

Información importante y Todas las Convenciones : RISC - V

INDEX “NO ACTUALIZADO” rehacer al final

Información importante : desde Proceso de compilación

¿Cuál es el proceso de compilación? L6 (Proceso de compilación)

¿Qué es la compilación? L6

¿Qué es el ensamblador? Agregar resto de preguntas, modificar index

Convenciones :

Conjunto de instrucciones RISC-V

Introducción al RV32I L8

¿Qué son las instrucciones?

¿Qué es el lenguaje ensamblador?

¿Qué son los registros?

¿Qué son las localidades de memoria?

¿Cuáles son los tipos de instrucciones?

Operadores Aritméticos y Lógicos : I7

Asignación de constantes e inicialización de variables : A8, E8

Carga de variables

Algoritmo de inicialización de registros : F9, A9

Uso de lui, addi, slli, mul, div, srl, rem, xori, andi, slti

Multiplicación por potencias de dos : F10, A10, E10

Fórmulas (add, addi, slli, sub)

Negación con xori

Compilación de if

Compilación de while, do while, y for

Organización de la memoria

Funciones Hoja, Anidadas, y Recursivas : Creación de marco

Ejemplos

Revisión de instrucciones

Desensamble

Información Importante :

¿Cuál es el proceso de compilación? L6 (Proceso de compilación)

Definición : Conjunto de reglas y algoritmos que se ejecutan con el fin de convertir un código escrito en un lenguaje de alto nivel al lenguaje máquina de una familia de procesadores.

Pasos del proceso :

1.- Preprocesamiento 2.- Compilación 3.- Ensamble 4.- Enlace

¿Qué es el procesamiento?

Es la etapa del proceso de compilación en la que se resuelven todas las directivas en cada programa, las directivas son las cabeceras con símbolo de almohadilla, o gato (#)

Ejemplos :

#define, #include, #if, y también existen directivas de uso exclusivo de compilador. El preprocesador también elimina los comentarios. El preprocesador incrusta código donde se usan macros, puede recibir argumentos

¿Qué es la compilación?

La compilación es el proceso de compilación, es decir, la conversión entre lenguajes de alto nivel y de ensamblador correspondiente al de la arquitectura en cuestión (RISK-V, MIPS, ARM, Intel x86, Sparck, etc).

¿Qué es el compilador?

El compilador es un programa muy complejo capaz de optimizar el código del desarrollador. Entre las optimizaciones que el compilador hace está el cálculo de operaciones o la sustitución de instrucciones, así como el cambio de orden, entre otras.

¿Qué es el ensamblador?

Proceso de traducción de lenguaje ensamblador a lenguaje máquina (objeto). (se hace con objdump) Durante el ensamblado se procesan macros a nivel ensamblador y se definen los símbolos que el enlazador. A diferencia de la compilación el proceso es directo, sin embargo requiere conocimiento de la arquitectura objetivo.

En ésta etapa se definen los símbolos que el enlazador necesita para crear un ejecutable o código ensamblador, cualquier código máquina es único.

¿Qué es el código máquina?

El código máquina u objeto, es el resultante de la compilación del código fuente. Se escribe en hexadecimal o binario, y se puede hacer legible con el comando

`objdump -d main.o > main.dump`
en el compilador.

¿Qué es el enlace?

Concatenado del código máquina de funciones a código fuente : gcc → print → fuente

Convenciones :

Conjunto de instrucciones RISC-V

Introducción al RV32I L8

¿Qué son las instrucciones?

Las instrucciones son el vocabulario de comandos que son entendidos por una computadora. Este conjunto de instrucciones define de manera abstracta a una computadora ya que explícitamente especifica el hardware que un procesador debe tener. Por ejemplo, si en el conjunto se especifica la operación de la suma, debe existir el hardware para efectuarla. Dicho hardware se encuentra en la unidad de aritmética y lógica o ALU por sus siglas en inglés (Arithmetic and Logic Unit). Éstas instrucciones se definen por la arquitectura, RISC-V funciona con una ISA (Instruction Set Architecture).

¿Qué es una ISA?

Una arquitectura de conjunto de instrucciones, es un modelo abstracto de una computadora que define cómo el CPU es controlado por el software. La ISA actúa como una interfaz entre el hardware y el software, especificando qué puede hacer el procesador y cómo lo hace.

El ISA proporciona la única forma a través de la cual un usuario puede interactuar con el hardware. Puede verse como un manual del programador porque es la parte de la máquina que es visible para el programador de lenguaje ensamblador, el escritor del compilador y el programador de aplicaciones.

El ISA define los tipos de datos admitidos, los registros, cómo el hardware administra la memoria principal, características clave (como la memoria virtual), qué instrucciones puede ejecutar un microprocesador y el modelo de entrada/salida de múltiples implementaciones de la ISA. El ISA se puede ampliar agregando instrucciones u otras capacidades, o agregando soporte para direcciones y valores de datos más grandes.

Ejemplos de ISA son CISC y RISC, que CISC son Complex Instruction Set Computer y ejemplos de implementaciones de CISC son x86 de Intel, de RISC (Reduced Instruction Set Computer) está ARM, MIPS, RISC-V.

¿Qué es el lenguaje ensamblador?

El lenguaje ensamblador es una representación legible del lenguaje máquina. Esto se debe a que a cada instrucción en ensamblador le corresponde una única instrucción en código máquina.

¿Por qué se llama RISC?

Porque el conjunto de instrucciones no solo es menor, aunque el tamaño de instrucciones sea mayor, también son más simples, ocupan menos hardware y menos microcódigo, y entonces incrementa la velocidad de reloj. La idea de RISC viene del descubrimiento de John Cocke de IBM Research, sobre que el 20% de las instrucciones de un ordenador podían realizar el 80% del trabajo.

¿Qué y cuáles son los operandos de RISC-V?

Un operando es un elemento al que se le aplica una operación, entonces los operandos en RISC-V son los registros, la memoria de datos, y las constantes inmediatas.

Los procesadores RISC-V poseen 32 registros, algunos de propósito específico. El espacio de direcciones de memoria es de 4 GiB. Las constantes tienen su código binario y como no necesitas 2 ubicaciones de memoria, sólo depende de las instrucciones. Las constantes se codifican en complemento a 2 y se encuentran integradas en las instrucciones.

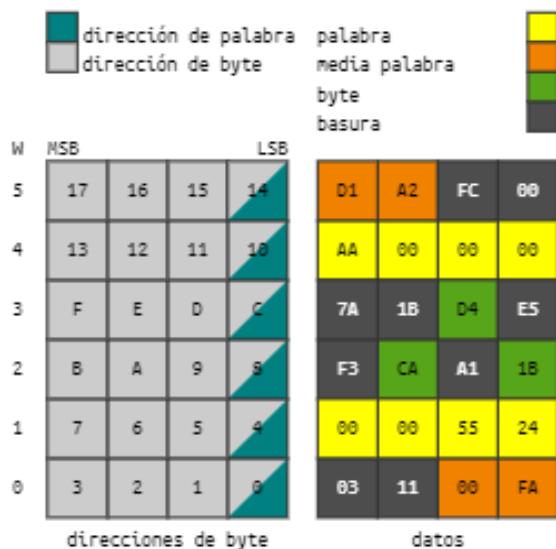
Registro	Nombre	Descripción
x0	zero	constante cero
x1	ra	dirección de retorno
x2	sp	apuntador de pila
x3	gp	apuntador global
x4	tp	apuntador de hilo
x5-x7	t0-t2	datos temporales
x8	fp/s0	apuntador de marco
x9	s1	datos persistentes
x10-x17	a0-a7	argumentos de función
x18-x27	s2-s11	datos persistentes
x28-x31	t3-t6	datos temporales

¿Qué son los registros?

