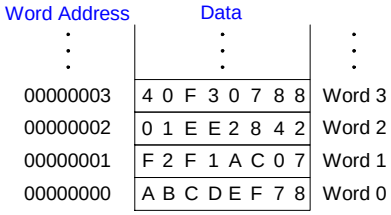
Cualquier registro podría ser usado para la dirección base

Leyendo una Memoria Direccionable por Palabra

- **Ejemplo:** Lea una palabra de datos de la dirección de memoria 1 y guárdelo en el registro $s3
  - dirección = **($0** + 1) = 1
  - $s3 = 0xF2F1AC07 luego de cargar/leer

Código Ensamblador

```
lw $s3, 1($0)  # lee de dirección 1 una palabra que se guarda en
    $s3
```



Escribiendo una Memoria Direccionable por Palabra

- Las escrituras de memoria se llaman ***store***
- **Mnemotecnia:** *store word* (sw)

Escribiendo una Memoria Direccionable por Palabra

- **Ejemplo:** Escriba (guarde) el valor de $t4 en la direccion de memoria 7
  - Sume a la dirección base ($0) el offset (0x7) dirección: ($0 + 0x7) = 7

El offset puede ser escrito en decimal (por defecto) o hexadecimal

Codigo Ensamblador

```
sw $t4, 0x7($0)  # escriba la palabra en $t4
                 # a la dirección de memoria 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

Memoria Direccionable por Byte

- Cada byte de datos tiene una direccion única
- Load/store palabras o bytes: load byte (lb) y store byte (sb)
- Palabra de 32-bit = 4 bytes, luego las direcciones de palabras se incrementan en 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

Leyendo una Memoria Direccionable por Byte

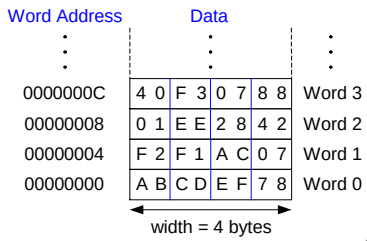
- La dirección de palabra en memoria debe ser ahora multiplicada por 4.
- Ejemplo,
  - La dirección de memoria de la palabra 2 es 2 × 4 = 8
  - La dirección de memoria de la palabra 10 es 10 × 4 = 40 (0x28)
- **MIPS es direccionable por BYTE, no es direccionable por palabra (word)**

Leyendo una Memoria Direccionable por Byte

- **Ejemplo:** Cargue la palabra de datos ubicada en la dirección de memoria 4 en $s3.
- $s3 guarda el valor 0xF2F1AC07 al momento de la carga

Codigo Ensamblador MIPS

lw $s3, 4($0) # lee palabra de la dirección 4 y la escribe en $s3

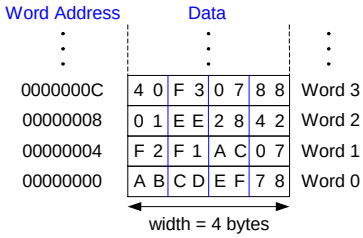


Escribiendo una Memoria Direccionable por Byte

- **Ejemplo:** guarde el valor almacenado en $t7 en la dirección de memoria 0x2C (44)

Código Ensamblador MIPS

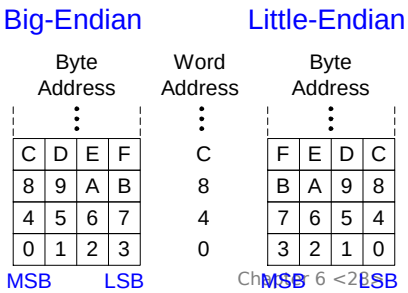
```
sw $t7, 44($0)  # escribe $t7 en la dirección 44
```



computer Architecture

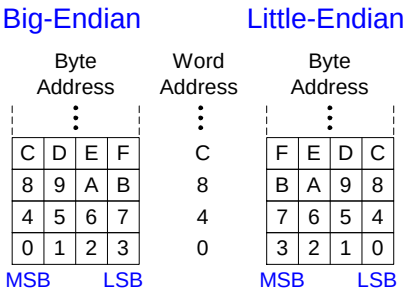
Memoria Big-Endian & Little-Endian

- ¿Como enumerar los bytes de una palabra?
- **Little-endian:** la enumeración de los bytes parte por el final de la fila o extremo derecho (menos significativo)
- **Big-endian:** la enumeración de los byte parte por el comienzo de la fila o extremo izquierdo (el mas significativo)
- **La dirección de una palabra** es la **misma** para big-endian y little-endian



Memoria Big-Endian & Little-Endian

- En “Los viajes de Gulliver” de Jonathan Swift: Los “Little-Endians” rompieron sus huevos en el extremo menor del huevo y los Big-Endians rompieron sus huevos en extremo mayor
- Realmente no importa cual tipo de direccionamiento se use – ¡excepto cuando dos sistemas requieren compartir datos!



Ejemplo Big-Endian & Little-Endian

- Suponga que $t0 inicialmente contiene el valor 0x23456789
- Después de ejecutar el siguiente código en un sistema big-endian , ¿que valor tiene $s0?
- Y ¿cual tendría en un sistema little-endian?

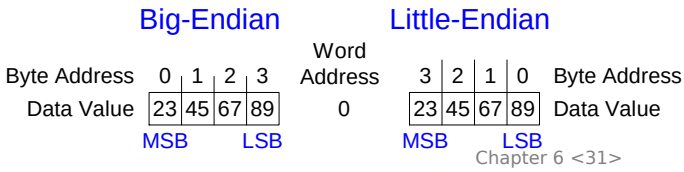
```
sw $t0, 0($0)
lb $s0, 1($0)
```

Ejemplo Big-Endian & Little-Endian

- Suponga que $t0 inicialmente contiene el valor 0x23456789
- Después de ejecutar el siguiente código en un sistema big-endian , ¿que valor tiene $s0?
- Y ¿cual tendría en un sistema little-endian?

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

- Big-endian: 0x00000045
- Little-endian: 0x00000067



Principio de Diseño 4

Un buen diseño demanda buenos compromisos

- Formatos de múltiples instrucciones permite flexibilidad
  - add, sub: use 3 operandos de registros
  - lw, sw: use 2 operandos de registros y una constante
- El numero de formatos de instrucciones se mantiene bajo
  - Para adherir a los principios de diseño 1 y 3 (simplicidad favorece regularidad y mientras mas pequeño es mas rápido).

Operandos: Constantes/Inmediatos

- lw y sw usan constantes o *inmediatos*
- Están *inmediatamente* disponibles en la instrucción
- Números en complemento dos de 16 bits
- addi: suma inmediata
- ¿Es restar inmediatamente (subi) necesario?

Código C

