

Arquitectura del conjunto de instrucciones (ISA)

*“Buenos Días, doctor Chandra. Soy HAL.
Estoy listo para mi primera lección”
2001. A Space Odyssey, Arthur Clarke*



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.



Tipos de datos

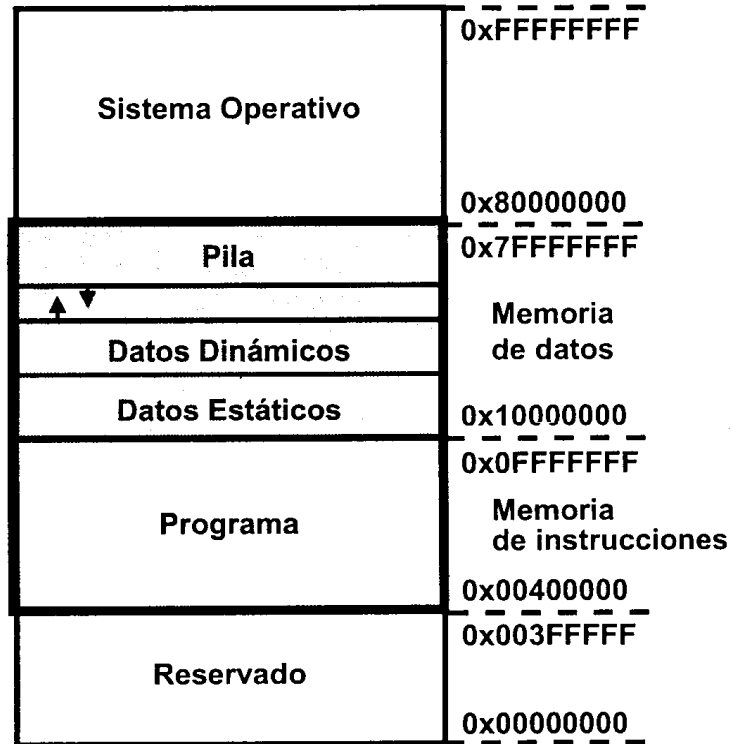
- Tipos de datos soportados por el MIPS R2000

Dato	Tamaño	Datos representados
Ascii	8 bits	Caracteres
Byte	8 bits	Números enteros con/sin signo
Word	32 bits	
Float	32 bits	
Double	64 bits	

- **Byte** : el procesador no opera en 8 bits pero dispone de instrucciones para carga y descarga de este tipo de datos. Con el mismo se puede representar caracteres o números.
- **Word** : el procesador dispone de instrucciones para el manejo de datos word sin signo y con signo en notación complemento a dos.
- **Float** : representa números reales en simple precisión según el estandar IEEE-754.
- **Double** : idem a float en doble precisión.

Organización de la memoria

- ✎ Espacio direccionable en el MIPS R2000
 - ✎ 4 Gbytes = 2 Ghalf = 1 Gword
 - ✓ 2^{32} bytes con direcciones que van desde 0 a $2^{32} - 1$
 - ✓ 2^{30} palabras (4 bytes) con direcciones: 0, 4, 8, ..., $2^{32} - 4$
- ✎ Memoria accesible por el usuario se encuentra en el rango [0x00400000, 0x7FFFFFFF]





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

```
a = b - c;
```

Código ensamblador MIPS

```
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 $\text{dir} = (\$t1 + 4)$
- Resultado:
 - `$s0` almacena el valor de la memoria 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* (*sw*)
- Ejemplo:
`sw $s1, 0($t1)`
- Cálculo de la dirección:
 - Se suma la dirección base ($\$t1$) al desplazamiento (0), o sea $\text{dir} = (\$t1 + 0)$ 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

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



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
```