Los registros son el almacenamiento directo del procesador, es decir, el procesador trabaja directamente con los registros.. Los operandos se guardan en registros guardados dentro del archivo de registros.

Éstos almacenan datos de 32 bits. En hardware se implementan como registros de carga paralela. En RISC - V (RV32I) hay los registros de la imagen a la izquierda.

¿Qué son las localidades de memoria de datos?



Las localidades de memoria de datos son los espacios donde se guarda información de tamaño de un byte por dirección, en RISC-V hay 2^{32} bytes (4 GiB) de direcciones y datos. RISC-V trabaja con codificación little-endian, significando que el byte más significativo de una palabra se almacena en la dirección más alta. RISC tiene huecos, funciona así para ser rápida, en RISC eso implica que las direcciones de palabras son siempre múltiplos de 4.

Las palabras siempre están alienadas a direcciones múltiplo de cuatro.

¿Qué son y cuáles son las características de las constantes imm?

Son valores numéricos integrados dentro de las instrucciones.

Su rango es de $[-2^{11}, 2^{11} - 1]$ o [-2048, 2047], se leen directamente desde la instrucción. Pueden escribirse en binario, octal, decimal y hexadecimal, y se utilizan para traducir inicializaciones de variables.

¿Cuáles son los tipos de instrucciones?

Tipo	Formato	Descripción
R	mn rd, rs, rt	$rd \leftarrow mn(rs, rt)$
I	mn rd, rs, imm12	$rd \leftarrow mn(rs, imm12)$
S	mn rt, imm12(rs)	$M[imm12 + rs] \leftarrow rt$
B	mn rs, rt, imm13	$PC \leftarrow PC + imm13 \text{ if } mn(rs, rt)$
U	mn rd, imm20	$rd \leftarrow imm20 \ll 12$
J	mn rd, imm21	$rd \leftarrow PC + 4, PC \leftarrow PC + imm21$

(RV32I Base Integer Instructions)

- **Tipo R:** 3 registros, destino (rd) y 2 fuentes (rs y rt)
add, sub, xor, or, and, sll, srl, slt, sltu :
- **Tipo I:** constantes inmediatas, tiene 3 registros y imm12 es que la constante es máximo de 12 bits. (aritméticas):
addi, xori, ori, andi, slli, srli, srai, slti, sltiu
(acceso a memoria):
lb, lh, lw, lbu, lhu +
(salto incondicional): jalr
+ ecall & ebreak
- **Tipo S:** acceso a memoria para guardar : (sw word sh half y sb byte).
- **Tipo B:** para hacer saltos, branch :
beq, bne, blt, bge, bltu, bgeu
- **Tipo U:** para trabajar con la parte superior de bits en una palabra
lui, auipc
- **Tipo J:** para saltar entre funciones en un programa o entre archivos
jal

Asignación de constantes e inicialización de variables : A8, E8

1. Inicialización mediante la instrucción addi (identidad $x + 0 = x$).

```
#a = 3;  
addi t0 zero 3  
nop
```

2. Restablecimiento de una variable mediante la instrucción xor (identidad $x \oplus x = 0$).

```
#a = 0;  
addi t0 zero 1  
xor t0 t0 t0  
nop
```

3. Inicialización de una variable mediante la pseudoinstrucción li.

```
#a = 5;  
li t0 5  
nop
```

4. Restablecimiento de una variable mediante la instrucción or (identidad $0 | 0 = 0$).

```
#a = 0;  
addi s0 zero 0  
or s0 s0 s0  
nop
```

5. Inicialización de una variable mediante la instrucción ori (identidad $x | 0 = x$).

```
#a = 10;  
addi t0 zero 1  
ori t0 t0 0  
nop
```

6. Restablecimiento de una variable mediante la instrucción and (identidad $0 \& x = 0$).

```
#a = 0;  
addi t0 zero 2  
and t0 zero t0  
nop
```

7. Inicialización de una variable con un número negativo mediante la instrucción addi (identidad $x + 0 = x$).

```
#a = -9;  
addi t0 zero -9  
nop
```

8. Restablecimiento de una variable mediante la instrucción and (identidad $0 \& 0 = 0$).

```
#a = 0;  
addi t0 zero 0  
and t0 t0 t0
```

9. Inicialización de una variable con una constante fuera del rango $[-(2^{11}), (2^{11})-1]$ mediante las instrucciones lui y addi.

```
#a = 4100;
lui t0 0x1
addi t0 t0 0x4
nop
```

- 1.- Conversión de 4100 a hex : 0x1004
 2.- Separar 0x1 de 0x1004, y si en la parte inferior en posición dos (derecha a izquierda) hay un número $\Rightarrow 8$, sumar 1 al valor de 0x1, y el resto del valor en hex se suma con addi

10. Inicialización de una variable con una constante, mediante las instrucciones lui y addi, que es afectada por el extensor de signo.

```
#a = -6144;  FFFF E800
lui t0 0xFFFFF
addi t0 t0 0xFFFFF800
nop

#a = 6144;    1800
lui t0 0x2
addi t0 t0 0xFFFFF800
nop
```

- 1.- Conversión de -6144 a hex : FFFF E800
 2.- Separar 0x1 de 0x1004, y si en la parte inferior en posición dos (derecha a izquierda) hay un número $\Rightarrow 8$, sumar 1 al valor de 0x1, y el resto del valor en hex se suma con addi

Algoritmo de inicialización de registros : F9, A9

Algoritmo

Input: a 32-bit constant K

Procedure Load Immediate

Begin

```
if  $K_{10} \in [-2048, 2047]$  then
     $r_d \leftarrow 0 + K$ 
else
     $lo \leftarrow \text{and}(K_{16}, (0000FFFF)_{16})$ 
     $hi \leftarrow \text{srl}(K, 12)$ 
    if  $lo \geq (800)_{16}$  then
         $hi \leftarrow hi + 1$ 
         $lo \leftarrow \text{or}(lo, (FFFFF000)_{16})$ 
    end if
     $r_d \leftarrow \text{sll}(hi, 12)$ 
     $r_d \leftarrow r_d + lo$ 
endif
```

End

Comments

K_{10} is the decimal value of K

K_{16} is the hexadecimal value of K

Bitwise AND function

Bit shifting to the right

Bitwise OR function

Bit shifting to the left

Operadores Aritméticos y Lógicos : (A9, E9), A10, E10

- Suma de variables :

```
#int a = 5000, b = -12500, c;  
#c = a + b;  
  
lui s1 0x00001 #5000  
addi s1 s1 0x388  
lui s2 0xFFFFD #-12500  
addi s2 s2 0xFFFFFFF2C  
add s3 s1 s2 #-7500  
nop  
  
#int a = 51234, b = -5465, c = 100, d = 20, e;  
#e = a + b + c + d;  
  
lui s1 0xD  
addi s1 s1 0xFFFFF822  
lui s2 0xFFFFF  
addi s2 s2 0xFFFFFAA7  
addi s3 zero 100  
addi s4 zero 20  
add t0 s1 s2  
add t0 t0 s3  
add s5 t0 s4  
nop
```