```
a = a + 4;
b = a – 12;
```

Código Ensamblador MIPS

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```


Lenguaje de Maquina

- Representación binaria de las instrucciones
- Un computador solo entiende 1’s y 0’s
- Instrucciones de 32-bit
  - Simplicidad favorece regularidad: instrucciones & datos de 32-bit data
- 3 formatos de instrucciones:
  - **Tipo R:** operandos de registros
  - **Tipo I:** operando inmediato
  - **Tipo J:** para saltos (lo discutiremos pronto)

Tipo R

- *Tipo-Registros*
- 3 operandos de registros:
  - rs, rt: registros de origen
  - rd: registros de destino
- Otros campos:
  - op: el código de operación o *opcode* (0 para instructiones tipo R)
  - funct: la *función*
    con opcode, le dice al computador que operación realizar
  - shamt: la *cantidad de desplazamiento* para las instrucciones shift, si no hay se coloca 0

R-Type

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

Ejemplo Tipo R

Assembly Code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

Field Values

op	rs	rt	rd	shamt	funct
0	17	18	16	0	32
0	11	13	8	0	34
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10001	10010	10000	00000	100000	(0x02328020)
000000	01011	01101	01000	00000	100010	(0x016D4022)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

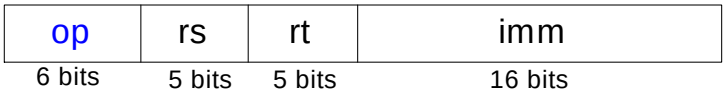
Note el orden de los registros en el código ensamblador:

add rd, rs, rt

Tipo I

- *Tipo Inmediato*
- 3 operandos:
  - rs, rt: operandos de registros
  - imm: inmediato de 16-bit en complemento dos
- Otros campos:
  - op: el opcode
  - La simplicidad favorece la regularidad: todas las instrucciones tienen opcode
  - Una operación esta completamente definida por su opcode

I-Type



Ejemplos Tipo I

Assembly Code	op	rs	rt	imm
`addi $s0, $s1, 5`	8	17	16	5
`addi $t0, $s3, -12`	8	19	8	-12
`lw $t2, 32($0)`	35	0	10	32
`sw $s1, 4($t1)`	43	9	17	4
	6 bits	5 bits	5 bits	16 bits

Field Values

Note la diferencia en el orden de los registros en el assembler y en los códigos de maquina:

addi rt, rs, imm

lw rt, imm(rs)

sw rt, imm(rs)

Machine Code

op	rs	rt	imm	
001000	10001	10000	0000 0000 0000 0101	(0x22300005)
001000	10011	01000	1111 1111 1111 0100	(0x2268FFF4)
100011	00000	01010	0000 0000 0010 0000	(0x8C0A0020)
101011	01001	10001	0000 0000 0000 0100	(0xAD310004)
6 bits	5 bits	5 bits	16 bits	

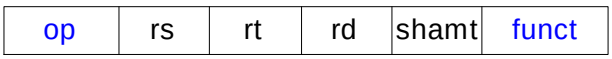
Lenguaje de Maquina: Tipo J

- *Tipo Salto (Jump)*
- El operando es una direccion de 26-bit (addr)
- Se usa para instrucciones “jump” (j)

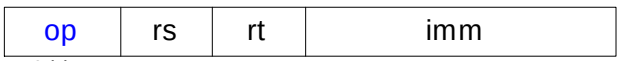
J-Type

Revisión: Formatos de Instrucción

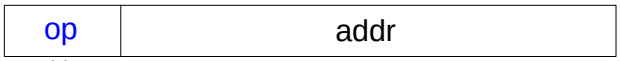
R-Type



I-Type



J-Type



computer Architectur

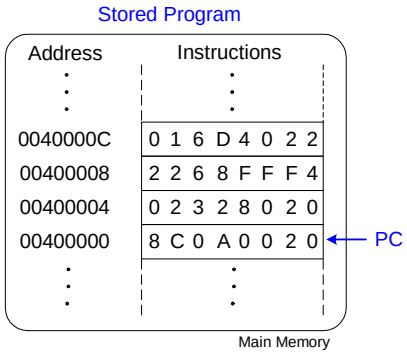
Potencia de un Programa Almacenado

- Las instrucciones & datos de 32-bit instrucciones & son almacenados en memoria
- La secuencia de instrucciones: es la única diferencia entre dos aplicaciones
- Para ejecutar un programa:
  - No se requiere de un re-cableado
  - Simplemente guarde el nuevo programa en memoria
- Ejecución de un programa:
  - El Procesador *busca* (fetch/lee) las instrucciones de la memoria en secuencia

computer Architectur

Programa Almacenado

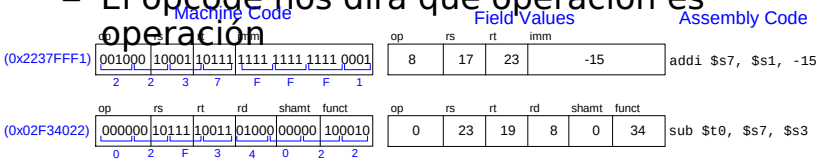
Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`



Program Counter/Contador de Programa (PC): hace seguimiento de la instrucción actual que esta en ejecución

Interpretando el código de maquina

- Comience por el opcode: nos dice como parsear el resto
- Si el opcode es todo 0's
  - Instrucción tipo R
  - Los bits de función nos dice la operación
- Sino
  - El opcode nos dirá que operación es



Programación

- Lenguajes de alto nivel:
  - Ejemplos: C, Java, Python
  - Escritos a un nivel de abstracción mas alto
- El software de alto nivel comúnmente esta construido con:
  - Sentencias if/else
  - Ciclos for
  - Ciclos while
  - arreglos
  - Llamados de función

Ada Lovelace, 1815-1852

- Escribió el primer programa computacional
- Su programa calculo los números de Bernoulli en la maquina analítica de Charles Babbage
- Ella fue la hija del poeta Lord Byron

Instrucciones lógicas

- **and, or, xor, nor**
  - and: util para **enmascar** bits
    - Enmascarar todos excepto el byte menos significativo de un valor:
      0xF234012F AND 0x000000FF = 0x0000002F
  - or: util para **combinar** campos de bit
    - Combine 0xF2340000 con 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - nor: util parar **invertir** bits:
    - A NOR $0 = NOT A
- **andi, ori, xori**
  - El inmediato de 16-bit es extendido con cero (*no* es extendido con signo)
  - nori no se necesitado

Instrucciones Lógicas Ejemplo 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
`and $s3, $s1, $s2`	$s3								
`or  $s4, $s1, $s2`	$s4								
`xor $s5, $s1, $s2`	$s5								
`nor $s6, $s1, $s2`	$s6								

Instrucciones Lógicas Ejemplo 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
`and $s3, $s1, $s2`	$s3	0100	0110	1010	0001	0000	0000	0000	0000
`or  $s4, $s1, $s2`	$s4	1111	1111	1111	1111	1111	0000	1011	0111
`xor $s5, $s1, $s2`	$s5	1011	1001	0101	1110	1111	0000	1011	0111
`nor $s6, $s1, $s2`	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Instrucciones Lógicas

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code		Result							
`andi $s2, $s1, 0xFA34`	$s2								
`ori  $s3, $s1, 0xFA34`	$s3								
`xori $s4, $s1, 0xFA34`	$s4								

Instrucciones Lógicas Ejemplo 2

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

zero-extended

Assembly Code		Result							
`andi $s2, $s1, 0xFA34`	$s2	0000	0000	0000	0000	0000	0000	0011	0100
`ori  $s3, $s1, 0xFA34`	$s3	0000	0000	0000	0000	1111	1010	1111	1111
`xori $s4, $s1, 0xFA34`	$s4	0000	0000	0000	0000	1111	1010	1100	1011

Instrucciones Shift (Desplazamiento)

- sll: desplazamiento lógico a la izquierda
  - **Ejemplo:** sll $t0, $t1, 5 # $t0 <= $t1 << 5
- srl: desplazamiento lógico a la derecha
  - **Ejemplo:** srl $t0, $t1, 5 # $t0 <= $t1 >> 5
- sra: desplazamiento aritmetico a la derecha
  - **Ejemplo:** sra $t0, $t1, 5 # $t0 <= $t1 >>> 5

Instrucciones de Desplazamiento Variable

- sllv: desplazamiento variable lógico a la izquierda
  - **Ejemplo:** sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2
- srlv: desplazamiento variable lógico a la derecha
  - **Ejemplo:** srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2
- srav: desplazamiento variable aritméticc a la derecha
  - **Ejemplo**: srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2

Instrucciones de Desplazamiento

Assembly Code / **Field Values**

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Generando Constantes

- addi usa constantes de 16-bit:

Código C

