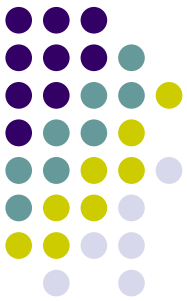
Introducción a Arquitectura MIPS

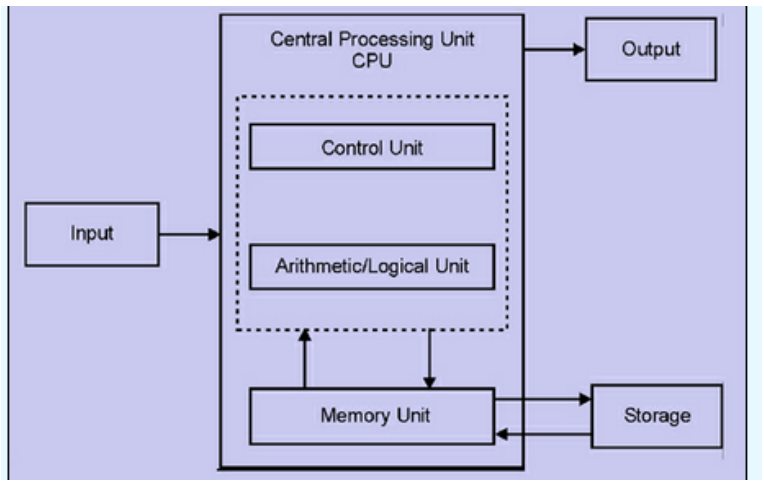
Elías Todorovich

Intro. Arquitectura de Sistemas - 2017

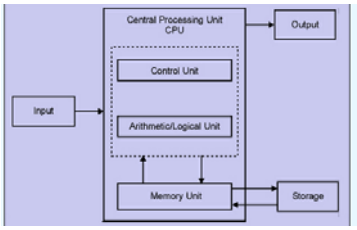


Estructura de una computadora

- Una computadora digital consiste de un sistema interconectado de procesadores, memorias y dispositivos de entrada/salida.

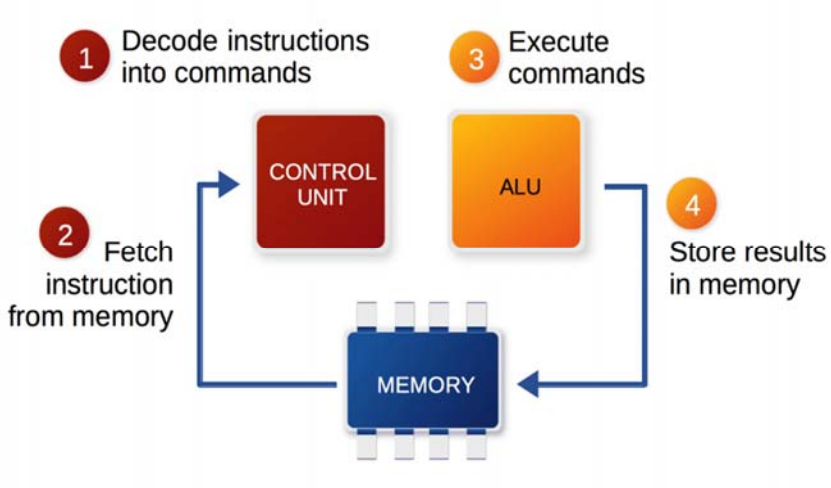


CPU



- La CPU está compuesta por:
  - unidad de control (UC): responsable del la captura de instrucciones desde la memoria principal.
  - unidad aritmética lógica (ALU): ejecuta operaciones para llevar a cabo las instrucciones.
  - registros (memoria pequeña de alta velocidad: almacena resultados temporales y cierta información de control.
  - contador de programa (PC), apunta a la próxima instrucción a ejecutar.
  - registro de instrucción (IR), contiene la instrucción a ejecutar.

