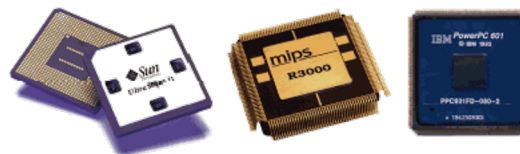


# Tema 2C: Ensamblador MIPS

## Fundamentos de Ordenadores y Sistemas Operativos

Mario Martínez Zarzuela  
marmar@tel.uva.es

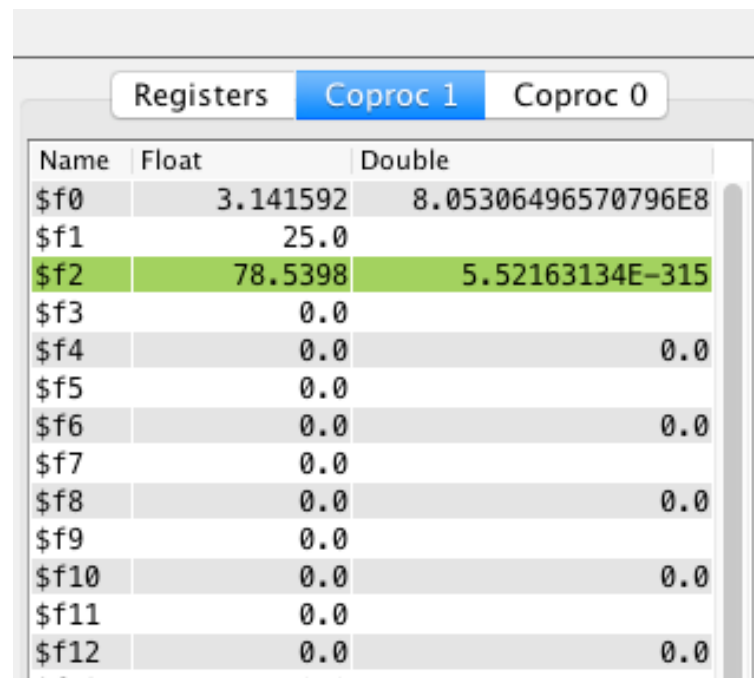


# Contenidos

- Operaciones en punto flotante
- Llamadas a procedimientos
- Tamaño de datos inmediatos
- Conclusiones

# Operaciones en punto flotante

- Coprocesador matemático CP1
  - Registros específicos tipo \$f
  - 32 registros de tamaño 4 bytes
    - Dato double ocupa un registro par + uno impar



Name	Float	Double
\$f0	3.141592	8.05306496570796E8
\$f1	25.0	
\$f2	78.5398	5.52163134E-315
\$f3	0.0	
\$f4	0.0	0.0
\$f5	0.0	
\$f6	0.0	0.0
\$f7	0.0	
\$f8	0.0	0.0
\$f9	0.0	
\$f10	0.0	0.0
\$f11	0.0	
\$f12	0.0	0.0

# Operaciones en punto flotante

- Carga/almacenamiento hacia/desde C1 desde/hacia memoria

<code>l.s</code>	<code>fd,imm(rs)</code>	<code>#load single</code>
<code>s.s</code>	<code>fd,imm(rs)</code>	<code>#store single</code>

- Operaciones con registros f

<code>mov.s</code>	<code>fd, fs</code>	<code>#copy fs → fd</code>
<code>add.s</code>	<code>fd, fs, ft</code>	<code># fd = fs + ft</code>
<code>sub.s</code>	<code>fd, fs, ft</code>	<code># fd = fs - ft</code>
<code>div.s</code>	<code>fd, fs, ft</code>	<code># fd = fs / ft</code>
<code>mul.s</code>	<code>fd, fs, ft</code>	<code># fd = fs * ft</code>

# Ejecuta el código y examina el contenido de registros y memoria

```
1      .data
2  A:    .float    2.5
3  B:    .float    5
4  C:    .float    0
5
6      .text
7  init:  .l.s      $f0, A($zero)
8         .l.s      $f1, B($zero)
9
10         .mul.s    $f2, $f0, $f1
11         .s.s      $f2, C($zero)
12  end:   .nop
```