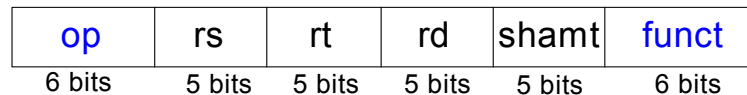
Código Máquina

- 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



- 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

Field Values

	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

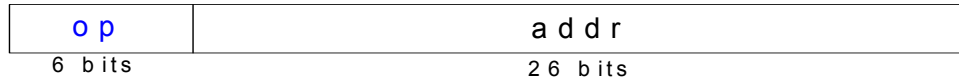
Machine Code

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 - T y p e

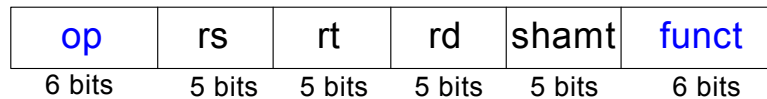


- El operando es una dirección de 26 bits (addr)
- Se usa en instrucciones como jump (j)

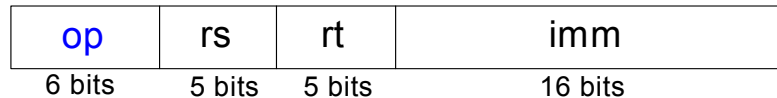


Los 3 formatos en MIPS

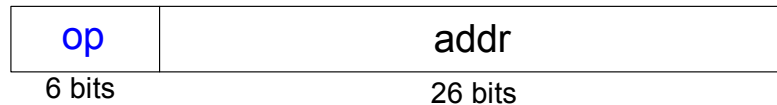
R-Type



I-Type



J-Type



Instrucciones lógicas

Source Registers

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

Assembly Code

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

Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000



Instrucciones lógicas

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



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

sll \$t0, \$s1, 2

srl \$s2, \$s1, 2

sra \$s3, \$s1, 2

Field Values

op	rs	rt	rd	shamt	funct
0	0	17	8	2	0
0	0	17	18	2	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
```



Salto 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
```

Salto 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:

Dirección 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, else1
0x14          addi   $v0, $0, 1
0x18          addi   $sp, $sp, i
0x1C          j      continue
0x20      else1: addi   $a0, $a0, -1
0x24          jal    factorial
```

Assembly Code

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

Field Values

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

Modos de direccionamiento

5. Pseudo directo

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

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

El programa almacenado

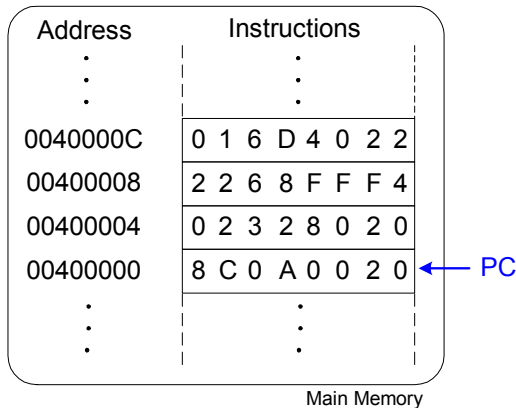
Assembly Code

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

Machine Code

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```

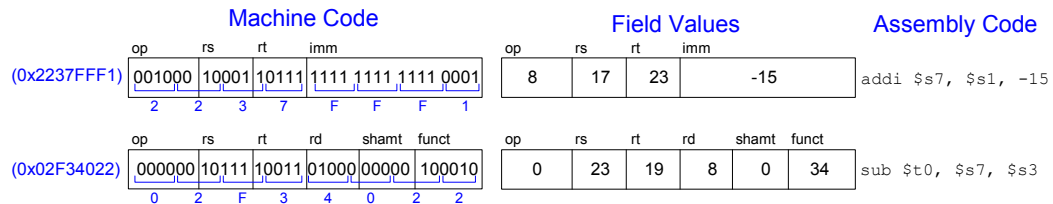
Stored Program



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 definen la operación
- Si no es 0
 - opcode determina la operación



Lenguajes de alto nivel

- Por ejemplo
 - C, C++, Java, Python, PHP, bash
 - Mayor nivel de abstracción
- Sentencias y construcciones comunes en lenguajes de alto nivel:
 - if/else
 - for
 - while
 - arrays
 - Llamados a función



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 ($i \neq j$) 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 # $s0 = pow, $s1 = x
// x tal que 2x = 128
int pow = 1;
int x    = 0;

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

Código ensamblador de MIPS