- Suma y Resta

```
#int a = 4998, b = -7876, c = -10000, d;  
#d = a + b - c;  
  
lui s1 0x00001  
addi s1 s1 0x386  
lui s2 0xFFFFE  
addi s2 s2 0x13C  
lui s3 0xFFFFE  
addi s3 s3 0xFFFFF8F0  
add t0 s1 s2  
sub s4 t0 s3  
nop
```

- Suma y Multiplicación

```
#int a = 50, b = -500, c = 100, d;  
#d = a * 32 + b * c;  
  
addi s1 zero 50  
addi s2 zero -500  
addi s3 zero 100  
slli t0 s1 5  
mul t1 s2 s3  
add s4 t0 t1  
nop
```

- Resta y División

```
#int a = 50, b = -500, c = 100, d;  
#d = a / b - c / 8;  
  
addi s1 zero 50  
addi s2 zero -500  
addi s3 zero 100  
div t0 s1 s2  
srli t1 s3 3  
sub s4 t0 t1  
nop
```

Convención : para dividir usas srli si el divisor es múltiplo de 2 y solo si no es variable

- Cálculo del módulo de un número (revisar)

```
#int a = 5, b = 512, c;  
#c = b % a;  
  
addi s1 zero 5  
addi s2 zero 512  
rem s3 s2 s1  
nop
```

Convención : usa la instrucción rem, no se puede usar la optimización de and porque ésta se usa cuando el segundo operando es una constante y esta constante es una potencia de dos.

- Expresión aritmética con signos de agrupación (revisar)

```
#int a = 8100, b = 6500, c = 10, d;  
#d = (((a * c) / 2) - b) * 16;  
  
lui s1 0x00002  
addi s1 s1 0xFFFFFA4  
lui s2 0x00002  
addi s2 s2 0xFFFF964  
addi s3 zero 10  
mul t0 s1 s3  
srlti t0 t0 1  
sub t0 t0 s2  
slli s4 t0 4  
nop
```

- Negación de un bit de una variable (revisar)

```
#unsigned a = 0xFFFFFFFF;  
#a = ~a;  
  
addi s1 zero 0xFFFFFFFF  
xori s1 s1 -1  
nop
```

-1 se niega todo, con 1 solo el - significativo (--)

- Verificar si una variable es múltiplo de 16 (revisar)

```
#int a = 0x3451, b;  
#b = a % 16 == 0;  
  
lui s1 0x00003  
addi s1 s1 0x450  
andi t0 s1 0xF  
slti s2 t0 1  
nop
```

Si haces and x (xor) 1 = x x (xor) 0 = 0 ,

porque 16 tiene 4 ceros al final, el valor de todo múltiplo de 16 tendrá en sus últimos 4 bits, 4 ceros, entonces el valor que quieras preservar es el de los últimos 4 ceros entonces con últimos 4 bits, xor 1 quedarán solo los

andi de lo que sea = residuo, si residuo es menor que 1 guarda 1
si regresa 0 no se cumple la condición

Se hacía con bg

- **Verificar si un número es menor que 2**

```
#int a = 6234;  
#int b = a % 4 < 2;  
  
lui s1 0x00002  
addi s1 s1 0xFFFFF85A  
andi t0 s1 0x003  
slti s2 t0 2  
nop
```

Descripción del uso de cada instrucción de cada operación anterior

lui : carga inmediato superior (load upper immediate), carga parte de un número fuera del rango de [-2048,2047]

addi : puedes sumar constantes dentro del rango [-2048,2047] y con lui fuera del rango también. No es exclusiva la suma, puedes restar.

slli : se usa en las multiplicaciones, para las fórmulas de la multiplicación y para multiplicar constantes que son exactamente potencias de 2. es desplazamiento izquierdo

srlt : se usa en las divisiones (¿qué más?) es desplazamiento a la derecha

mul : se usa cuando los valores no tienen nada que ver con factores o múltiplos de 2

div : se usa cuando los valores no tienen nada que ver con factores o múltiplos de 2

rem : se usa cuando los valores tienen factores o múltiplos de dos para sacar el módulo y sólo se usa cuando el módulo es entre variables.

andi : se usa para módulos con constantes que son múltiplos de dos. (ejemplo : verificar si una variable es múltiplo de 16)

xori : se usa para negar variables con la forma xori t0 t0 1

slti : se usa para confirmar con un valor como límite (set less than immediate) si x número es divisible por una constante. siendo 1 resultado como falso, y 0 verdadero
(Verificar si un número es menor que 2)

Operadores Aritméticos y Lógicos : A9, E9, (A10, E10)

Multiplicación por potencias de dos : F10, A10, E10

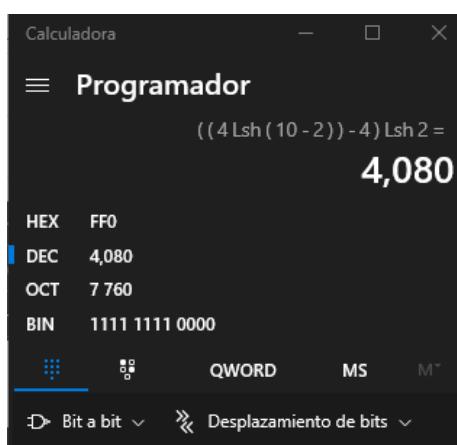
Las fórmulas de abajo se basan en $2^i * x = x \ll i$.

Éstas fórmulas evitan que el hardware de la multiplicación se use cuando se quiere calcular el producto de una variable por una constante cuyo valor es próximo a una potencia de dos.

Operación	Negativo	Positivo
$\pm(2^i - 5) \cdot x$		$((x \ll (i-2)) - x) \ll 2 - x$
$\pm(2^i - 4) \cdot x$	$((x - (x \ll (i-2))) \ll 2)$	$((x \ll (i-2)) - x) \ll 2$
$\pm(2^i - 3) \cdot x$	$((x - (x \ll (i-1))) \ll 1) + x$	$((x \ll (i-2)) - x) \ll 2 + x$
$\pm(2^i - 2) \cdot x$	$(x - (x \ll (i-1))) \ll 1$	$((x \ll (i-1)) - x) \ll 1$
$\pm(2^i - 1) \cdot x$	$x - (x \ll i)$	$(x \ll i) - x$
$\pm(2^i) \cdot x$	$-(x \ll i)$	$x \ll i$
$\pm(2^i + 1) \cdot x$	$-((x \ll i) + x)$	$(x \ll i) + x$
$\pm(2^i + 2) \cdot x$	$-((x \ll (i-1)) + x) \ll 1$	$((x \ll (i-1)) + x) \ll 1$
$\pm(2^i + 3) \cdot x$		$((x \ll (i-2)) + x) \ll 2 - x$
$\pm(2^i + 4) \cdot x$		$((x \ll (i-2)) + x) \ll 2$
$\pm(2^i + 5) \cdot x$		$((x \ll (i-2)) + x) \ll 2 + x$