```
// int es una palabra con signo de
    32-bit
int a = 0x4f3c;
```

Codigo Ensamblador MIP

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- load upper immediate (lui) y ori usan constantes de 32-bit:

Código C

```
int a = 0xFEDC8765;
```

Codigo Ensamblador MI

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

Multiplicación, División

- Registros especiales: lo, hi
- Multiplicación de 32 bits x 32 bits, Resultado de 64 bit
  - mult $s0, $s1
  - Resultado en {hi, lo}
- División de 32-bit, cociente de 32-bit, resto
  - div $s0, $s1
  - Cociente en lo
  - Resto in hi
- Mueve desde los registros especiales lo/hi
  - mflo $s2

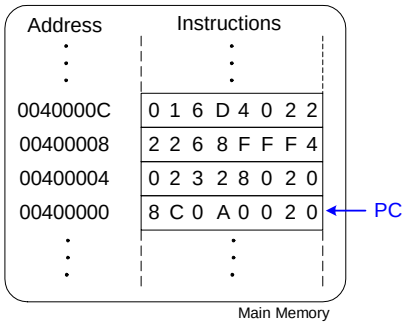
Bifurcación (Branching)

- Ejecute instrucciones fuera de la secuencia
- Tipos de bifurcaciones:
  - **Condicional**
    - branch if equal (beq)
    - branch if not equal (bne)
  - **Incondicional**
    - jump (j)
    - jump con registro (jr)
    - jump y link (jal)

Revisión: Programa Almacenado

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



Bifurcación condicional (beq)