The screenshot shows a MIPS simulator interface with three main panels:

- Text Segment:** A table showing the assembly code being executed. The instructions are: `lwc1 $f0, 0($zero)`, `lwc1 $f1, 4($zero)`, `mul.s $f2, $f0, $f1`, `s.s $f2, 0($zero)`, and `nop`. The addresses are 12288, 12292, 12296, 12300, and 12304 respectively.
- Data Segment:** A table showing memory values at addresses 0, 32, and 64. The values are: 0x40200000, 0x40a00000, and 0x41480000 respectively.
- Registers:** A table showing the state of registers \$f0 through \$f14. Register \$f2 is highlighted in green, showing a value of 0x41480000.

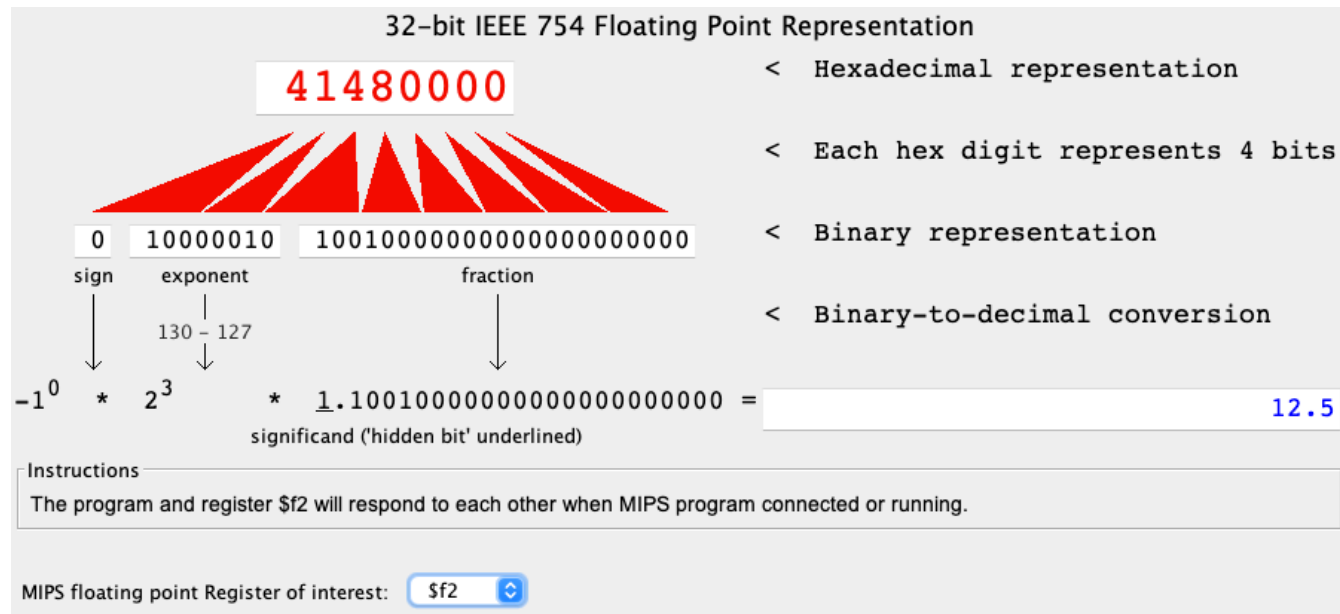
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	12288	0xc4000000	lwc1 \$f0, 0x00000000...	7: init: .l.s \$f0, A(\$zero)
<input type="checkbox"/>	12292	0xc4010004	lwc1 \$f1, 0x00000004...	8: .l.s \$f1, B(\$zero)
<input type="checkbox"/>	12296	0x46010082	mul.s \$f2, \$f0, \$f1	10: .mul.s \$f2, \$f0, \$f1
<input type="checkbox"/>	12300	0xe4020008	swc1 \$f2, 0x00000008...	11: .s.s \$f2, C(\$zero)
<input type="checkbox"/>	12304	0x00000000	nop	12: end: .nop