Forma de usar la tabla : con ejemplo

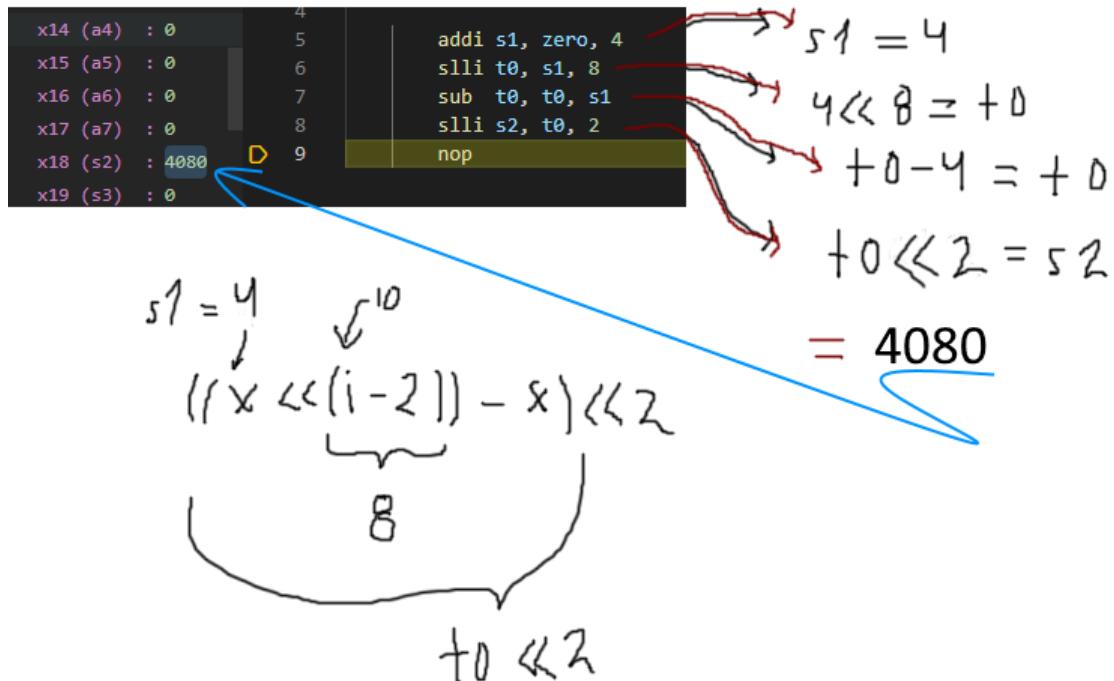
```
int a = 4, b = 1020*a;
```



2(proceso). $\rightarrow x = 4$
fórmula vacía $\rightarrow i = 10$
 $\pm(2^i - 4) * x \rightarrow +(2^{10} - 4) * 4 = 1020 * 4 = 4080$
 $((x \ll (i-2)) - x) \ll 2 \rightarrow ((4 \ll (10-2)) - 4) \ll 2$

- 1.- hacer el cálculo del problema : $1020 * 4 = 4080$
- 2.- para saber cuál fórmula usar, checas por cuál 2^i restando o sumando, y multiplicando con x, obtienes el resultado del problema. Como en 2(proceso).

En el código de la solución a éste problema, solo inicializas 4, porque según la misma fórmula 1020 o cualquier constante no es ocupada, la asignación de matemáticas con el ejercicio es la siguiente.



Ejercicio 2 :

```
int a = -4, b = -510*a;
```

$$\begin{aligned}
1.- \quad -510 * -4 &= 2040 \\
2.- \quad \pm(2^i - 2) * x \rightarrow -(2^9 - 2) * -4 &= -510 * -4 \\
(x - (x \ll (i-1))) \ll 1 \rightarrow ((-4) - ((-4) \ll (9-1))) \ll 1 &
\end{aligned}
\rightarrow$$

x14 (a4) : 0	3	addi s1, zero, -4
x15 (a5) : 0	4	slli t0, s1, 8
x16 (a6) : 0	5	sub t0, t0, s1
x17 (a7) : 0	6	slli s2, t0, 1
x18 (s2) : 2040	7	nop
	8	
	9	

≡ Programador					
$((0-4)-((0-4)\text{Lsh}(9-1)))\text{Lsh}1 =$					
2,040					
HEX	7F8	DEC	2,040	OCT	3 770
BIN	0111 1111 1000				
Bit a bit	Bit a bit	Desplazamiento de bits	WORD	MS	M+

Ejercicio 3 :

```
int a = 134, b = 69*a;
```

$$\begin{aligned}
1.- \quad 69 * 134 &= 9246 \\
2.- \quad +(2^i + 5) * x \rightarrow +(2^6 + 5) * 134 &= 69 * 134 \\
((x \ll (i-2)) + x) \ll 2 + x \rightarrow (((134 \ll (6-2)) + 134) \ll 2) + 134 &
\end{aligned}
\rightarrow$$

x13 (a3) : 0	5	addi s1, zero, 134
x14 (a4) : 0	6	slli t0, s1, 4
x15 (a5) : 0	7	add t0, t0, s1
x16 (a6) : 0	8	slli t0, t0, 2
x17 (a7) : 0	9	add s2, t0, s1
x18 (s2) : 9246	D	nop

≡ Programador					
$((((134\text{Lsh}(6-2)) + 134)\text{Lsh}2) + 134 =$					
9,246					
HEX	241E	DEC	9,246	OCT	22 036
BIN	0010 0100 0001 1110				
Bit a bit	Bit a bit	Desplazamiento de bits	WORD	MS	M+

Ejercicio 4 :

```
int a = 0xFFFFF987, b = 4091*a;
```

1.- $0xFFFFF987 = -1657 * 4091 = -6,778,787$

x09 (s1) : -1657	4	addi s1, zero, 0xFFFFF987
x10 (a0) : 0	D 5	slli t0, s1, 4

2.- $+(2^i - 5) * x \rightarrow +(2^{12} - 5) * (-1657) = -1657 * 4091$

$$((x << (i - 2)) - x) << 2) - x = (((-1657) << (12 - 2)) + 1657) << 2) + 1657$$

x14 (a4) : 0	4	addi s1, zero, 0xFFFFF987
x15 (a5) : 0	5	slli t0, s1, 10
x16 (a6) : 0	6	sub t0, t0, s1 # -(-1657)
x17 (a7) : 0	7	slli t0, t0, 2
x18 (s2) : -6778787	D 9	sub s2, t0, s1 # -(-1657)
		nop

Programador

$$((((0 - 1657) \text{Lsh}(12 - 2)) + 1657) \text{Lsh} 2) + 1657 = \\ -6,778,787$$

HEX FFFF FFFF FF98 905D

DEC -6,778,787

OCT 1 777 777 777 777 746 110 135

BIN 1111 1111 1111 1111 1111 1111 1111 1111

1111 1001 1000 1001 0000 0101 1101

Bit a bit Desplazamiento de bits

Bloques condicionales : Compilación de if E11 A11

Los siguientes problemas se resuelven empleando instrucciones y pseudoinstrucciones de bifurcación.

	C++	RISC-V
branch if equal	==	beq rd rt label
branch if not equal	!=	bne rd rt label
branch if less than	<	blt rd rt label
branch if greater or equal	>=	bge rd rt label
branch if greater	>	bgt rd rt label
branch if less or equal	<=	ble rd rt label
unsigned blt	<	bltu rd rt label
unsigned bge	>=	bgeu rd rt label
jump (incondicional)		j

Convención de if:

Para hacer la traducción de un bloque condicional escrito en alto nivel a ensamblador, debe usarse la **instrucción complemento** del operador relacional usado en alto nivel para determinar la condición de salto.

Esta regla es válida siempre que la condición del bloque use un único *operador relacional* y no operadores booleanos.