```
# Assembler MIPS
 addi $s0, $0, 4          # $s0 = 0 + 4 = 4
addi $s1, $0, 1           # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2    # $s1 = 1 << 2 = 4
beq  $s0, $s1, target  # esta rama es elegida
addi $s1, $s1, 1       # no se ejecuta
sub  $s1, $s1, $s0       # no se ejecuta

target:                   # label/etiqueta
 add  $s1, $s1, $s0       # $s1 = 4 + 4 = 8
```

La etiqueta o label indica la ubicación de la instrucción
No se pueden usar palabras reservadas y deben ser seguidas por dos puntos (:)

La rama no elegida (bne)

Asembler MIPS

```
  addi    $s0, $0, 4        # $s0 = 0 + 4 = 4
  addi    $s1, $0, 1        # $s1 = 0 + 1 = 1
  sll   $s1, $s1, 2       # $s1 = 1 << 2 = 4
  bne     $s0, $s1, target  # rama no elegida
  addi    $s1, $s1, 1       # $s1 = 4 + 1 = 5
  sub     $s1, $s1, $s0     # $s1 = 5 – 4 = 1

target:
  add     $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```


Bifurcación incondicional (j)

Assembler MIPS

```
addi $s0, $0, 4             # $s0 = 4
  addi $s1, $0, 1           # $s1 = 1
  j       target      # salte a target
  sra     $s1, $s1, 2       # no se ejecuta
  addi    $s1, $s1, 1       # no se ejecuta
  sub     $s1, $s1, $s0 # no se ejecuta

target:
  add     $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

Bifurcación incondicional (jr)

 # Asembler MIPS

0x00002000	addi $s0, $0, 0x2010
0x00002004	jr $s0
0x00002008	addi $s1, $0, 1
0x0000200C	sra $s1, $s1, 2
0x00002010	lw $s3, 44($s1)

jr es una instrucción **tipo-R**.

Componentes de código de alto nivel

- Sentencias if
- Sentencias if-else
- Ciclos while
- Ciclos for

Sentencia If

Código C

```
if (i == j)
  f = g + h;

f = f – i;
```

Código Asembler MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
```


Sentencia If

Código C

```
if (i == j)
  f = g + h;

f = f – i;
```

Código Asembler MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
   bne $s3, $s4, L1
   add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Asembler testea caso opuesto (i != j) al código de alto nivel (i == j)

¿Por qué conviene esto?

Sentencia If/Else

Código C

```
if (i == j)
  f = g + h;
else
  f = f – i;
```

Código Asembler MIPS

Sentencia If/Else

Código C

```
if (i == j)
  f = g + h;
else
  f = f – i;
```

Código Asembler MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
     bne $s3, $s4, L1
     add $s0, $s1, $s2
     j   done
L1:  sub $s0, $s0, $s3
done:
```

Ciclos While

Codigo C

```
// determina la potencia
// de x tal que 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

Código Asembler MIPS

Assembler testea el caso opuesto (pow == 128) al codigo C (pow != 128).

Ciclos While

Código C

```
// determina la potencia
// de x tal que 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

Código Asembler MIPS

```
# $s0 = pow, $s1 = x

      addi $s0, $0, 1
      add  $s1, $0, $0
      addi $t0, $0, 128
while: beq  $s0, $t0, done
      sll  $s0, $s0, 1
      addi $s1, $s1, 1
      j    while
done:
```

Asembler testa el caso opuesto (pow == 128) al código C (pow != 128).

Ciclos For

for (inicialización; condición; operación del loop)
sentencia

- **inicialización:** lo ejecuta antes que parta el loop
- **condición:** es testeada al comienzo de cada iteración
- **Operación del ciclo:** se ejecuta al final de cada iteración
- **sentencia:** se ejecuta cada vez que la condiciona se cumpla

Ciclos For

Código de alto nivel

```
// sume los números del 0 al 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

Código Asembler MIPS

```
# $s0 = i, $s1 = sum
```

Ciclos For

Código C

```
// sume los números del 0 al 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

Código asembler MIPS

computer Architecture

Ciclos For

Código C

```
// sume los números del 0 al 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

Código asembler MIPS

```
# $s0 = i, $s1 = sum
      addi $s1, $0, 0
      add  $s0, $0, $0
      addi $t0, $0, 10
for:  beq  $s0, $t0, done
      add  $s1, $s1, $s0
      addi $s0, $s0, 1
      j    for
done:
```

Comparacion menor que

Código C

```
// sume las potencias de 2
// desde 1 hasta 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

Código asembler MIPS

Comparacion menor que

Código C

```
// sume las potencias de 2
// desde 1 hasta 100
int sum = 0;
int i;

for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

Codigo asembler MIPS

```
# $s0 = i, $s1 = sum
      addi $s1, $0, 0
      addi $s0, $0, 1
      addi $t0, $0, 101
loop:  slt  $t1, $s0, $t0
      beq  $t1, $0, done
      add  $s1, $s1, $s0
      sll  $s0, $s0, 1
      j    loop
done:
```

$t1 = 1 if i < 101

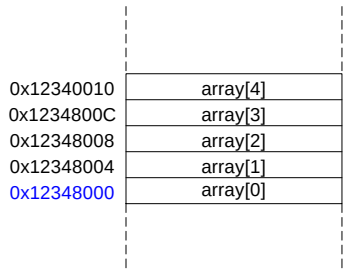
Arreglos

- Queremos acceder a grandes volúmenes de datos similares
- **Indice**: accede a cada elemento
- **Tamaño***:* numero de elementos

computer Architectur

Arreglos

- Arreglo de 5 elementos
- **Dirección base** = 0x12348000 (dirección del primer elemento, array[0])
- El primer paso al acceder a un arreglo: cargar dirección base en un registro