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
0	0x40200000	0x40a00000	0x41480000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
32	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
64	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Na...	Float	Double
\$f0	0x40200000	0x40a0000040200000
\$f1	0x40a00000	
\$f2	0x41480000	0x0000000041480000
\$f3	0x00000000	
\$f4	0x00000000	0x0000000000000000
\$f5	0x00000000	
\$f6	0x00000000	0x0000000000000000
\$f7	0x00000000	
\$f8	0x00000000	0x0000000000000000
\$f9	0x00000000	
\$f10	0x00000000	0x0000000000000000
\$f11	0x00000000	
\$f12	0x00000000	0x0000000000000000
\$f13	0x00000000	
\$f14	0x00000000	0x0000000000000000

# Representación de punto flotante en binario y hexadecimal

- Representación binaria según estándar IEEE754
- En MARS Tools → Floating Point Representation



# ¿Qué hacemos si tenemos el dato en el procesador?

- Por ejemplo si queremos realizar una división con decimales entre dos enteros que ya están en registros r
- Copiar valores desde/hacia procesador hacia/desde C1

`mtc1        rs, fd        # move to C1        rs → fd`

`mfc1        rd, fs        # move from C1    fs → rd`

- Conversión entre tipos de datos

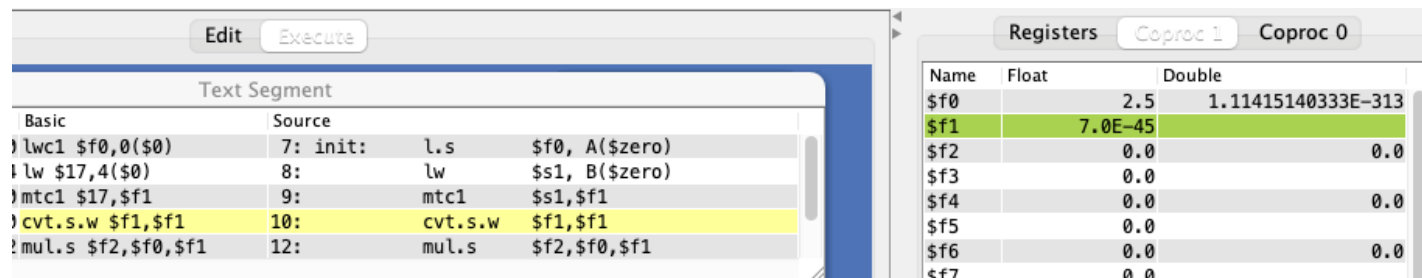
`cvt.s.w    fd, fs        # convert to single from word`

`cvt.w.s    fd, fs        # convert to word from single`

# Ejecuta el código y observa qué ocurre si comentas la línea 10

- ¿Qué valor toma \$f1 en la línea 9? Antes de hacer la conversión de tipo de dato con la instrucción `cvt.s.w`

```
1      .data
2  A:    .float    2.5
3  B:    .word     5
4  C:    .float    0
5
6      .text
7  init:  l.s      $f0, A($zero)
8         lw      $s1, B($zero)
9         mtc1    $s1, $f1
10        cvt.s.w $f1, $f1
11
12        mul.s   $f2, $f0, $f1
13        s.s     $f2, C($zero)
14  end:   nop
```



Text Segment	
Basic	Source
lwc1 \$f0,0(\$0)	7: init: l.s \$f0, A(\$zero)
lw \$t7,4(\$0)	8: lw \$s1, B(\$zero)
mtc1 \$t7,\$f1	9: mtc1 \$s1,\$f1
cvt.s.w \$f1,\$f1	10: cvt.s.w \$f1,\$f1
mul.s \$f2,\$f0,\$f1	12: mul.s \$f2,\$f0,\$f1

Registers		
Name	Float	Double
\$f0	2.5	1.11415140333E-313
\$f1	7.0E-45	
\$f2	0.0	0.0
\$f3	0.0	
\$f4	0.0	0.0
\$f5	0.0	
\$f6	0.0	0.0
\$f7	0.0	