Problema 1:

```
int a = 0;  
if (a == 0)  
    a++;
```

```
        addi t0 zero 0  
        bne t0 zero L1  <-- si (t0 != 0), salta a L1, si no, baja.  
        addi t0 t0 1  
L1:    nop
```

Problema 2:

```
int a = 0;  
if (a <= 0)  
    a++;  
else  
    a--;
```

```
        addi t0 zero 0  
        bgt t0 zero L1  
        addi t0 t0 1  
        j    L2  
L1:    addi t0 t0 -1  
L2:    nop
```

Problema 3:

```
int a = 0;  
if (a != 0)  
    a--;  
else if (a > 0)  
    a = 0;  
else  
    a++;
```

```
        addi t0 zero 0  
        beq t0 zero L1  
        addi t0 t0 -1  
        j    L2  
L1:    ble t0 zero L3  
        addi t0 zero 0  
        j    L2  
L3:    addi t0 t0 1  
L2:    nop
```

Problema 4:

```
int a = 3000, b = ((4098*a) % 10 < 5) ? 8190*a : -8190*a;
```

1.- 3000 --> hex : 0xBB8
lui 0x1
addi 0xFFFFFB8

x04 (tp) : 0	1 lui t0 0x1
x05 (t0) : 3000	2 addi t0 t0 0xFFFFFB8
x06 (t1) : 0	3 nop

2.- $4098 \times 3000 = 12,294,000 \rightarrow +(2^{12}+2) \times 3000 = 4098 \times 3000$
 $((x \ll (i-1)) + x) \ll 1 \rightarrow ((3000 \ll (12-1)) + 3000) \ll 1$

x15 (a5) : 0	1 lui s1 0x1
x16 (a6) : 0	2 addi s1 s1 0xFFFFFB8
x17 (a7) : 0	3 slli t0 s1 11
x18 (s2) : 12294000	4 add t0 t0 s1
x19 (s3) : 0	5 slli s2 t0 1
x20 (s4) : 0	6 nop

≡ Programador

 $((3000 \text{ Lsh } (12 - 1)) + 3000) \text{ Lsh } 1 =$
12,294,000

HEX BB 9770	DEC 12,294,000	OCT 56 713 560	BIN 1011 1011 1001 0111 0111 0000	
Bit a bit	Desplazamiento de bits	QWORD	MS	M+

3.- $(12,294,000 \% 10 = 0) < 5?$ true $\rightarrow 8190 * 3000 = 24,570,000$

rem s2 s2 10
addi t1 zero 5
bge s2 t1 L1

≡ Programador

 $(3000 - (3000 \text{ Lsh } (13 - 1))) \text{ Lsh } 1 =$
-24,570,000

Pero no usas lui y addi para cargar la constante, porque usas las fórmulas para multiplicar por un número cercano a múltiplo de 2

$+(2^{13} - 2) * 3000 = 8190 * 3000 \quad y \quad -(2^{13} - 2) * 3000 = -8190 * 3000$
 $((x \ll (i-1)) - x) \ll 1 \quad \quad \quad (x - (x \ll (i-1))) \ll 1$
 $((3000 \ll (13-1)) - 3000) \ll 1 \quad \quad \quad (3000 - (3000 \ll (13-1))) \ll 1$

≡ Programador

 $((3000 \text{ Lsh } (13 - 1)) - 3000) \text{ Lsh } 1 =$
24,570,000

```

    lui      s1 0x1
    addi    s1 s1 0xFFFFFB8
    slli    t0 s1 11
    add     t0 t0 s1
    slli    s2 t0 1
    addi   t0 zero 10
    rem    s2 s2 t0
    addi   t0 zero 5
    bge   s2 t0 L1
    slli   s2 s1 12
    sub    s2 s2 s1
    slli   s2 s2 1
    j      L2
L1:   slli   s2 s1 12
      sub    s2 s1 s2
      slli   s2 s2 1
L2:   nop

```

Así se cargarían con lui y addi si no usaramos la convención de multiplicación por 2 :

8190 → hex : 0x1FFE
lui 0x2
addi 0xFFFFFFFF

x03 (gp) : 268435456	1
x04 (tp) : 0	2 lui t0 0x2
x05 (t0) : 8190	3 addi t0 t0 0xFFFFFFFF
	4 nop

-8190 → hex : 0xFFFFE002
lui 0xFFFFE
addi 0x00000002

x04 (tp) : 0	1 lui t0 0xFFFFE
x05 (t0) : -8190	2 addi t0 t0 0x00000002
	3 nop

Problema 5:

```
int sel = 0;      //este programa funciona como un if, else if, else
int a = (sel == 0) ? 0xFFFFFFFF
                 : (sel == 1) ? 0x00010ADF
                 : (sel == 2) ? 0xFFFFFAAA
                 : (sel == 3) ? 0xF0000AAA
                 : (sel == 4) ? 0x0000F000
                 : 0xF0000111;
```

Con BitCalculator, en el recuadro de hexadecimal escribes cada valor, los que tienen una F en la posición más significativa son negativos entonces usas el botón de +/- para negarlos y obtener valores (sé que se tienen que negar pero debo revisar mi explicación con C2)

```
(0) 0xFFFFFFFF = -1 (no necesita lui)

(1) 0x00010ADF = 68,319
lui s2 0x11
addi s2 s2 0xFFFFFADF

(2) 0xFFFFFAAA = -1366 (no necesita lui)

(3) 0xF0000AAA = -268,432,726
lui s2 0xF0001
addi s2 s2 0xFFFFFAAA

(4) 0x0000F000 = 61440
lui s2 0xF          no necesitas 0x0000F o addi

(else) 0xF0000111 = -268,435,183
lui s2 0xF
addi s2 s2 0x111
```

```

        addi    s1 zero 0
        bne    s1 zero L1
        addi    s2 zero -1
        j      L2
L1:   bne    s1 1 L3
        lui    s2 0x11
        addi    s2 s2 0xFFFFFADF
        j      L2
L3:   bne    s1 2 L4
        addi    s2 zero -1366
        j      L2
L4:   bne    s1 3 L5
        lui    s2 0xF0001
        addi    s2 s2 0xFFFFFAAA
        j      L2
L5:   bne    s1 4 L6
        lui    s2 0xF
        j      L2
L6:   lui    s2 0xF0000
        addi    s2 s2 0x111
L2:   nop

```

A12, S13, A13 : Bifurcación y Condiciones Lógicas

Cuando la condición de salto en alto nivel tiene la forma (a && b), **basta con que a sea falso para que el salto se produzca**. Si a es verdadero, entonces se debe evaluar la veracidad de b para producir o no el salto.

Si la condición de salto tiene la forma (a || b), entonces **basta con la a sea verdadera para que el bloque de instrucciones de la estructura if se ejecute**, si no, se debe evaluar b para decidir si el salto condicional debe o no ocurrir.