Accediendo a un Arreglo

```
// Código C
    int array[3]9
    array[. ] :  array[. ] ( 09
    array[/ ] :  array[/ ] ( 09
```


Accediendo a un Arreglo

```
// Código C
    int array[3]9
    array[. ] :  array[. ] ( 09
    array[/ ] :  array[/ ] ( 09

# Código asembler MIPS
# $s0 = dirección base del arreglo
    lui  $s. *. x/ 012          # . x/ 012 en la mitad superior de $ s.
  ori  $s. *$s. *. x6. . .      # . x6. . .  en la mitad inferior de $ s.

  lw     $t/ *. &$s. '          # $t/  :  array[. ]
  sll  $ $t/ *$t/ */            # $t/  :  $ $t/  ( 0
  sw     $t/ *. &$s. '          # array[. ] :  $t/

  lw   $t/ *2&$s. '       # $ $t/  :  array[/ ]
  sll   $t/ *$t/ */       # $ $t/  :  $ $t/  ( 0
  sw  $t/ *2&$s. '        # array[/ ] :  $ $t/
```

Usando un For para acceder a un Arreglo

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

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

# Código Asembler MIPS
# $s0 = dirección  base del arreglo, $s1 = i
```


Usando un For para acceder a un Arreglo

```
# Codigo asembler MIPS
# $s0 = direccion base del arreglo, $s1 = i
# codigo de inicionalización
  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      # en caso contrario fin (done)
  sll  $t0, $s1, 2       # $t0 = i * 4 (offset del byte)
  add  $t0, $t0, $s0      # dirección de 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             # repetir
done:
```

Código ASCII

- *American Standard Code for Information Interchange*
- Cada carácter tiene un valor único en byte
  - Por ejemplo, S = 0x53, a = 0x61, A = 0x41
  - Las minúsculas y mayúsculas difieren en 0x20 (32)

Tabla de caracteres

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

Llamada de Funciones

- **El llamador (Caller):** la función que llama (en este caso, main)
- **El destino (Callee):** la función llamada (en este caso, suma)

Código C

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

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


Convenciones para funciones

- **Llamador (Caller):**
  - pasa **argumentos** al destino
  - Salta al destino
- **Destino (Callee):**
  - **ejecuta** la función
  - **retorna** resultado al llamador
  - **retorna** al punto de la llamada
  - **No debe sobrescribir** registros o memoria asignada al llamador

Convenciones para Funciones MIPS

- **Para llamar a función:** salto y enlace (jal)
- **Return** de una función: salto de registro (jr)
- **Argumentos**: $a. +$a1
- **Valor de retorno**: $v.

Llamada de una función

Código C

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

void simple() {
  return;
}
```

Código asembler MIPS

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

0x00401020 simple: jr $ra
```

void significa que simple no retorna ningún valor

Llamadas a una Función

Código C

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

void simple() {
  return;
}
```

Código asembler MIPS

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

0x00401020 simple: jr $ra
```

jal: salta a simple
$ra = PC + 4 = 0x00400204

jr $ra: salta a la dirección en $ra (0x00400204)

Entrada de Argumentos & Valor de retorno

Convenciones MIPS:

- Valores de argumentos: $a0 - $a3
- Valor de retorno: $v0

Entrada de Argumentos & Valor de retorno

Código C

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 argumentos
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;              // valor de retorno
}
```


Entrada de Argumentos & Valor de retorno

Codigo asembler MIPS

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2    # argumento 0 = 2
  addi $a1, $0, 3    # argumento 1 = 3
  addi $a2, $0, 4    # argumento 2 = 4
  addi $a3, $0, 5    # argumento 3 = 5
  jal  diffofsums    # llamada a Función
  add  $s0, $v0, $0  # y = valor retornado
  ...

# $s0 = resultado
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # resultado = (f + g) - (h + i)
  add $v0, $s0, $0   # pon valor retornado en $v0
  jr  $ra            # retornar al llamador
```

Entrada de Argumentos & Valor de retorno

Codigo asembler MIPS

```
# $s0 = resultado
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # resultado = (f + g) - (h + i)
  add $v0, $s0, $0   # pon valor retornado en $v0
  jr  $ra            # retornar al llamador
```

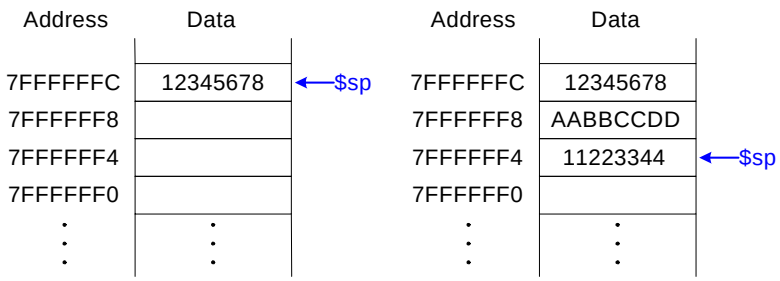
- diffofsums sobrescribe 3 registros: $t0, $t1, $s0
- diffofsums puede usar una *pila* para guardar temporalmente esos registros

La Pila o Stack

- Memoria utilizada para guardar temporalmente las variables
- Al igual que una pila de platos, el último en entrar, es el primero en salir (LIFO)
- **Se expande:** utiliza más memoria cuando se necesita más espacio
- **Se contrae**: utiliza menos memoria cuando el espacio ya no es necesario

La Pila

- Crece hacia abajo (desde las direcciones de memoria mas altas hacia las mas bajas)
- Stack pointer: $sp apunta al tope de la pila



Como las Funciones usan la Pila

- Las funciones llamadas no deben producir efectos colaterales
- Pero diffofsums sobrescribe 3 registros: $t0, $t1, $s0