# Operaciones en punto flotante con datos tipo double

- ¿Qué cambia respecto a operaciones con tipo float?
  - Sustituir .s (single precision) por **.d** (double precision)
  - Registros f origen y destino tienen que ser registros **pares**
- Carga/almacenamiento hacia/desde C1 desde/hacia memoria

<b>l.d</b>	fd, imm(rs)	#load single
<b>s.d</b>	fd, imm(rs)	#store single

- Operaciones con registros f

<b>mov.d</b>	fd, fs	#copy fs → fd
<b>add.d</b>	fd, fs, ft	# fd = fs + ft
<b>sub.d</b>	fd, fs, ft	# fd = fs - ft
<b>div.d</b>	fd, fs, ft	# fd = fs / ft
<b>mul.d</b>	fd, fs, ft	# fd = fs * ft

# ¿Qué hacemos si tenemos que convertir entre tipos de datos?

- Conversión entre tipos de datos

<code>cvt.w.d</code>	<code>fd, fs</code>	<code># convert to word from double</code>
<code>cvt.d.w</code>	<code>fd, fs</code>	<code># convert to double from word</code>

<code>cvt.s.d</code>	<code>fd, fs</code>	<code># convert to single from double</code>
<code>cvt.d.s</code>	<code>fd, fs</code>	<code># convert to double from single</code>

# Laboratorio

- Modifica el código anterior para que A y C sean tipo `.double` en lugar de `.float`

# Operaciones E/S con números en punto flotante

Llamada a invocar	Configuración \$v0	Argumento \$f12	Resultado \$f0
print float	2	\$f12: float a imprimir	
print double	3	\$f12: double a imprimir	
read float	6		\$f0: float leído
read double	7		\$f0: double leído

**syscall**

# Operaciones en punto flotante

```
1 #include <stdio.h>
2
3 void main()
4 {
5     float radio=5;
6     double pi=3.141592653589793238462643383279;
7     double perimetro=0;
8     char* textoPerimetro="Perimetro: ";
9
10    perimetro=(double) 2*pi*(double) radio;
11
12    printf("%s",textoPerimetro);
13    printf("%f",perimetro);
14 }
```

```

radio: .data
       .float 5
pi:    .double 3.14159265358973238
perim: .double 0
texto: .asciiz "\nPeri'metro: "
       .text
main:  l.s      $f1,radio($zero)
       cvt.d.s $f2,$f1
       l.d      $f0,pi($zero)

       mul.d    $f4,$f0,$f2      #pi*r
       addi     $s0,$zero,2      #li $s0,2
       mtc1     $s0,$f3
       cvt.d.w  $f6,$f3
       mul.d    $f6,$f6,$f4

       s.d      $f6,perim($zero)

       la       $a0,texto($zero)
       addi     $v0,$zero,4      #li $v0,4
       syscall                      #print string

       mov.d    $f12,$f6
       addi     $v0,$zero,3      #li $v0,3
       syscall                      #print double

fin:    nop
```

# Laboratorio

- Ejercicio 2.10
  - Modifica el código del ejemplo anterior para que el tipo de datos de “pi” y “perim” sea float.
    - pi:                    .float    3.141592
    - perim:                .float    0
  - El código final deberá simplificarse

# Laboratorio

- Ejercicio 2.11
  - Escribe un programa que calcule el área de una circunferencia.
    - El radio se almacenará como float y se solicitará por teclado.

# Contenidos

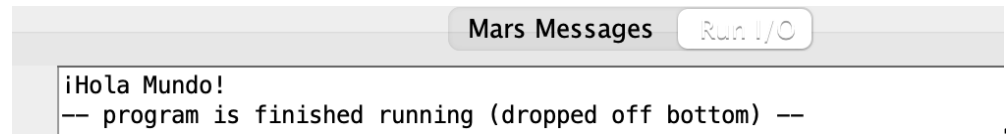
- Operaciones en punto flotante
- Llamadas a procedimientos
- Tamaño de datos inmediatos
- Conclusiones