Operadores Lógicos Precedencia

- 1.- not
- 2.- and
- 3.- or

Convenciones de and y or

Si es Estructura AND

(bin : branch instruction) (bin es la negativa de condición de alto nivel)

bin L1 salta si no se cumple a siguiente condición

bin L1 salta si no se cumple a siguiente condición

asumes que se cumplen todas las condiciones

Si es **Estructura OR**:

(bin : branch instruction) (bin es la negativa de condición de alto nivel)

bin1 : instrucción de igualdad/diferencia

bin2 : instrucción bin1 negada

or salta al resultado verdadero, desde el primero

asumes que te da 0,

salto en la último si es verdadero

Para compilar condiciones lógicas, compilas de la siguiente manera :

a<5 && b>1 || c ≤ 12 && d>9

amarillos – apuntan a siguiente condición/ fuera de la expresión

rojos – apuntan a la siguiente condición/ cuerpo

1. if a<5 = true

→ if b>1 = true → $((1 \&\& 1 = 1) \parallel x) \rightarrow (1 \parallel x = 1)$ → resultado = 1

1. if a<5 = falso → checas $c \leq 12$

if $c \leq 12 = \text{true} \rightarrow \text{checas } d > 9$

if $c \leq 12 = \text{falso} \rightarrow 0 = \text{resultado}$

if $d > 9 = \text{true} \rightarrow (0 \parallel 1) \rightarrow 1 = \text{resultado}$

if $d > 9 = \text{falso} \rightarrow 0 = \text{resultado}$

a	b	c	d	a&&b	c&&d	a&b c&d
1	1	x	x	1	x	1
1	0	1	1	0	1	1
0	x	1	1	0	1	1
0	x	1	0	0	0	0
0	x	0	x	0	0	0

primero checas que a sea verdadero, porque es and, debes checar b si a es verdadero.
porque ambos tienen que ser verdaderos, entonces solo saltas si ($b < 5$) pero la
mnemotecnia es que haces el salto si es cero porque siempre debes comparar con cero

asumes que es 0 entonces saltas si es 1

1. Compilación de un bloque condicional que usa el operador *not*. Modifica el valor de la variable *a* para que observes el flujo de ejecución del programa.

```
bool a = true, b = false;
if (!a)
    b = true;

    addi s1, zero, 0  #a = true
    xor s2, s2, s2  #b = false
    xori t0, s1, 1  #!a
    beq t0, zero, L1      #if !a = 0 → nop
    addi s2, zero, 1      #if !a = 1 → b = 1
L1:   nop

# s1 <-> a, s2 <-> b
```

2. Compilación de un bloque condicional que usa el operador *and*. Modifica el valor de las variables *a* y *b* para que observes el flujo de ejecución del programa.

```
bool a = true, b = true, c;
if (a && b)
    c = !(a && b)
else
    c = a && b;

# s1 <-> a, s2 <-> b, s3 <-> c
    addi s1, zero, 1  # a = true
    addi s2, zero, 1  # b = true
    beq s1, zero, L1  # a = false?
    beq s2, zero, L1  # b = false?
    xori t0, s1, 1    # !(a) -> t0
    beq t0, zero, L3  # !(a) = 0?      L3
    j    L4
L3:   xori t0, s2, 1    # !(b) -> t0
    beq t0, zero, L5  # !(b) = 0?      L5
L4:   addi s3, zero, 1  # s3 = 1
    j    L2
L5:   xor s3, s3, s3
    j    L2
L1:   beq s1, zero, L6  # else
    beq s2, zero, L6
    addi s3, s3, 1
    j    L2
L6:   xor s3, s3, s3      # negación
L2:   nop
```

3. Compilación de un bloque condicional que usa el operador *or*. Modifica el valor de las variables *a* y *b* para que observes el flujo de ejecución del programa.

```
bool a = true, b = false, c;
if (a || b)
    c = !a || b;
else
    c = a || !b;
```

```

# s1 <-> a, s2 <-> b, s3 <-> c
    addi s1, zero, 1
    addi s2, zero, 1
    bne s1, zero, L1 #a true? → c = !a || b
    beq s2, zero, L2 #b false? → c = a || !b
L1:   xor t0, s1, 1
    bne t0, zero, L4
    beq s2, zero, L5
L4:   addi s3, zero, 1
    j    L3
L5:   xor s3, s3, s3
    j    L3
L2:   bne s1, zero, L6 #a true? → c = 1  c = true
    xor t0, s2, 1      #b false? → c = a || !b
    beq t0, zero, L7    #b true? → c = !(c)
L6:   addi s3, zero, 1
    j    L3
L7:   xor s3, s3, s3
L3:   nop

```

4. Compilación de un bloque condicional que verifica que la variable x se encuentra dentro de los rangos $[-10, 1]$ o $[1, 10]$, matemáticamente, $x \in [-10, 1] \cup [1, 10]$, donde $x \in \mathbb{Z}$. Escribe el código C++ correspondiente y tradúcelo a RISC-V. Modifica el valor de la variable x para observes el flujo de ejecución del programa

```

int main()
{
    int x = 0;
    bool F = false;
    if (x >= -10 && x <= -1 || x >= 1 && x <= 10)
        F = true;
    return 0;
}

```

**xor : negamos si es registro consigo mismo porque $1 \ 1 = 0 \ 00 = 0$
xori : hacer xor, negar bits en la posición que quieras**

**si tienes (xori) de y con 0 =
3 = 011 →
1010 → posiciones 1 y 3 se niegan
and
andi**

0x000A8071 Andi

- Checar si es múltiplo de 16
 $16 \rightarrow \underbrace{10000}_{\text{Tiene 4 ceros}}$
- Hacemos andi con $0xF\{4$ unos
- $\delta 0x000A8071$
 $\delta 0x0000000F$
- $\delta \begin{array}{r} 01110001 \\ 00001111 \end{array}$
- $\overline{0...00000001}$ $\times \delta 1 = X$
 Res = 1
- $\times \delta 0 = 0$

- Cargar \emptyset en la variable

$$\left. \begin{array}{l} x \oplus x = 0 \\ x \oplus 1 = x' \\ x \oplus 0 = x \end{array} \right\} \quad \begin{array}{r} : 0x000A8071 \\ \oplus 0x000A8071 \\ \hline \emptyset \end{array}$$

- Negar la variable

$$0x000A8071 \quad \begin{array}{r} \oplus 0001 \\ 1111 \\ \hline 1110 \end{array}$$

$$0x000A8071 \quad 0x000A8071 \quad 0x000A8071$$

$$0xFFFFFFFF = -1$$

$$\overline{0xFFFFF78E} \rightarrow \text{Negación}$$

* En binario: $11111111111111111111111111110000$

5. Compilación de la ecuación booleana $F(x, y) = x \oplus y$. Escribe el código C++; después tradúcelo a RISC-V. Comprueba que tu código ensamblador funciona para cualquier combinación de las variables x, y y z.

```
int main()
{
    bool x = false, y = false;
    bool F = (!x && y || x && !y); //definición de xor talvez no se puede hacer directamente xor
    return 0;
```

```
}
```

Programa que checa el mayor de 3 números (A13.1):

```
int main()
{
    int a=3, b=4, c=5, max;
    if(a>b && a>c)
        max = a;
    else if(b>a && b>c)
        max = b;
    else if(c>a && c>b)
        max = c;
    else
        max = 0;
    return 0;
}
```

```
addi s1 zero 3
addi s2 zero 4
addi s3 zero 5
ble s1 s2 L1
ble s1 s3 L1
addi s4 = s1
j L2
L1: ble s2 s1 L3
ble s2 s3 L3
addi s4 = s2
j L2
L3: ble s3 s1 L4
ble s3 s2 L4
addi s4 = s3
j L2
L4: addi s4 zero 0
L2: nop
```