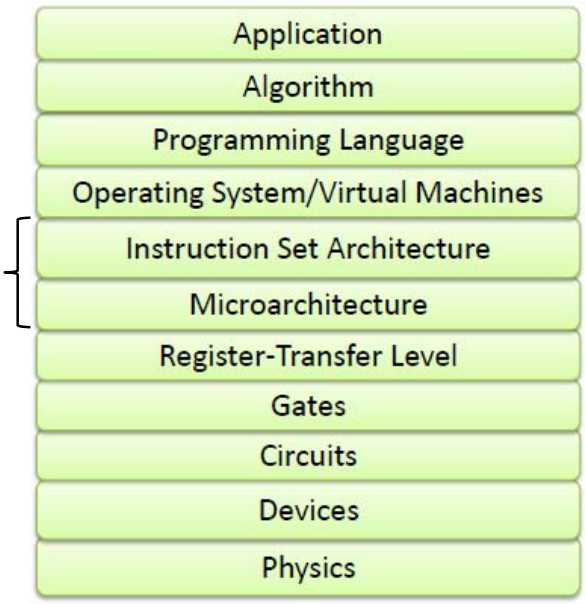
Funcionamiento de la CPU



Funcionamiento de la CPU

1. Pone la próxima instrucción de la memoria en el registro de instrucción.
2. Cambia el contador de programa para que apunte a la instrucción siguiente.
3. Determina el tipo de instrucción.
4. Si la instrucción usa una palabra de memoria, determina donde está.
5. Copia la palabra, si es necesario, en un registro de la CPU.
6. Ejecuta la instrucción.
7. Vuelve al punto inicial 1 para comenzar con la siguiente instrucción.

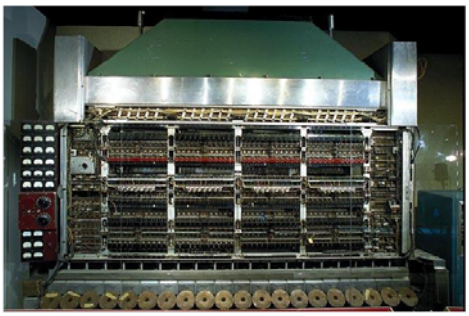
Arquitectura de Computadoras: Abstracción



Programas
Estructuras de datos
C, C++, Java, …
Device drivers
Instrucciones y registros
Ruta de datos, unidad de control
Sumadoes, multiplexores, memorias
AND, OR, NOT, NAND
CMOS
Transistores, pull-up, pistas
Electrones

Arquitectura de Computadoras: Von-Neumann

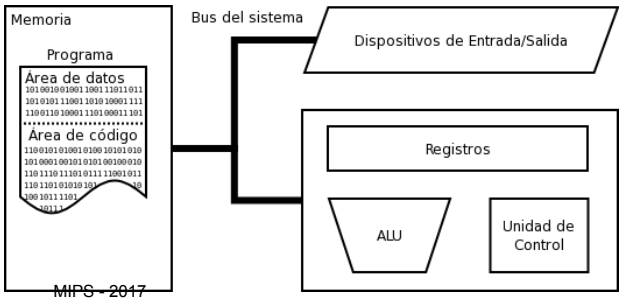
- Las primeras computadoras tenían programas fijos. En esos casos se "diseñaba a medida", en vez de programar el procesador.
- En la época de von-Neumann aparece el concepto de computadora con **programa almacenado**. Para ello hace falta:
  - Disponer de la arquitectura del juego de instrucciones
  - Detallar la computación como una serie de instrucciones: el programa.



IAS Machine. Design directed by John von Neumann.
First booted in Princeton NJ in 1952
Smithsonian Institution Archives
(Smithsonian Image 95-06151)
3m x 3m x 0.80m

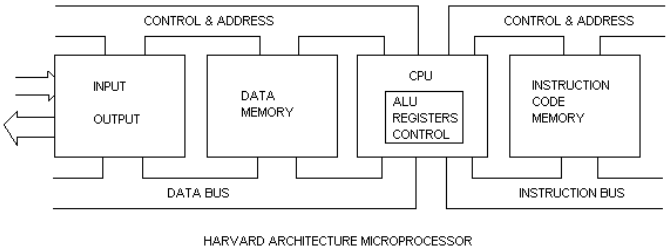
Arquitectura Von-Neumann

- La arquitectura de **von Neumann** (Eckert-Mauchly) es un modelo de diseño de computadores que usa:
  - Unidad de procesamiento
  - Una sola memoria para instrucciones y datos.
- Problemas de la arquitectura de **von Neumann** :
  - Cuello de botella: velocidad de transferencia de datos entre memoria y CPU

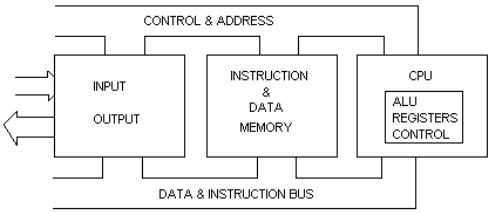


Arquitectura Harvard

- Característica: Dos memorias físicamente separadas para datos e instrucciones.
- De esta manera se puede acceder a instrucciones y datos en paralelo.