```
# Asembler MIPS
# $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   # pon valor retornado en $v0
  jr  $ra            # retornar al llamador
```

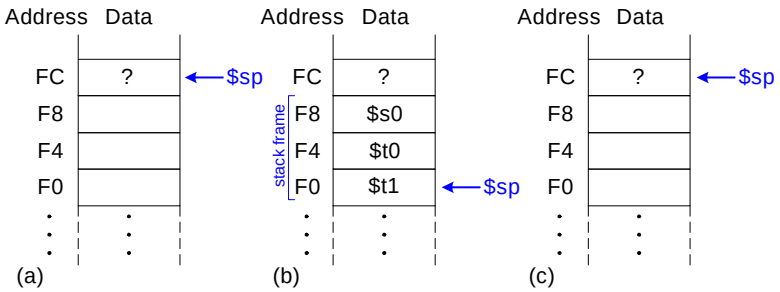
computer Architectur

Almacenando los valores de los registros en la Pila

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # Haga espacio en la pila
                      # para guardar 3 registros
  sw   $s0, 8($sp)    # guarde $s0 en la pila
  sw   $t0, 4($sp)    # guarde $t0 en la pila
  sw   $t1, 0($sp)    # guarde $t1 en la pila
  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   # pon valor retornado en $v0
  lw   $t1, 0($sp)    # restaure $t1 desde la pila
  lw   $t0, 4($sp)    # restaure $t0 desde la pila
  lw   $s0, 8($sp)    # restaure $s0 desde la pila
  addi $sp, $sp, 12   # Libere espacio de la pila
  jr   $ra            # retorne al llamador
```


computer Architectur

La pila durante la llamada de diffofsums



Registros

Preservado *Guardados por destino*	No preservado *Guardados por el llamador*
$s0-$s7	**$t0-$t9**
$ra	**$a0-$a3**
$sp	**$v0-$v1**
La pila esta sobre $sp	**La pila esta debajo $sp**

Múltiples llamadas de Funciones

```
proc1:
  addi $sp, $sp, -4   # Haga espacio en la pila
  sw   $ra, 0($sp)    # Guarde $ra en la pila
  jal  proc2
  ...
  lw   $ra, 0($sp)    # restaure $ra desde la pila
  addi $sp, $sp, 4    # libere espacio de la pila
  jr  $ra             # retorne al llamador
```

Almacenando registros guardados en la pila

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # haga espacio en la pila para
                     # almacenar un registro
  sw  $s0, 0($sp)    # guarde $s0 en la pila
                     # no necesita guardar $t0 o $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   # pon valor retornado en $v0
  lw  $s0, 0($sp)    # restaure $s0 de la pila
  addi $sp, $sp, 4   # libere espacio de la pila
  jr  $ra            # retorne al llamador
```


Llamada a Función Recursiva

Código de Alto Nivel

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Llamada a Función Recursiva

Código Asembler MIPS

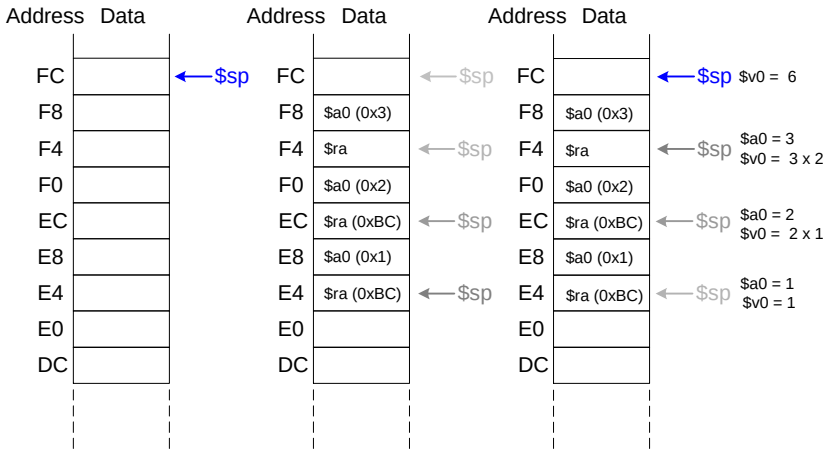
computer Architectur

Llamada a Función Recursiva

Código Asembler MIPS