# Intenta averiguar cuál es el flujo en este programa

- Pistas:
  - Todas las instrucciones que comienzan por `j` son de salto
  - La función `func_imp` no devuelve ningún valor

```
1      .data
2  mensaje: .asciiz "¡Hola Mundo!"
3      .text
4  init:   la    $a0, mensaje
5          jal   func_imp
6          j     end
7  func_imp: li    $v0, 4
8            syscall
9            jr    $ra
10 end:     nop
```



# Instrucciones para funciones o procedimientos

- Para invocar la función:
  - Colocar argumentos para la función
    - Utilizar registros `$a0-$a3`
  - Llamar a la función: instrucción de salto tipo J (`jump`)  
`jal addr #jump and link`
    - Almacena en `$ra` la dirección de la siguiente instrucción
- En la función invocada:
  - Si debe devolver algún valor a su salida
    - Utilizar registros `$v0, $v1`
  - Salir de la función: instrucción de salto tipo R (`r`)  
`jr rs #jump to address in register`

# Ejecuta el código anterior

- Ejecuta el código anterior en MARS paso por paso
  - Familiarízate con su funcionamiento
- Averigua dónde apunta la dirección de memoria almacenada en \$ra
  - ¿Al ejecutar qué línea del programa toma el valor \$ra?
  - ¿Podría producirse un bucle infinito si comentas la instrucción de la línea 6?

# Función que devuelve un valor

- Valor devuelto en \$v0
- ¿Cuál es el resultado almacenado en area?

```
1      .data
2  base:  .word    3
3  altura: .word    8
4  area:  .word
5      .text
6  init:  lw      $a0, base($zero)
7         lw      $a1, altura($zero)
8         jal     calcArea
9         sw      $v0, area($zero)
10        j      end
11  calcArea: #Calcula area del triangulo base*altura/2
12         mul     $t0, $a0, $a1
13         li      $t1, 2
14         div     $t0, $t1
15         mflo    $v0
16         jr      $ra
17  end:   nop
```

# Compara estos dos programas

- ¿Cuál es el resultado almacenado en suma en cada caso?

```
1      .data
2  valor: .word      3
3  suma:  .word      1
4      .text
5  main:  lw    $s0,suma($zero)
6         lw    $a0,valor($zero)
7         jal   dupli
8         add   $s0,$s0,$v0
9         sw    $s0,suma($zero)
10        j     fin
11  dupli: #Duplica el número recibido
12         li    $s0,2
13         mul   $s0,$a0,$s0
14         move  $v0,$s0
15         jr    $ra
16  fin:  nop
```

```
int func(int);

void main()
{
    int suma = 1;
    int valor = 3;
    suma = suma + duplica(valor);
}

int duplica(int v)
{
    int aux = 2*v;
    return aux;
}
```

# Compara estos dos programas

En el programa en C: 7, en MIPS: 12

```
1      .data
2  valor: .word 3
3  suma:  .word 1
4      .text
5  main:  lw    $s0,suma($zero)      $s0 ← 1
6         lw    $a0,valor($zero)
7         jal   dupli
8         add   $s0,$s0,$v0          $s0 ← 12
9         sw    $s0,suma($zero)
10        j     fin
11  dupli: #Duplica el número recibido
12        li    $s0,2                $s0 ← 2
13        mul   $s0,$a0,$s0          $s0 ← 6
14        move  $v0,$s0
15        jr    $ra
16  fin:  nop
```

# ¿Cómo evitamos este problema?

- Función invocadora e invocada trabajan conjuntamente para garantizar que no se sobrescriben valores necesarios en registros
- La función invocada
  - Responsable de salvar (y restaurar) **\$s0-\$s7**, \$ra si los va a sobrescribir
- La función invocadora
  - Responsable de salvar **\$t0-\$t9**, \$a0-\$a3, **\$v0-\$v1** antes de invocar a la otra función
  - Necesario si contienen valores que deban ser utilizados después de la llamada