HARVARD ARCHITECTURE MICROPROCESSOR

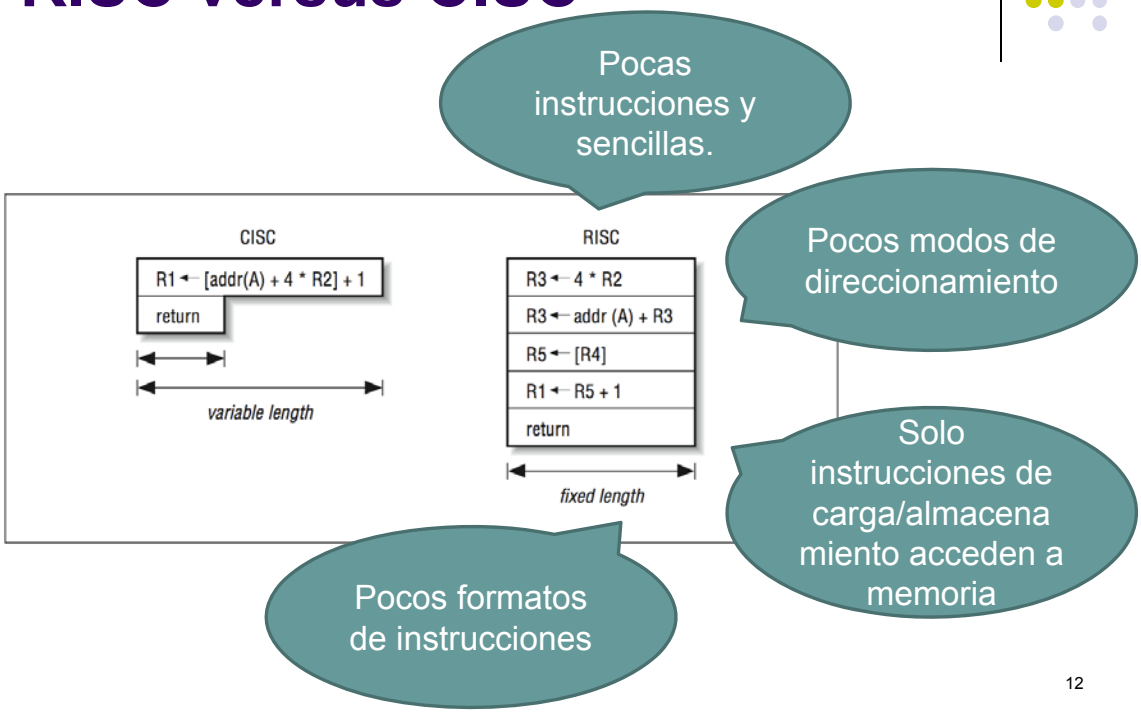


VON NEUMANN ARCHITECTURE MICROPROCESSOR

Arquitectura del juego de instrucciones

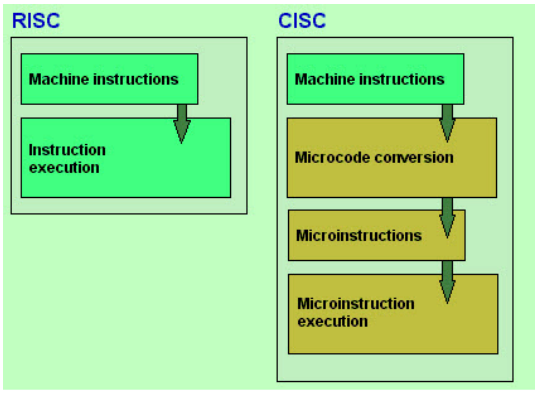
- El procesador visto desde afuera
  - Conjunto de instrucciones
  - Operandos
  - Registros
- **Instrucciones:** comandos en el lenguaje de la computadora
  - **Lenguaje máquina:** instrucciones en el formato que leen las computadoras (1s y 0s)
  - **Lenguaje ensamblador:** Instrucciones en un lenguaje legible por las personas.