Programa que checa tipos de triángulos (A13.2)

```
#s1 <--> a
#s2 <--> b
#s3 <--> c
#s4 <--> type
#t0 <--> is_triangle
#t1 <--> temporal auxiliar

addi s1 zero 1 # unsigned a = 1
```

```

addi s2 zero 2
addi s3 zero 3
addi t0 zero 1 #bool is_triangle = true
bleu s1 s2 L1 # 1 if (a > b && a > c)
bleu s1 s3 L1
add t1 s2 s3
bltu s1 t1 L1 #if (a >= (b + c))
addi t0 zero 0 #bool is_triangle = false
j L2
L1: bleu s2 s1 L3 # 2 else if (b > a && b > c)
bleu s2 s3 L3
add t1 s1 s3
bltu s2 t1 L3 #if (b >= (a + c))
addi t0 zero 0 #bool is_triangle = false
j L2
L3: bleu s3 s1 L2 # 3 else if (c > a && c > b)
bleu s3 s2 L2
add t1 s1 s2
bltu s3 t1 L2 #if (c >= (a + b))
addi t0 zero 0 #bool is_triangle = false
j L2 #tal vez no es necesario
L2: addi t1 zero 1 # 4 if (is_triangle)
bne t0 t1 L4 # is_triangle == true --> t0 ya no tiene que ser is_triangle
bne s1 s2 L5 # if (a == b && a == c)
bne s1 s3 L5
addi s4 zero 3 # type 3
j L6
L5: beq s1 s2 L7 # else if ([a != b] && a != c && b != c)
beq s1 s3 L7 # else if (a != b && [a != c] && b != c)
beq s2 s3 L7 # else if (a != b && a != c && [b != c])
addi s4 zero 1 # type 1
j L6
L7: addi s4 zero 2 # type 2
j L6
L4: addi s4 zero 0 # type 0
L6: nop

```

S14: Traducción de estructuras iterativas

Traducción de un bucle do-while.

Línea	C++	RISC-V
1	do	
2	{	
3	accum++;	do1: addi s1, s1, 1
4	} while (accum < max);	blt s1, s2, do1

Traducción de un bucle while.

Línea	C++	RISC-V
1	while (accum < max)	j wh1
2	{	
3	accum++;	L1: addi s1, s1, 1
4	}	wh1: blt s1, s2, L1

Traducción de un bucle for.

Línea	for	RISC-V
1	for (int accum = 0; accum < max; accum++)	addi s1, s1, 0
2		j for1
3		L1: addi s1, s1, 1
4		for1: blt s1, s2, L1

For y While son iguales

RISC-V (for)	RISC-V (while)
addi s1, s1, 0	addi s1, s1, 0
j for1	j wh1
L1: addi s1, s1, 1	L1: addi s1, s1, 1
for1: blt s1, s2, L1	wh1: blt s1, s2, L1

S15 : Bucles Anidados

Estructura regular de for/while:

- 1.- (inicialización)
 - 2.- (cuerpo)
 - 3.- (revisar condición)
 - 4.- (actualización)
 - 5.- (cuerpo)
 - 3.- (revisar condición)
 - 4.- (actualización)
- sucesivamente

Orden de Compilación de for(/while) anidado :

- 1.- bucle externo (inicialización) → cuerpo de bucle externo (bucle interno)
- 2.- bucle interno (inicialización)
- 3.- bucle interno (cuerpo)
- 4.- bucle interno (actualización)
- 5.- bucle interno (revisar condición) → sale y regresa a bucle externo
- 6.- bucle externo (actualización)
- 7.- bucle externo (revisar condición) → sale del For anidado, o regresa al bucle interno y se repite

Estructura regular de do while:

- 1.- (cuerpo)
 - 2.- (revisar condición)
 - 3.- (cuerpo / actualización)
 - 4.- (revisar condición)
 - 5.- (cuerpo / actualización)
- sucesivamente

Ciclo while/for externo

C++	RISC-V
int i = 0, accum = 0;	addi s1, zero, 0
	addi s3, zero, 0

C++	RISC-V
while(i < ROW_MAX)	j wh1
\\" bloque de instrucciones	L1: # bloque de instrucciones

Ciclo while/for interno

inicialización de j = 0 addi s3, zero, 0
 salto incondicional j wh2
bloque de instrucciones
 condición del bucle wh2: addi t0, zero, 10
 blt s1, t0, L2

Ciclo completo:

C++	RISC-V
while (j < COL_MAX)	j wh2
{	L2: add t0, s1, s3
accum += i + j;	add s2, s2, t0
j++;	addi s3, s3, 1
}	wh2: addi t0, zero, 10
	blt s1, t0, L2

```

        addi s1, zero, 0
        addi s2, zero, 0
        j    wh1                  # outer while loop
L2:   addi s3, zero, 0
        j    wh2                  # inner while loop
L1:   add  t0, s1, s3
        add  s2, s2, t0
        addi s3, s3, 1
wh2:  addi t0, zero, 10      # loop condition (inner)
        blt  s3, t0, L1
        addi s1, s1, 1
wh1:  addi t0, zero, 10      # loop condition (outer)
        blt  s1, t0, L2
        nop
    
```

Convenciones :

- los acumuladores se guardan en registros s
- operaciones que incrementen el valor de un registro s, deberán hacerse en un temporal y sumarse al registro
- El orden de compilación es el de “Orden de Compilación de for(/while) anidado”

Organización de la memoria - S16

Organización de 32 bits si es de más bits la distancia de cada parte crece porque hay más direcciones

Tabla Diagrama

368 del libro de Harris de memoria

0xFFFFFFF 0xC0000000	OS, BIOS & Input Output	Aquí está la última palabra de la memoria. I/O se mapea, las direcciones de los periféricos, que también tienen o se ven como direcciones, el código se guarda en buffer y se copia en memoria principal, Si quieras leer el teclado llegas a la dirección 0xC0000000.
0xBFFFFFFF 0x10001000	Stack v Dynamic Data Heap ^ Memoria dinámica en montículo/Heap, guarda estructuras de datos, árboles, es complicado en nivel ensamblador entonces no checamos eso. Sirve para almacenar todos los objetos creados o que resultan de usar new. Cuando usas malloc o memory allocation se guarda en heap, va hasta 0x10001000.	Se guardan datos creados en tiempo de ejecución (datos dinámicos), se controla de dos formas con stack pointer (sp), que apunta a los datos dinámicos que se van generando, y para checar los valores usas la referencia del tope de la pila, con stack pointer, al ser apuntador, almacena direcciones. datos generados por funciones, en pila, Memoria estática en el stack
0x10000FFC 0x10000000	Global Data Cualquier función puede acceder a la sección de variables globales. para acceder a variables globales se usa gp (global pointer)	Variables globales que pueden ser accedidas por cualquier programa o función en cómputo concurrente, varios programas al mismo tiempo que modifican la variable global, procesos(programas) ligeros (hilos) y programas normales.

0x10000000 0x00010000	Text La sección texto almacena el código máquina de los programas del usuario se accede con el apuntador pc, program counter, se mueve de 4 en 4, no se puede manipular directamente. Muy importante porque sumando valores al apuntador, se producen los saltos, cuando es un salto incondicional se suma una compensación para producir el salto, y también cuando no es incondicional.	porque cada instrucción ocupa 4 bytes. entonces 4 + 4 ... Al depurar se puede ver, esa instrucción se almacena en el PC. si es positiva la compensación, entonces puede ser if si es negativa se regresa y tal vez fue un for. 0x00010000 PC guarda direcciones de instrucciones como valores de 4 en 4.
0x00010000 0x00000000	Exception Handlers Para “dormirlos” tendrías que agregar excepciones . el procesador no tiene ésta sección. pero es importante que sepas de su existencia. 0x00000000	Tratan eventos externos. El programa que siempre se ejecuta. Para pararlo se espera que lea la excepción cargue, y haga lo demás. Porque hay muchos programas al mismo tiempo, entonces checa los gestores de excepción.