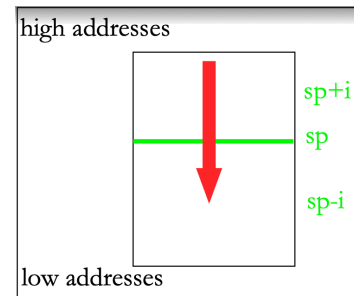
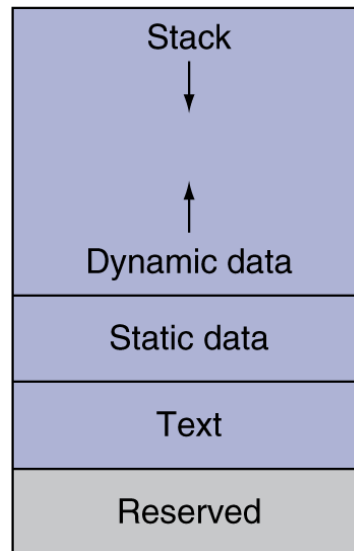
# ¿Cómo evitamos este problema?

- La función invocadora utiliza la pila
  - Estructura LIFO (Last In First Out)
    - Salvaguardar registros (*register spilling*)
  - **\$sp** puntero de pila: dirección último dato almacenado
    - Restar  $i$  a  $\$sp$  para reservar  $i$  bytes en la pila
    - Sumar  $i$  a  $\$sp$  para liberar  $i$  bytes de la pila

$\$sp \rightarrow 7fff\ fffc_{hex}$

$\$gp \rightarrow 1000\ 8000_{hex}$   
 $1000\ 0000_{hex}$

$pc \rightarrow 0040\ 0000_{hex}$   
 $0$





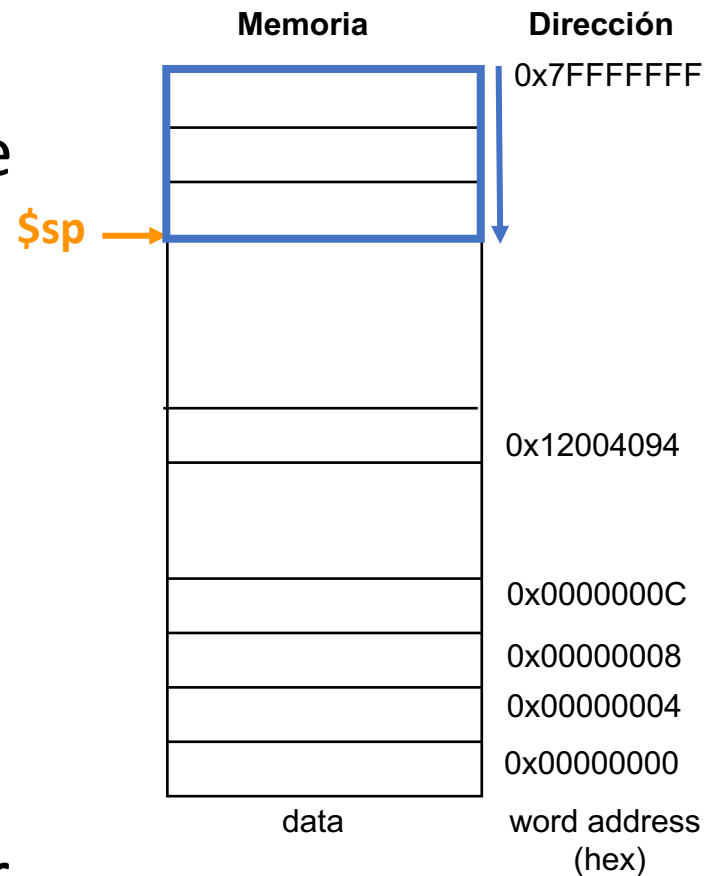
# Funciones

```
valor: .data
suma: .word 3
      .text
main: lw $s0,suma($zero)
      lw $a0,valor($zero)
      jal dupli
      add $s0,$s0,$v0
      sw $s0,suma($zero)
      j fin
dupli: #Duplica el número recibido
      addi $sp, $sp, -4      #Reservar espacio de pila
      sw $s0, 0($sp)        #Salvar $s0 en la pila
      li $s0,2
      mul $s0,$a0,$s0
      move $v0,$s0
      lw $s0, 0($sp)        #Restaurar $s0
      addi $sp, $sp, 4      #Liberar espacio de pila
      jr $ra
fin: nop
```

