Sistemas digitales

Codificación de instrucciones
Ciclo de Instrucción

2024 - Segundo Cuatrimestre

Universidad de Buenos Aires - FCEyN - Departamento de Computación

1

Usaremos el manual de RISC-V

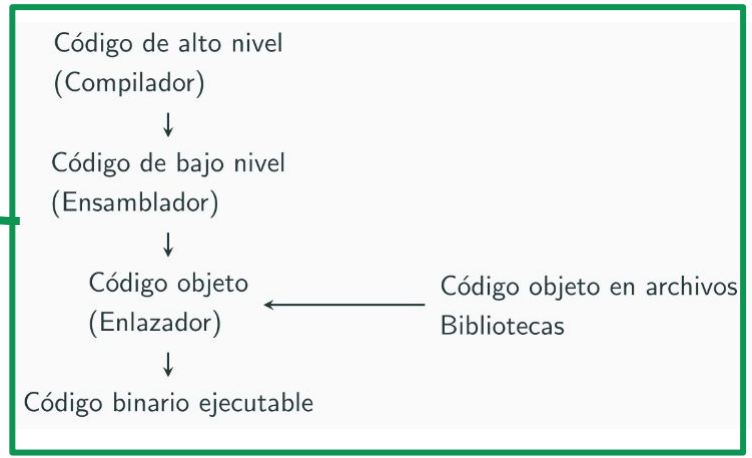
Programa escrito en Assembler

ADDI a0, a1, 0x002

SRLI a0,a0,3

CODIFICAR

0001100110000110011
0000110011001110001
10



2

Programa escrito en Assembler

CODIFICAR

0001100110000110011
0000110011001110001
10

CÓDIGO BINARIO

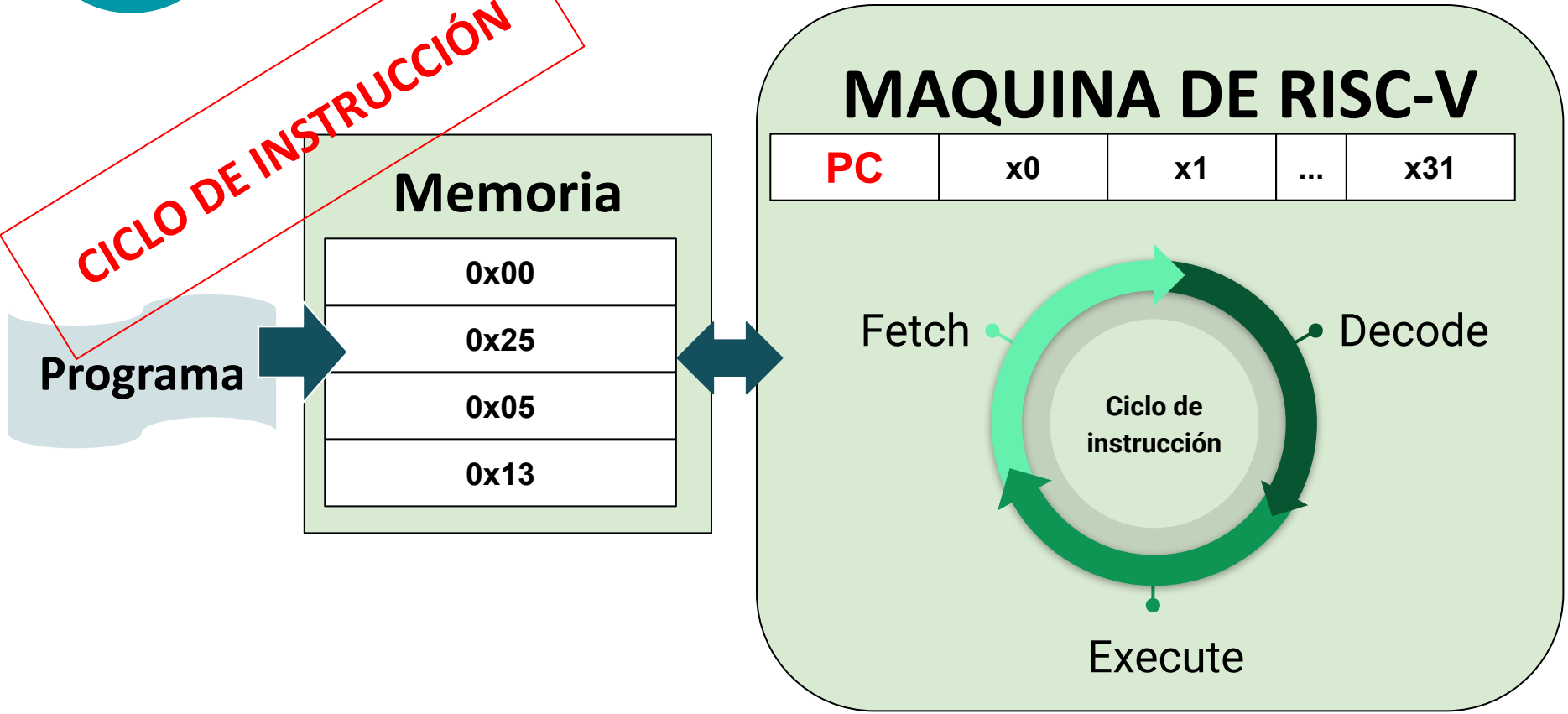
MÁQUINA
(RISC-V)