```
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;

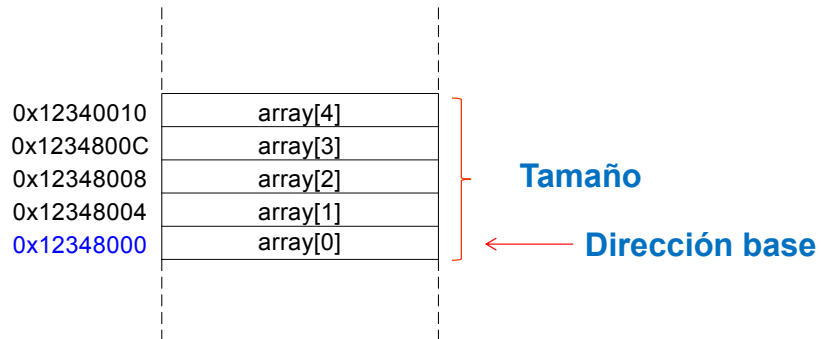
for (i=1; i < 101; i = i*2)
{
    sum = sum + i;
}
```

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      # si 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 sobrescribir 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



Convenciones: Ejemplo

Código C

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

```
void simple() {  
    return;  
}
```

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

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

Código ensamblador de MIPS

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

```
0x00401020 simple: jr    $ra
```



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
```



leaf

**Sobreescribe
3 registros**



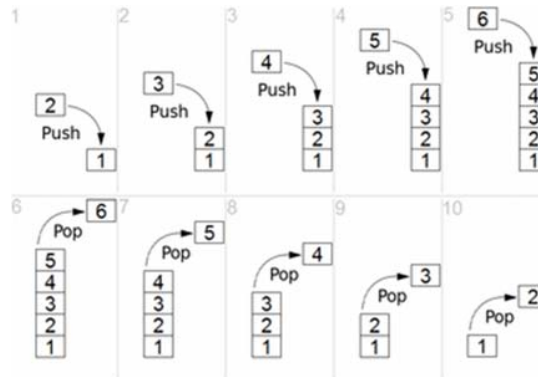
Resguardo de registros entre llamadas

- `diffofsums` sobrescribe 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

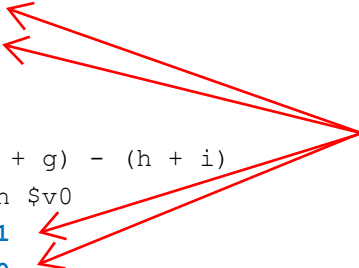
Address	Data		Address	Data	
7FFFFFFC	12345678	← $\$sp$	7FFFFFFC	12345678	
7FFFFFF8			7FFFFFF8	AABBCCDD	
7FFFFFF4			7FFFFFF4	11223344	← $\$sp$
7FFFFFF0			7FFFFFF0		
⋮	⋮		⋮	⋮	
⋮	⋮		⋮	⋮	

diffofsums revisado

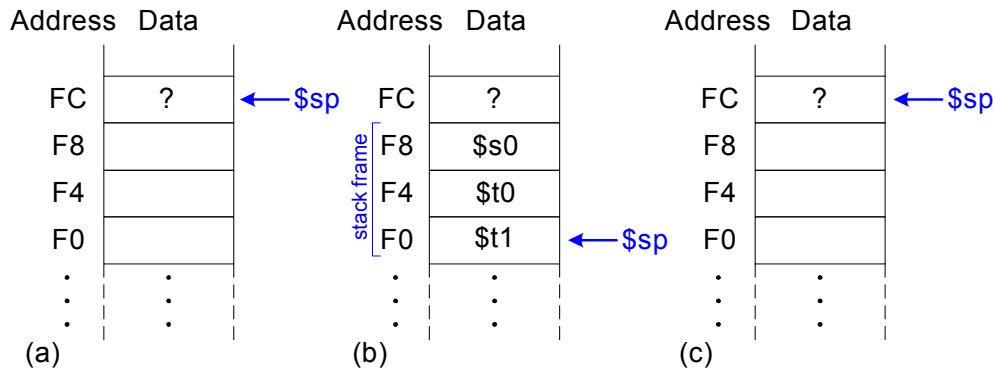
diffofsums:

```
    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
```