Si es necesario guardar más de un registro tipo \$s repetir las operaciones

# Operaciones con la pila

- Salvaguardar en la pila el estado de los registros antes de la llamada al invocado
  - Registros tipo \$s a sobrescribir
  - No es necesario respetar contenido de registros \$t
  - Salvaguardar \$ra si el procedimiento llama a otro anidado
- Devolver los registros a su estado anterior antes de devolver el control al invocador



# Laboratorio

- Ejercicio 2.12

- Escribe el equivalente del siguiente programa en código MIPS.
- Dentro de la función, emplea los registros \$s0 y \$s1 para las sumas parciales

```
int func(int g,int h,int i,int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

```
void main()
{
    int A[]={20,11,02,20,18,5,-3,6,452,0};
    int resul=37;

    resul+=func(A[0],A[1],A[2],A[3]);

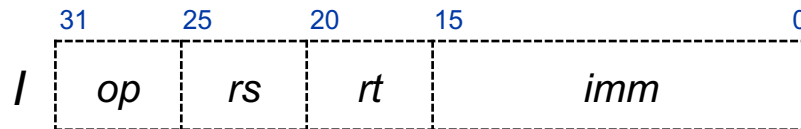
}
```

# Contenidos

- Operaciones en punto flotante
- Llamadas a procedimientos
- Tamaño de datos inmediatos
- Conclusiones

# Dato inmediato en instrucciones tipo I

- Las instrucciones en MIPS ocupan 32 bits
- En instrucciones tipo-I el dato inmediato **imm** ocupa 16 bits
  - Almacenar un dato unsigned entre 0 y  $2^{16}-1$  (0 a 65535)
- Veremos los campos de las instrucciones máquina en el Tema 3



# Dato inmediato en instrucciones tipo I

- ¿Qué ocurre si tengo que cargar en el registro un dato que no cabe en los primeros 16 bits?
  - Combinación de instrucciones: cargar parte alta del registro (bits 16 a 31) con instrucción **lui**

Diagram illustrating the upper immediate field in a MIPS instruction. The instruction is 32 bits long, with the upper immediate field (bits 16-31) and the lower immediate field (bits 0-15). The upper immediate field is labeled 'imm' and the lower immediate field is labeled 'imm'.

- Y luego completar parte baja haciendo operación con otra instrucción: ori, addi, addu

**ori**      **rt, rs,** **imm**      #**rt** ← **rs** (**or**) **imm**

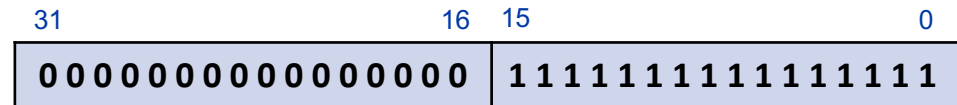
31                          16    15                          0

**imm**                          **imm**

# Dato inmediato en instrucciones tipo I

- Ejemplos:

- `li $s0, 65535`



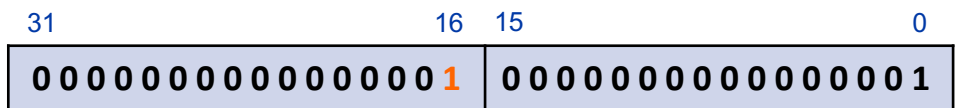
- `li $s0, 65537`

(65537 = 65536 + 1)

→ `lui $at, 1`



→ `ori $s0, $at, 1`



- `li $s0, 65538`

→ `lui $at, 1`



→ `ori $s0, $at, 2`



# Laboratorio

- Prueba a cargar el valor 65538 en un registro con la instrucción li y observa en qué instrucciones se convierte dependiendo de la configuración de la memoria en settings → memory configuration



# Acceso a etiquetas en direcciones altas de memoria

- Prueba a ensamblar con dos configuraciones distintas de la memoria:
  - 1) Settings → Memory configuration → Compact, Data at Address 0
  - 2) Settings → Memory configuration → Default

SEGMENTO DE DATOS			
1		.data	
2	var1:	.word	14
3	var2:	.word	-6
4			
5		.text	
6	main:	lw	\$s1, var1(\$zero)
7		lw	\$s2, var2(\$zero)
8			
9		add	\$s1, \$s1, \$s2
0		sw	\$s1, var1(\$zero)
1			
2	end:	nop	
-			
SEGMENTO DE INSTRUCCIONES			