RISC versus CISC



RISC versus CISC

Instrucciones y sencillas se ejecutan en máquinas sencillas.



RISC versus CISC

- En el mercado de procesadores…

Diferentes ISAs

- x86
- MIPS
- ARM
- PowerPC
- IA-64
- SPARC
- CODE-2
- … otras

- Arquitectura RISC
- Desarrollada por John Hennessy y equipo en Stanford en la década de1980.
- Para el año 2004 se habían vendido mas de 300 millones de procesadores MIPS.

Procesadores MIPS

- En 1981, un equipo liderado por John L. Hennessy en la Univ. de Stanford comenzó a trabajar en el primer procesador MIPS.
- A principios de los '90 MIPS Technologies comenzó a otorgar licencias de sus diseños a terceros, de ahí procedían más de la mitad de los ingresos de MIPS.
- Los procesadores MIPS se utilizaron por ejemplo en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2, etc.
- Debido a que su conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades a menudo se basan en la arquitectura MIPS.

R10000 (Toshiba TC86R10000-200,1996)

Emotion Engine (Sony, 2000)
MIPS-IV (R4000) de 128 Bits

Principios de diseño en MIPS

1. Simplicidad. La simplicidad favorece la regularidad.
2. Optimizar el caso más común.
3. Cuanto más pequeño, más rápido.
4. Los buenos diseños requieren de buenas decisiones de compromiso.

Juego de Instrucciones: Suma

Código C

```
a = b + c;
```

Código ensamblador MIPS

```
add a, b, c
```

- **add:** el mnemónico indica la operación a ejecutar
- **b, c:** operandos (registros sobre los que la operación se ejecuta)
- **a:** registro destino (en el que se escribe el resultado)

Resta

- Similar a la suma – solo el mnemónico cambia

Código C	Código ensamblador MIPS
`a = b - c;`	`sub a, b, c`

- **sub:** mnemónico
- **b, c:** operandos
- **a:** resultado

Lectura de memoria

- *Load word* (`lw`)
- Ejemplo:

```
lw $s0, 4($t1)
```

- Cálculo de la dirección:
  - Se suma la dirección base (`$t1`) al desplazamiento (4), o sea dir = (`$t1` + 4)
- Resultado:
  - `$s0` almacena el valor almacenado en (`$t1` + 4)

Banco de Registros

Nombre	Número de registro	Uso
`$0`	0	Valor constante 0
`$at`	1	Reservado ensamblador
`$v0-$v1`	2-3	Retorno de función
`$a0-$a3`	4-7	Argumentos función
`$t0-$t7`	8-15	Temporarios
`$s0-$s7`	16-23	Variables a almacenar
`$t8-$t9`	24-25	Más temporarios
`$k0-$k1`	26-27	Reservado OS
`$gp`	28	Puntero global
`$sp`	29	Puntero a la pila
`$fp`	30	Puntero al frame
`$ra`	31	Dirección retorno función

Escritura en memoria

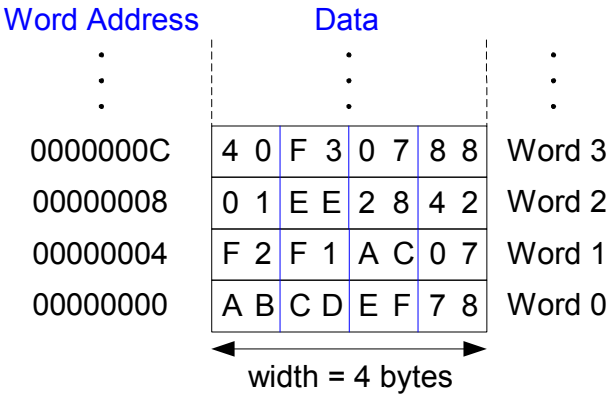
- *Store word* (`lw`)
- Ejemplo:

```
sw $s1, 0($t1)
```

- Cálculo de la dirección:
  - Se suma la dirección base (`$t1`) al desplazamiento (0), o sea dir = (`$t1` + 0) en este ejemplo
- Resultado:
  - en (`$t1` + 0) se almacena el valor contenido en `$s1`