¿es necesario?



diffofsums revisado: pila





Pila en MIPS: convenciones

Preserved <i>Callee-Saved</i>	Nonpreserved <i>Caller-Saved</i>
\$s0-\$s7	\$t0-\$t9
\$ra	\$a0-\$a3
\$sp, \$fp	\$v0-\$v1



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
```



Pasaje por referencia (punteros)

Código C

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

Complica las cosas

```
void diffofsums(int f, int g, int h, int i, int *result) {  
    *result = (f + g) - (h + i);  
}
```


diffofsums revisado con punteros

```
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
    la    $s0, y        # carga en $s0 la dirección de y
    addi $sp, $sp, -4    # hace espacio en la pila para otro argumento
    sw    $s0, 0($sp)    # guarda el 5to argumento en la pila
    jal   diffofsums     # llamado a funcion
    ...

diffofsums:
    la    $t3, 4($sp)    # carga en $t3 el 5to argumento
    add $t0, $a0, $a1    # $t0 = f + g
    add $t1, $a2, $a3    # $t1 = h + i
    sub $t4, $t0, $t1    # $t4 = (f + g) - (h + i)
    sw    $t4, 0($t3)    # guarda $t4 en la dirección result
    jr    $ra            # return
```

Funciones anidadas



non leaf

```
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
```



Funciones recursivas!

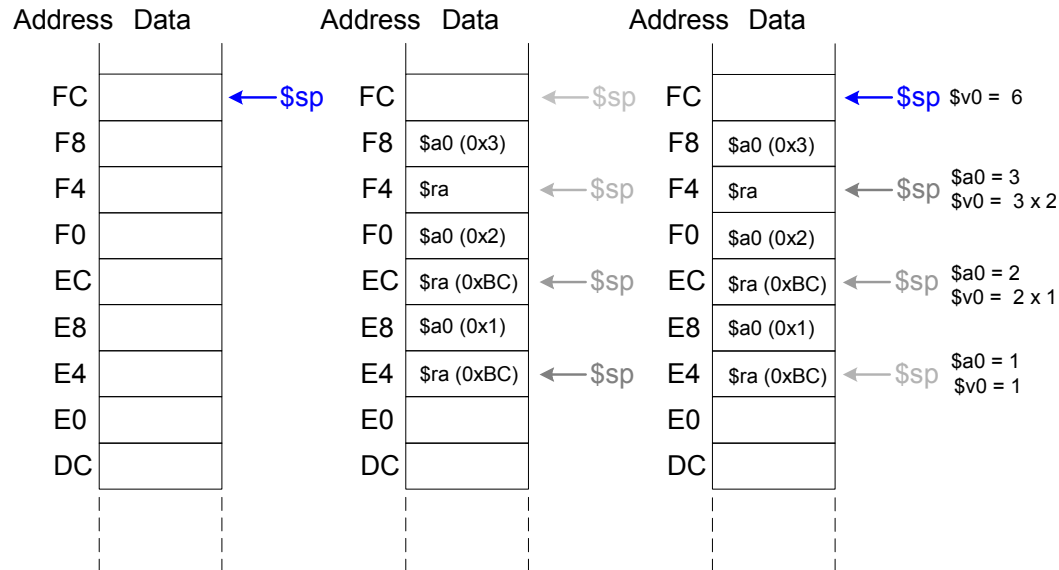
```
int factorial(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * factorial(n-1));  
}
```

Funciones recursivas en MIPS