# Acceso a etiquetas en direcciones altas de memoria

- 1) .data empieza en dirección 0x00000000

Text Segment				Labels	
Address	Code	Basic	Source	Label	Address ▼
0x00003000	0x8c110000	lw \$17,0x00000000(\$0)	6: main: lw \$s1, var1(\$zero)	T2AprimerProg.asm	
0x00003004	0x8c120004	lw \$18,0x00000004(\$0)	7: lw \$s2, var2(\$zero)	end	0x00003018
0x00003008	0x02328820	add \$17,\$17,\$18	8: add \$s1,\$s1,\$s2	main	0x00003000
0x0000300c	0xac110000	sw \$17,0x00000000(\$0)	9: sw \$s1,var1(\$zero)	var2	0x00000004
0x00003010	0x3c010001	lui \$1,0x00000001	10: li \$s0,65536	var1	0x00000000
0x00003014	0x34300000	ori \$16,\$1,0x00000000			
0x00003018	0x00000000	nop	11: end: nop		

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x0000000e	0xfffffffffa	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

- 2) .data empieza en dirección 0x10010000 (268500992)

Text Segment				Labels	
Address	Code	Basic	Source	Label	Address ▼
0x00400000	0x3c011001	lui \$1,0x00001001	6: main: lw \$s1, var1(\$zero)	T2AprimerProg.asm	
0x00400004	0x00200821	addu \$1,\$1,\$0		var2	0x10010004
0x00400008	0x8c310000	lw \$17,0x00000000(\$1)		var1	0x10010000
0x0040000c	0x3c011001	lui \$1,0x00001001	7: lw \$s2, var2(\$zero)	end	0x00400030
0x00400010	0x00200821	addu \$1,\$1,\$0		main	0x00400000
0x00400014	0x8c320004	lw \$18,0x00000004(\$1)			
0x00400018	0x02328820	add \$17,\$17,\$18	8: add \$s1,\$s1,\$s2		
0x0040001c	0x3c011001	lui \$1,0x00001001	9: sw \$s1,var1(\$zero)		

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x0000000e	0xfffffffffa	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

# Contenidos

- Operaciones en punto flotante
- Llamadas a procedimientos
- Tamaño de datos inmediatos
- **Conclusiones**

# Conclusiones

- Necesito un emulador de la arquitectura MIPS para ejecutar código MIPS en ensamblador en un ordenador (arquitectura x86-64)
- Códigos en ensamblador requieren de más líneas que en lenguajes de alto nivel
- Para realizar operaciones aritmético-lógicas es necesario cargar previamente los operandos en registros: desde memoria o utilizando datos inmediatos en las instrucciones (imm)
- Posteriormente copiar el resultado a la memoria

# Conclusiones

- Para cargar/salvar en memoria es necesario atender al tipo de dato (su tamaño) y calcular la dirección de lectura/escritura en bytes
- Las instrucciones de salto condicional (beq, bne), salto no condicional (j, jal, jr) y las pseudoinstrucciones (bgt, bge,...) permiten realizar un control de flujo del programa
- Las llamadas al sistema permiten interactuar con los periféricos, pidiendo permiso al sistema operativo a través de syscall

# Conclusiones

- Para realizar operaciones en punto flotante (ya sea con precisión simple o double) en MIPS, es necesario utilizar el coprocesador matemático C1
- Las funciones o procedimientos deben contemplar salvaguardar el estado de registros \$s que puedan sobrescribir, haciendo uso de la pila
- El espacio para datos inmediatos en la instrucción es de 16 bits (la mitad del tamaño de palabra). Necesaria instrucción lui para cargar valores que no quepan en esos 16 bits.