Contenido de memoria

- Cada byte en memoria tiene su dirección
- Una palabra de 32 bits = 4 bytes, de manera que las direcciones de las palabras se incrementan de 4 en 4



Operandos inmediatos

- A diferencia de `add` y `sub`, que usan registros, usar constantes o valores *inmediatos*
- Una variante de add es addi

Código C

```
a = a + 4;
b = a – 12;
```

Código ensamblador MIPS

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

Código Máquina

- Recordemos que las computadoras solo entienden 1s y 0s
- El código máquina es la representación binaria de las instrucciones
- En MIPS todas las instrucciones son de 32 bits
- 3 formatos de instrucciones:
  - **R-Type:** los operandos son registros
  - **I-Type:** un operando es inmediato
  - **J-Type:** útil para saltos

Formato Tipo-R

R-Type

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

- 3 registros como operandos:
  - `rs`, `rt`: fuentes
  - `rd`: resultado
- Otros campos:
  - `op`: código de operación
  - `funct`: la función
    en conjunto con opcode, definen la operación a realizar
  - `shamt`: *shift amount* para instrucciones de desplazamiento

Formato Tipo-R: ejemplos

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	

Formato Tipo-I

I-Type

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

- 3 operandos:
  - `rs`, `rt`: registros
  - `imm`: constante de 16 bits en complemento a 2
- Otros campos:
  - `op`: código de operación
  - Todas las instrucciones tienen opcode
  - En este caso solamente opcode determina la operación

Formato Tipo-I: Ejemplos

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

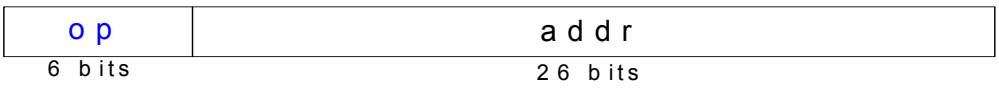
Field Values

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	

Formato Tipo-J

J-Type



- El operando es una dirección de 26 bits (`addr`)
- Se usa en instrucciones como jump (`j`)

Los 3 formatos en MIPS

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

Instrucciones lógicas

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

- En programación las instrucciones lógicas también se usan para:
  - Máscaras (AND)
  - Combinación de bits (OR)
  - Usos en lógica (NOR como inversor)
    - A NOR $0 = NOT A

Instrucciones de desplazamiento

- `sll:` shift left lógico
  - **Ejemplo:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl:` shift right lógico
  - **Ejemplo:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra:` shift right aritmético
  - **Ejemplo:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

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: lui

- Constantes de 16 bits usando `addi`:

Código C

```
// int is a 32-bit signed
  word
int a = 0x4f3c;
```

Código ensamblador MIPS

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

- Constantes de 32 bits usando load upper immediate (`lui`) y `ori`:

Código C

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

Código ensamblador MIPS

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

Saltos condicionales (beq)

```
# Ejemplo salto condicional
 addi $s0, $0, 4         # $s0 = 0 + 4 = 4
 addi $s1, $0, 2         # $s1 = 0 + 2 = 2
 add  $s1, $s1, $s1      # $s1 = 2 + 2 = 4
 beq  $s0, $s1, target   # salto tomado
 addi $s1, $s1, 1        # no se ejecuta
 sub  $s1, $s1, $s0      # no se ejecuta

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


Saltos condicionales (bne)

```
# Otro ejemplo salto condicional
  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    # salto no tomado
  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
```

Saltos incondicionales (j)