```
0x90 factorial: addi $sp, $sp, -8  # haga espacio
0x94            sw   $a0, 4($sp)   # guarde $a0
0x98            sw   $ra, 0($sp)   # guarde $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # n <= 1 ?
0xA4            beq  $t0, $0, else # no: ir al else
0xA8            addi $v0, $0, 1    # si: retorne 1
0xAC            addi $sp, $sp, 8   # restaure $sp
0xB0            jr   $ra           # retorne
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # llamada recursiva
0xBC            lw   $ra, 0($sp)   # restaure $ra
0xC0            lw   $a0, 4($sp)   # restaure $a0
0xC4            addi $sp, $sp, 8   # restaure $sp
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # retorne
```

Pila Durante llamadas recursivas



Resumen Llamada a Función

- **Llamador (Caller)**
  - Ponga los argumentos in $a. +$a1
  - Guarde cualquier registro necesario ($ra, quizás $t. +$t7)
  - jal destino&callee'
  - Restaure registros
  - Busque resultado en $v.
- **Destino (Callee)**
  - Guarde registros que pueden ser sobrescritos &$s. + $s5)
  - Ejecute función
  - Ponga resultado en $v.
  - Restaure registros
  - jr $ra

Modos de Direccionamiento

¿Como direccionamos los operandos?

- Solo por Registro
- Inmediato
- Con Direccionamiento Base
- Relativo al PC
- Pseudo Directo

Modos de Direccionamiento

Solo Por Registro

- Los Operandos se encuentran en registros
  - **Ejemplo:** add $s0, $t2, $t3
  - **Ejemplo:** sub $t8, $s1, $0

Inmediato

- Inmediato de 16-bit es usado como operando
  - **Ejemplo:** addi $s4, $t5, -73
  - **Ejemplo:** ori $t3, $t7, 0xFF

Modos de Direccionamiento

Direccionamiento Base

- Dirección del operando es:

  Dirección base + inmediato con signo-extendido

  - **Ejemplo:** lw $s4, 72($0)
    - dirección = $0 + 72
  - **Ejemplo:** sw $t2, -25($t1)
    - dirección = $t1 - 25

Modos de Direccionamiento

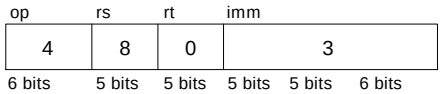
Direccionamiento relativo al PC

```
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



Modos de Direccionamiento

Direccionamiento Pseudo-directo

0x0040005C jal suma

...

0x004000A0 suma: 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

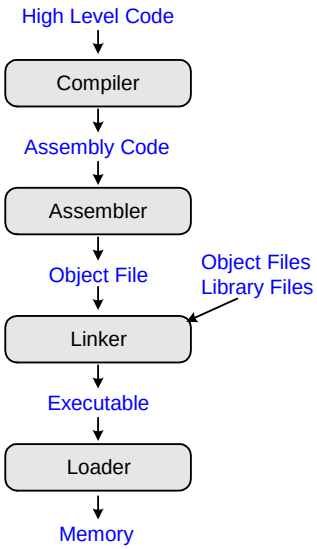
Field Values

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	

¿Como compilar & Correr un Programa?



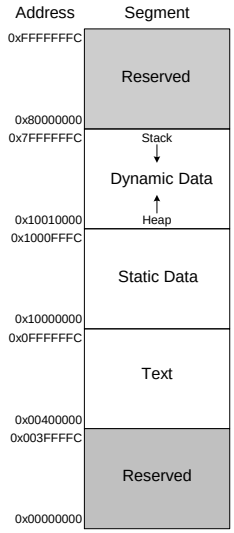
Grace Hopper, 1906-1992

- Se graduó de la Universidad de Yale con un Ph.D. en matemáticas
- Desarrollo el primer compilador
- Ayudo a desarrollar el lenguaje de programación COBOL
- Oficial Naval muy galardonada
- Recibió la Medalla Victoria de la II Guerra Mundial y la Medalla al Servicio de la Defensa de los EEUU, y muchas otras mas

¿Que se guarda en la Memoria?

- Instrucciones (también llamado *texto (text)*)
- Datos
  - Globales/estáticas: se asignan antes que el programa parta
  - Dinámica: asignada al ejecutar el programa
- ¿Cuan grande es la memoria?
  - A lo mas 2^{32} = 4 gigabytes (4 GB)
  - De la dirección 0x00000000 a la 0xFFFFFFFF

Mapa de Memoria MIPS



Ejemplo de Programa: Código C

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


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

  return y;
}


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

Ejemplo Programa: Asembler MIPS

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


int main(void)
{


 f = 2;
 g = 3;

 y = suma(f, g);
 return y;
}

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

```
.data
f:
g:
y:
.text
main:
 addi $sp, $sp, -4   # stack frame
 sw   $ra, 0($sp)    # guarde $ra
 addi $a0, $0, 2     # $a0 = 2
 sw   $a0, f        # f = 2
 addi $a1, $0, 3     # $a1 = 3
 sw   $a1, g         # g = 3
 jal  suma           # llamar suma
 sw   $v0, y         # y = suma()
 lw   $ra, 0($sp)    # restaure $ra
 addi $sp, $sp, 4    # restaure $sp
 jr   $ra           # retorne al SO
suma:
 add  $v0, $a0, $a1  # $v0 = a + b
 jr   $ra           # retornar
```