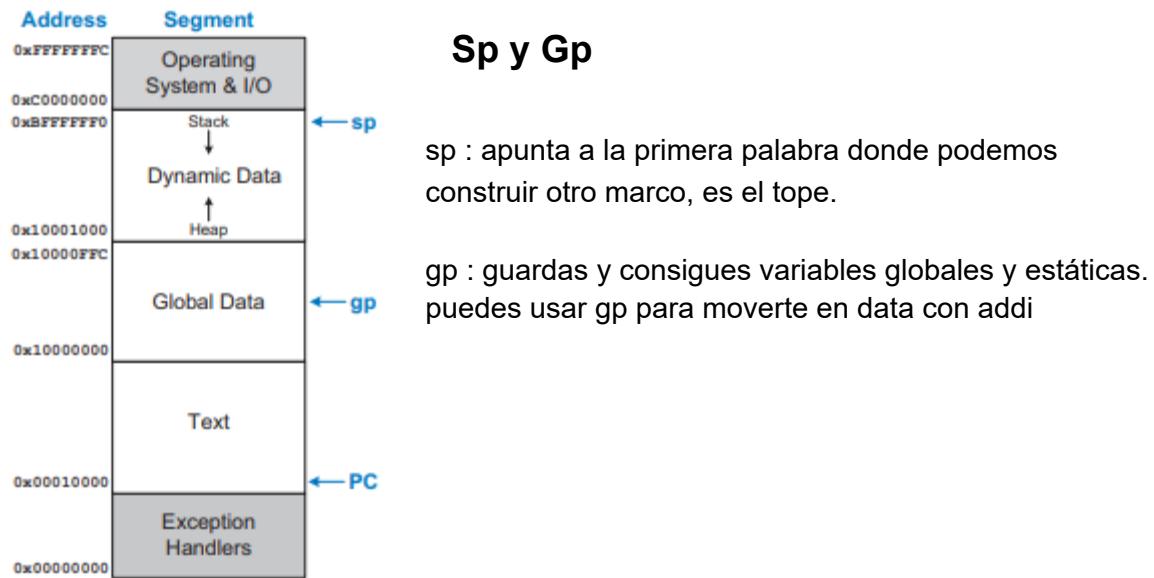


Figure 6.31 Example RISC-V memory map

Traducción de Variables Globales : S16 A16

Las variables globales se guardan en la directiva .data

Uso de lw y sw las direcciones son de 32 bits entonces usas lw y sw con caracteres, solo cuando usas apuntador a un valor requieres de load o store con el tamaño del tipo de dato.

Funciones Hoja, Anidadas, y Recursivas : Creación de marco

Ejemplos

Revisión de instrucciones

Desensamble

Convención sobre la compilación de funciones

address	value	sp	fp/s0	Reglas
dec	hex	3 2 1 0		1. Todos los bloques son opcionales con excepción del magenta. 2. El tamaño de los bloques es múltiplo de 16 bytes.
2147483632	7FFFFFFF0		<--	3. Los registros se almacenan en orden ascendente. 4. Las variables se almacenan según su orden de aparición. 5. Los arreglos se ordenan de manera descendente.
2147483628	7FFFFFFEC			6. El sp apunta al tope del marco. 8. El s0/fp apunta al fondo del marco.
2147483624	7FFFFFFE8			9. Las funciones anidadadas deben respaldar sus datos según el orden de los bloques. 10. Las funciones hoja no necesitan respaldar registros.
2147483620	7FFFFFFE4			
2147483616	7FFFFFFE0			
2147483612	7FFFFFFDC			
2147483608	7FFFFFFD8			
2147483604	7FFFFFFD4			
2147483600	7FFFFFFD0			
2147483596	7FFFFFFCC			
2147483592	7FFFFFFC8			
2147483588	7FFFFFFC4			
2147483584	7FFFFFFC0			
2147483580	7FFFFFFBC			
2147483576	7FFFFFFB8			
2147483572	7FFFFFFB4			
2147483568	7FFFFFFB0			
2147483564	7FFFFFFAC			
2147483560	7FFFFFFA8			
2147483556	7FFFFFFA4			
2147483552	7FFFFFFA0		<--	

Azul : Guardas argumentos extra (si es necesario usar más de 8, es decir ya no caben en registros a, ejemplos printf y cout : funciones variátricas.) y arreglos de tamaño variable (VLA, no los veremos pero se ven así “a[n];”).

Gris : Registros de argumentos (a0 - a7) y de temporales (t0 - t6) Todas las funciones anidadas y recursivas necesitan bloque gris.

Amarillo : Variables locales y arreglos.

Verde : Registros guardados (saved) tipo s, (s1 - s11) s0 es exclusivamente usado para fp.

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

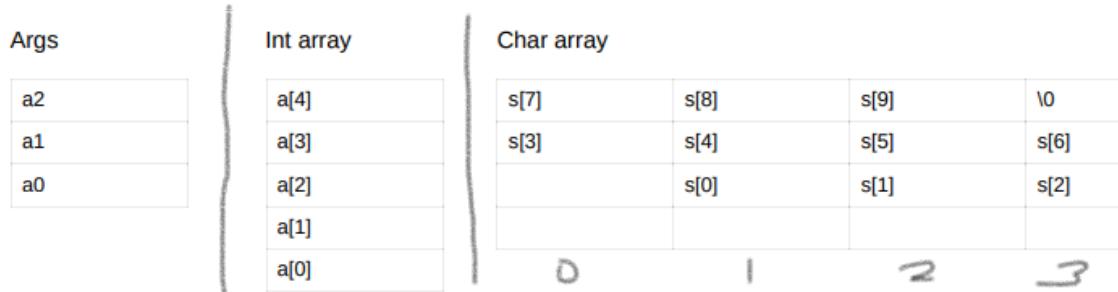
Magenta : Registros ra(dirección de retorno) y fp(s0) (Siempre necesaria)

dato : ra contiene la dirección de retorno de cada parte para el programa y de las funciones también, es diferente de dónde se guarda un valor que regresas. Cuando regresas un valor lo haces en a0. (Mide siempre 16 bytes)

1. Al entrar a una función anidada o recursiva debes de crear el marco, posteriormente cada que aparezca un return debes de cargar el valor del retorno en el registro a0, si es que lo hay, para después destruir el marco de la función y realizar el retorno.
2. Usar la menor cantidad de registros "t" y usarlos de menor a mayor.
3. Los argumentos siempre se calculan de mayor a menor, al llamar a una función

```
Funcion(arg1, arg2, arg3)
arg3 <- a2
arg2 <- a1
arg1 <- a0
```

4. Los arreglos y registros se almacenan en la memoria de abajo para arriba



3. Al almacenar un arreglo declarado dentro de una función debes usar la dirección base de esta, almacenandola en un registro temporal
4. Al almacenar variables en la memoria debes usar el frame pointer para almacenarlas dentro del marco al que corresponden
5. al cargar o guardar variables globales se usa el global pointer como referencia
6. para cargar la dirección base de un arreglo de variables globales debes de cargar la dirección del gp, más la compensación necesaria en un registro temporal
7. los valores que se almacenen en la memoria al crear un marco, se almacenan usando el sp, debido a que el frame pointer no ha sido inicializado (registros s)

Para más información, revisar libros de Harris y Patterson.

También las siguientes ligas en inglés :

https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture6.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture7.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture8.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture9.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture10.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture11.pdf
https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture12.pdf

hasta

https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/Lecture26.pdf

y

<https://robotics.shanghaitech.edu.cn/courses/ca/20s/projects/1/>
<https://github.com/michaeljclark/riscv-disassembler>
<https://github.com/kvakil/venus>
<https://venus.kvakil.me/>
<https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus>

como herramientas de apoyo para RISC V