```
# Ejemplo salto incondicional
  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    # salto no tomado
  addi      $s1, $s1, 1         # $s1 = 4 + 1 = 5
  sub       $s1, $s1, $s0       # $s1 = 5 – 4 = 1
  j         end                 # salto a end
target:
  add       $s1, $s1, $s0       # no se ejecuta
end:
  ...
```


Modos de direccionamiento

- Dónde está el operando?
- En MIPS son pocos y sencillos (RISC)
  1. Mediante registros
  2. Inmediato
  3. Mediante registro base
  4. Relativo a PC
  5. Pseudo directo

Modos de direccionamiento

1. Mediante registros

- El operando está en un registro
  - **Ej.:** `add $s0, $t2, $t3`
  - **Ej.:** `sub $t8, $s1, $0`

2. Inmediato

- El operando está en los 16 bits inferiores de la instrucción
  - **Ej.:** `addi $s4, $t5, -73`

Modos de direccionamiento

3. Mediante registro base

- La dirección del operando es:
  `Direccion base + inmediato (ext. signo)`
  - **Ej.** `lw  $s4, 72($0)`
    - dirección = `$0 + 72`
  - **Ej.** `sw  $t2, -24($t1)`
    - dirección = `$t1 - 24`

Modos de direccionamiento

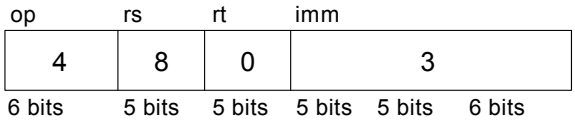
4. Relativo a PC

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

Assembly Code

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

Field Values

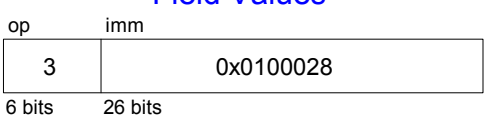


Modos de direccionamiento

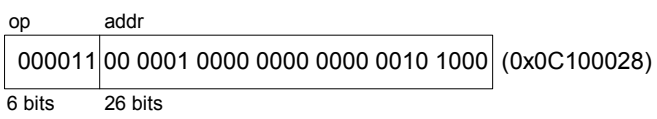
5. Pseudo directo

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

Field Values

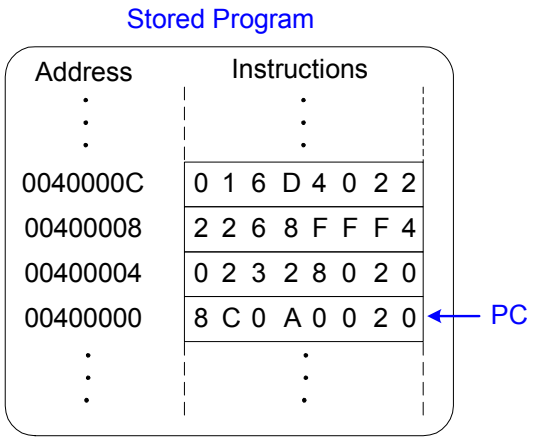


Machine Code



El programa almacenado

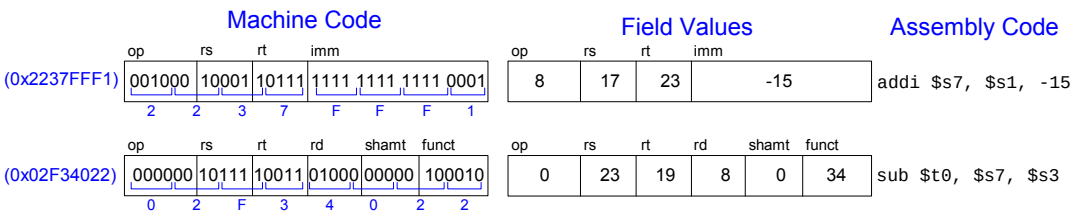
Assembly Code	Machine Code
`lw $t2, 32($0)`	`0x8C0A0020`
`add $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub $t0, $t3, $t5`	`0x016D4022`



Program Counter (PC): indica la dirección de la instrucción que se ejecuta

Decodificación de instrucciones

- Mirar el opcode
- Si opcode es 0
  - Es una instrucción de tipo-R
  - Los bits del campo Function definene la operación
- Si no es 0
  - opcode determina la operación



Lenguajes de alto nivel

- Por ejemplo
  - C, C++, Java, Python
  - Mayor nivel de abstracción
- Sentencias y construcciones comunes en lenguajes de alto nivel: :
  - if/else
  - for
  - while
  - arrays
  - function calls

Ada Lovelace (1815-1852) escribió el primer programa: calculaba números de Bernoulli en el Analytical Engine de Charles Babbage

Sentencia If

Código C

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

f = f - i;
```

Código ensamblador de MIPS

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

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

En ensamblador se evalúa el caso opuesto (`i != j`) que en C (`i == j`)

Sentencia If/Else

Código C

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

Código ensamblador de 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:
```


Bucle while

Código C

```
// determina el exponente
// x tal que 2^x = 128
int pow = 1;
int x   = 0;

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

Código ensamblador de 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:
```

En ensamblador se evalúa el caso opuesto **(pow == 128)** que en C **(pow != 128)**.

Bucle For

Código C