```
0x90 factorial: addi $sp, $sp, -8 # hace lugar en la pila
0x94           sw  $a0, 4($sp)   # apila $a0
0x98           sw  $ra, 0($sp)   # apila $ra
0x9C           addi $t0, $0, 2
0xA0           slt  $t0, $a0, $t0 # a <= 1 ?
0xA4           beq  $t0, $0, else1 # no: go to else
0xA8           addi $v0, $0, 1    # si: return 1
0xAC           addi $sp, $sp, 8   # restaura $sp
0xB0           jr   $ra           # return
0xB4     else1: addi $a0, $a0, -1  # n = n - 1
0xB8           jal  factorial     # llamada recursiva
0xBC           lw   $ra, 0($sp)   # desapila $ra
0xC0           lw   $a0, 4($sp)   # desapila $a0
0xC4           addi $sp, $sp, 8   # restaura $sp
0xC8           mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC           jr   $ra           # return
```



Funciones recursivas: la pila

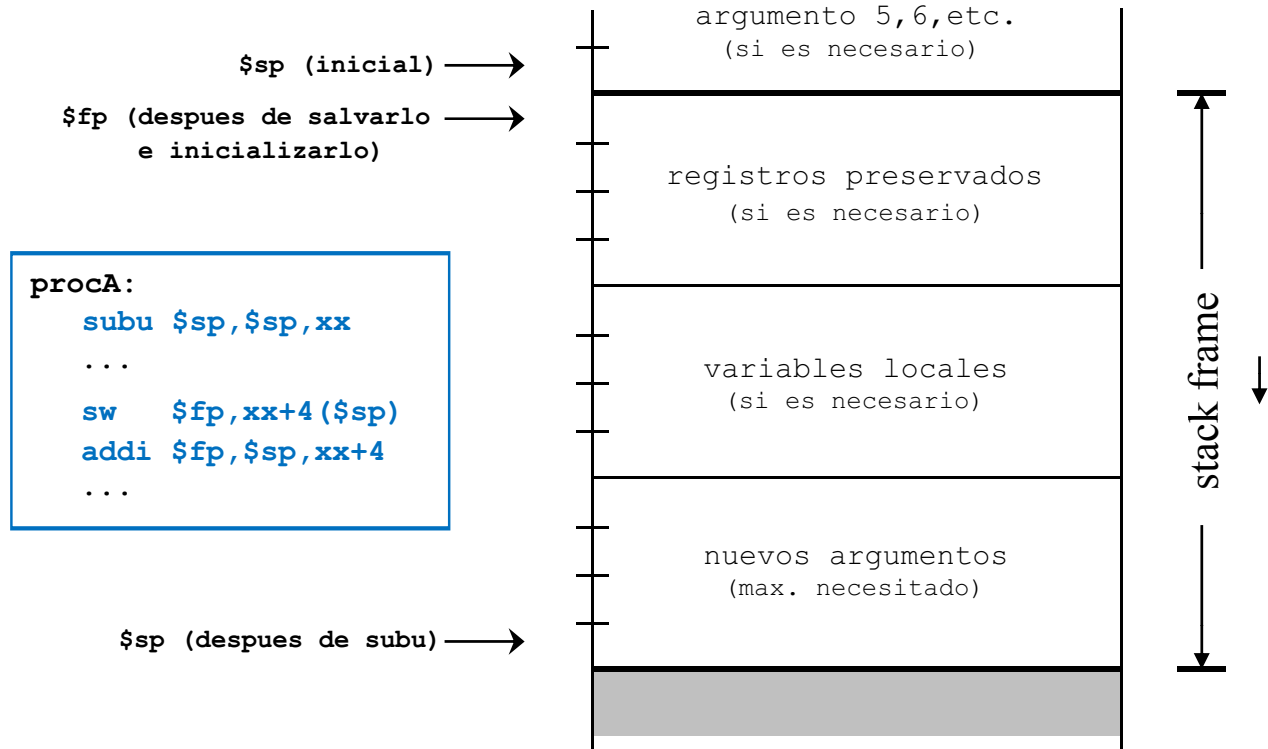




Uso del registro \$fp (frame pointer)

- El \$sp puede variar durante la ejecución de la subrutina
 - Cualquier referencia local a la pila se vuelve relativa
 - Se necesita una referencia estable dentro del ámbito de ejecución de la subrutina para toda la información contenida allí (scope)
- El stack frame es dicho segmento de pila
 - El \$fp apunta al inicio del stack frame
 - El \$fp no se mueve durante la ejecución de la subrutina (toda referencia a un dato en la pila desde el \$fp es fija)
 - No hay ventaja en usar el registro \$fp si \$sp no cambia durante la ejecución
 - Si no es necesario inicializar un stack frame \$fp se usa como \$s8
 - El stack frame se puede construir de diferentes formas
 - Adoptamos la convención que sigue a continuación

Bloque de activación (stack frame)





Resumen de funciones

• Función llamadora

- Poner argumentos en \$a0-\$a3 y los demás en la pila
- Guardar registros si es necesario
- jal funcion
- Restaurar registros si es necesario
- El resultado se encuentra en \$v0

• Función llamada

- Guardar registros si es necesario (\$fp, \$ra, \$s0-\$s7)
- Inicializar el stack frame si es necesario
- Ejecutar la función
- Poner el resultado en \$v0
- Restaurar registros y el stack si es necesario
- jr \$ra



Punto flotante (PF) en MIPS

- MIPS provee diferentes instrucciones para números en PF:
 - Aritméticas
 - Movimiento de datos (memoria y registros)
 - Saltos condicionales
- Las instrucciones PF trabajan con otro banco de registros:
 - Los registros se llaman \$f0 a \$f31 ("coprocessor 1")
 - \$f0 no es un registro especial (puede contener cualquier valor)
- Hay instrucciones para doble y simple precisión:
 - Una u otra, usan el mismo banco de registros
 - Los números FP de doble precisión se operan con los registros pares del banco de registros (\$f0, \$f2, ...)
 - Las instrucciones en simple precisión terminan con ".s" (ej. add.s)
 - Generalmente existe la correspondiente instrucción en doble precisión terminada con ".d"



Instrucciones aritméticas en PF

- | | |
|--------------------------|---------------------|
| • add.s \$fo, \$f1, \$f2 | \$fo := \$f1 + \$f2 |
| • sub.s \$fo, \$f1, \$f2 | \$fo := \$f1 - \$f2 |
| • mul.s \$fo, \$f1, \$f2 | \$fo := \$f1 * \$f2 |
| • div.s \$fo, \$f1, \$f2 | \$fo := \$f1 / \$f2 |
| • abs.s \$fo, \$f1 | \$fo := \$f1 |
| • neg.s \$fo, \$f1 | \$fo := -\$f1 |



Movimientos de datos en PF

- Instrucciones de transferencia de memoria:
 - `l.s $f0, 100($t2) // carga $f0 desde la dirección $t2+100`
 - `s.s $f0, 100($t2) // guarda $f0 en la dirección $t2+100`
- Movimiento de datos entre registros:
 - `mov.s $f0, $f2 // movimiento entre registros FP`
 - `mfc1 $t1, $f2 // mueve desde registros FP (no convierte)`
 - `mtc1 $t1, $f2 // mueve a registros FP (no convierte)`
- Conversión de datos:
 - `cvt.w.s $t1, $f2 // convierte de FP simple precisión a entero`
 - `cvt.s.w $t1, $f2 // convierte de entero a simple precisión FP`



Saltos condicionales en PF

- Los saltos condicionales se realizan en dos etapas:
 1. La comparación entre dos valores FP setea un código en un registro especial
 2. La instrucción de bifurcación salta dependiendo del valor del código
- Comparación
 - `c.eq.s $f2, $f4 // if $f2 == $f4 then code = 1 else code = 0`
 - `c.le.s $f2, $f4 // if $f2 <= $f4 then code = 1 else code = 0`
 - `c.lt.s $f2, $f4 // if $f2 < $f4 then code = 1 else code = 0`
- Conversión de datos:
 - `bc1f label // if code == 0 then jump to label`
 - `bc1t label // if code == 1 then jump to label`