Programa Ejemplo: Tabla de Símbolos

Símbolo	Dirección

Cargue programa en SPIM e identifique la dirección de cada símbolo

Programa Ejemplo: Tabla de Símbolos

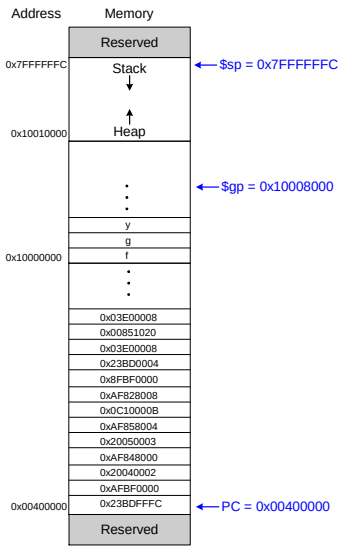
Símbolo	Dirección
f	0x100000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Programa Ejemplo: Ejecutable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	**Address**	**Instruction**
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008
Data segment	**Address**	**Data**
	0x10000000	f
	0x10000004	g
	0x10000008	y

```
addi $sp, $sp, -4
sw   $ra, 0 ($sp)
addi $a0, $0, 2
sw   $a0, 0x8000 ($gp)
addi $a1, $0, 3
sw   $a1, 0x8004 ($gp)
jal   0x0040002C
sw   $v0, 0x8008 ($gp)
lw   $ra, 0 ($sp)
addi $sp, $sp, -4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

Programa Ejemplo: En Memoria



Misceláneo

- Pseudo-instrucciones
- Excepciones
- Instrucciones con signo y sin signo
- Instrucciones con punto flotante

Pseudo-instrucciones

Pseudoinstrucción	Instrucciones MIPS
li $s0, 0x1234AA77	lui $s0, 0x1234 ori $s0, 0xAA77
clear $t0	add $t0, $0, $0
move $s1, $s2	add $s2, $s1, $0
nop	sll $0, $0, 0

Excepciones

- Llamadas a funciones no agendadas al *manager de excepciones* (*exception handler)*
- Causado por:
  - Hardware, también llamadas *interrupción*, e.g., teclado
  - Software, también llamadas *traps (trampas)*, e.g., instrucción no definidas
- Cuando una excepción ocurre, el procesador:
  - Registre la causa de la excepción
  - Saltar al manager de excepciones (en dirección de instrucciones 0x80000180)
  - Retornar al programa

Causas de Excepciones

Excepción	Causa
Interrupción de Hardware	0x00000000
Llamada de Sistema	0x00000020
Punto de quiebre/ División por 0	0x00000024
Instrucción Indefinida	0x00000028
Desbordamiento Aritmético	0x00000030

Mirando hacia el futuro

Por motivos de tiempo, el manejo de Excepciones y punto fijo ni punto flotante no se veran en detalle (vea referencias o consulte al profesor si le interesa)

¿Que viene?

Micro-arquitectura – construya un procesador MIPS en hardware

Traiga lápices de distintos colores

Veremos un par de casos sencillos por tiempo ...

Registros de Excepción

- No son parte de los archivos de registros
  - **Cause**: Registra causa de la excepción
  - **EPC** (Excepción del PC): Registra PC donde la excepcion ocurrió
- EPC y Cause: parte del Coprocesador 0
- Moverse desde el Coprocesador 0
  - mfc0 $k0, EPC
  - Mueve contenido del EPC en $k0

Flujo de una Excepción

- Procesador guarda causa y PC de la excepción en Cause y EPC
- Procesador salta al manager de excepciones (0x80000180)
- Manager de Excepción:
  - Guardar registros en la pila
  - Leer registro Cause
    mfc0 $k0, Cause
  - Maneja excepción
  - Restaura registros
  - Retorna al programa
    mfc0 $k0, EPC
    jr $k0

Instrucciones con Signo & Sin-Signo

- Suma y resta
- Multiplicación y división
- Set menor que

Suma & Resta

- **Con Signo:** add, addi, sub
  - La misma operación como versiones sin signo
  - Pero el procesador toma la excepcion al desbordar
- **Sin Signo:** addu, addiu, subu
  - No toma excepción al desbordar

Nota: addiu extiende signo del inmediato

Multiplicación & División

- **Con Signo:** mult, div
- **Sin Signo:** multu, divu

Set Menor que

- **Con Signo:** slt, slti
- **Sin Signo:** sltu, sltiu

Nota: sltiu extiende el signo del inmediato antes de compararlo con el registro

Cargas

- **Con Signo:**
  - Extiende signo para crear valor de bits de 32 bits para cargarlo en un registro
  - Cargar la mitad de una palabra (halfword): lh
  - Cargar byte: lb
- **Sin Signo:**
  - Extiende ceros para crear valor de bits de 32 bits
  - Cargar media palabra sin signo: lhu
  - Cargar byte: lbu

Instrucciones con Punto Flotante

- Coprocesador de Punto Flotante (Coprocesador 1)
- 32 registros de puntos flotante de 32-bits ($f0-$f31)
- Valores de Doble Precisión que se mantienen en dos registros de puntos flotantes
  - e.g., $f0 y $f1, $f2 y $f3, etc.
  - Registros de punto flotante de doble precisión: $f0, $f2, $f4, etc.

Instrucciones con Puntos Flotante

Nombre	Numero Registro	Uso
$ $fv. +$fv/	0, 2	Valores a retornar
$ $ft. +$ft1	4, 6, 8, 10	Variables temporales
$ $fa. +$fa/	12, 14	Argumentos Función
$ $ft2 +$ft6	16, 18	Variables temporales
$ $fs. +$fs3	20, 22, 24, 26, 28, 30	Variables guardadas

Formato Instrucción Tipo-F

- Opcode = 17 (010001_2)
- Simple precisión:
  - cop = 16 (010000_2)
  - add.s, sub.s, div.s, neg.s, abs.s, etc.
- Doble-precisión:
  - cop = 17 (010001_2)
  - add.d, sub.d, div.d, neg.d, abs.d, etc.
- 3 operandos de registros:
  - fs, ft: operandos de origen
  - fd: Operando de destino

F-Type

op	cop	ft	fs	fd	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Bifurcación de Punto Flotante

- Set/clear bandera (flag) de condición:fpcond
  - Igualdad: c.seq.s, c.seq.d
  - Menor que: c.lt.s, c.lt.d
  - Menor que o igual: c.le.s, c.le.d
- Bifurcacion condicional
  - bclf: cambia rama si fpcond es FALSE
  - bclt: cambia rama si fpcond es TRUE
- Cargar y Escribir
  - lwc1: lwc1 $ft1, 42($s1)
  - swc1: swc1 $fs2, 17($sp)