```
// suma los enteros de 0 a 9
int sum = 0;
int i;

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

Código ensamblador de 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:
```


Comparación por menor

Código C

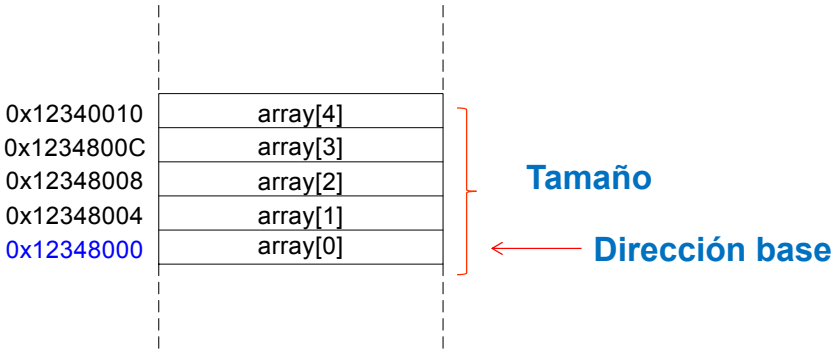
```
// suma las potencias de 2
// desde 1 hasta 100
int sum = 0;
int i;

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

Código ensamblador de 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

Arrays



Recorriendo arrays

Código C

```
int array[1000];
int i;

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

Código ensamblador de MIPS

```
# $s0 = direccion base array, $s1 = i
# inicialización
  lw  $s0, 0($t3)          # $s0 = mem($t3)
  addi $s1, $0, 0          # i = 0
  addi $t2, $0, 1000       # $t2 = 1000
```

Recorriendo arrays (cont.)

```
loop:
  slt  $t0, $s1, $t2      # i < 1000?
  beq  $t0, $0, done      # if no, ir a done
  sll  $t0, $s1, 2        # $t0 = i * 4 (byte offset)
  add  $t0, $t0, $s0      # direccion 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:
```


Llamados a funciones

Código C

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

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

1. main le pasa argumentos a sum
2. Salta a sum
3. sum hace su función
4. sum devuelve un resultado a main
5. sum retorna a main
6. sum no debe sobreescribir registros o memoria que necesite main

Convenciones para funciones

- **Para saltar a una función:** jump and link (`jal`)
- **Retorno desde una función**: jump register (`jr`)
- **Argumentos o parámetros**: $a0 - $a3
- **Valor retornado**: $v0

Conveniones: Ejemplo

Código C

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

void simple() {
  return;
}
```

Código ensamblador de 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)

Saltos jal y jr

```
# jal y jr son saltos incondicionales
# jal es una instruccion de tipo-J
# jr es una instruccion tipo-R

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

0x00401020 simple: jr $ra
```


Llamados a funciones: argumentos y valor de retorno

Código C

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

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

Funciones: código MIPS

```
main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # llamado a funcion
  ...

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    # resultado en $v0
  jr  $ra             # return
```

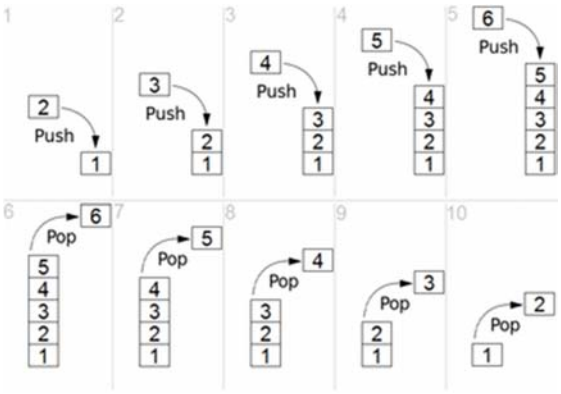
Sobreescribe 3 registros

Resguardo de registros entre llamadas

- `diffofsums` sobreescribe 3 registros: `$t0`, `$t1`, `$s0`
- `diffofsums` puede guardar temporariamente los registros en una pila en memoria

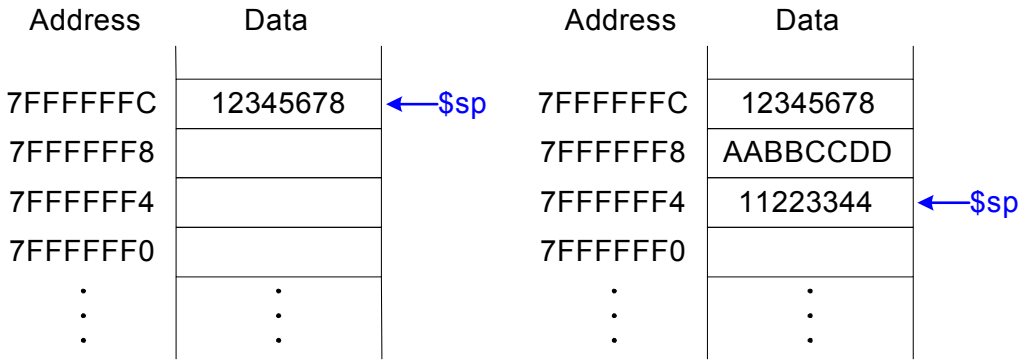
Pila en memoria

- Es una zona de la memoria que se usa para guardar variables temporalmente
- Estructura del tipo “el último que entra es el primero que sale” … o LIFO



Pila en MIPS

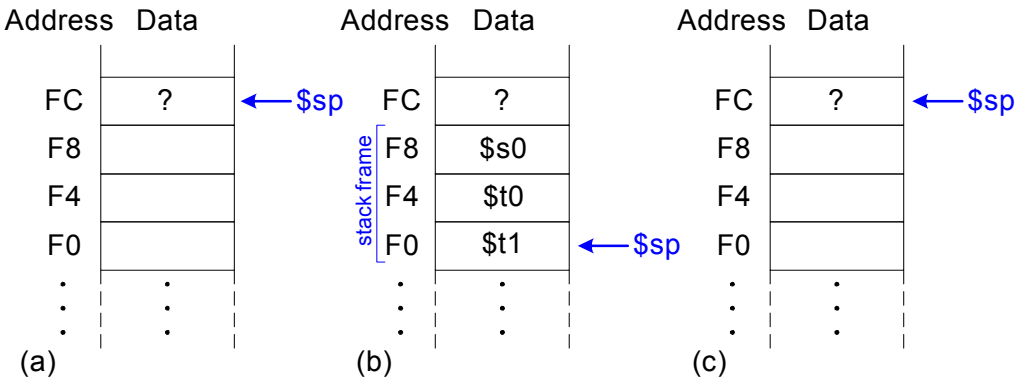
- Crece para abajo (desde direcciones superiores a inferiores)
- Stack pointer: `$sp` apunta al tope de la pila



diffofsums revisado

```
diffofsums:
  addi $sp, $sp, -12  # hace espacio en la pila
                      # para 3 registros
  sw   $s0, 8($sp)    # guarda $s0
  sw   $t0, 4($sp)    # guarda $t0
  sw   $t1, 0($sp)    # guarda $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   # resultado en $v0
  lw   $t1, 0($sp)    # restaura $t1
  lw   $t0, 4($sp)    # restaura $t0
  lw   $s0, 8($sp)    # restaura $s0
  addi $sp, $sp, 12   # recupera espacio de la pila en memoria
  jr   $ra            # return
```


diffofsums revisado: pila



Pila en MIPS: convenciones

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
$s0-$s7	**$t0-$t9**
$ra	**$a0-$a3**
$sp	**$v0-$v1**
Pila arriba de $sp	**Pila abajo de $sp**

diffofsums revisado según la convención

```
diffofsums:
  addi $sp, $sp, -4   # hace espacio en la pila
                      # para 1 registro
  sw  $s0, 0($sp)     # guarda $s0
                      # no es su tarea guardar $tx
  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    # resultado en $v0
  lw  $s0, 0($sp)     # restaura $s0
  addi $sp, $sp, 4    # recupera espacio de la pila
  jr  $ra             # return
```

Funciones que llaman a funciones

```
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
```


Resumen de funciones

- **Función llamadora**
  - Poner argumentos en $a0-$a3
  - Guardar registros si es necesario ($ra, $t0-t9)
  - jal funcion
  - Restaurar registros
  - El resultado se encuentra en $v0
- **Función llamada**
  - Guardar registros si es necesario ($s0-$s7)
  - Ejecurar la función
  - Poner el resultado en $v0
  - Restaurar registros
  - jr $ra