Memoria

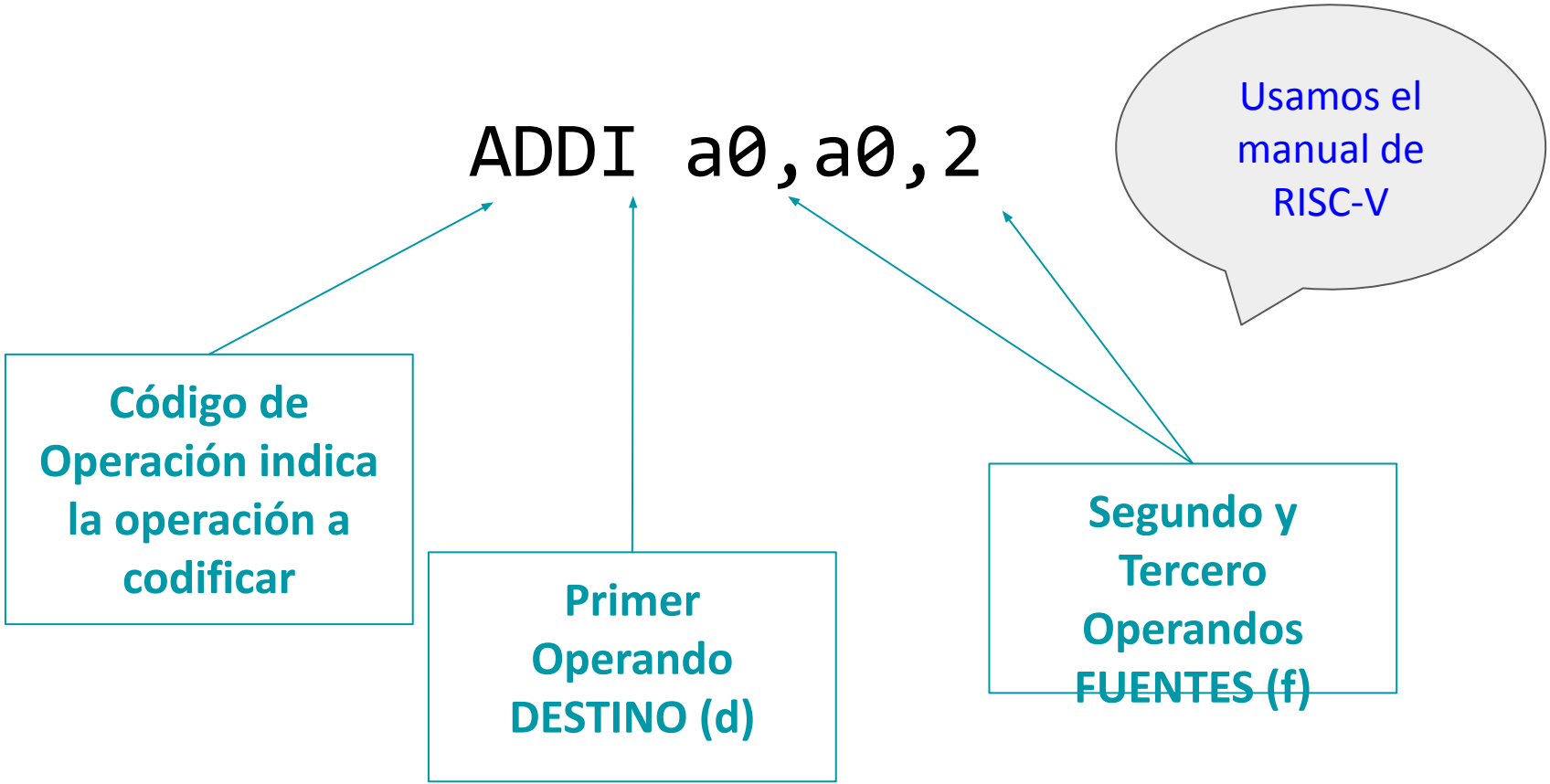
0x00
0x25
0x05
0x13

3

CICLO DE INSTRUCCIÓN



Codificación de programas en RISC-V



Codificación de instrucciones en RISC-V

Instrucción: ADDI (add immediate)

rd = registro destino
rs1 = registro fuente 1

efecto

extensiones

formato

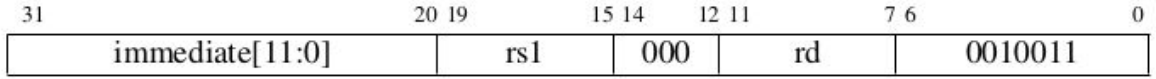
addi rd, rs1, immediate

`x[rd] = x[rs1] + sext(immediate)`

Add Immediate: Tipo I, RV32I y RV64I.

Suma el *inmediato* sign-extended al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.

Formas comprimidas: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm



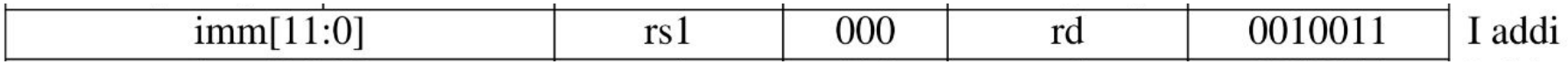
Codificación de instrucciones en RISC-V

```
ADDI a0,a0,2
```

Código de Operación: 0010011

Operando DESTINO: 01010

Operandos FUENTES: 01010 0000 0000 0010



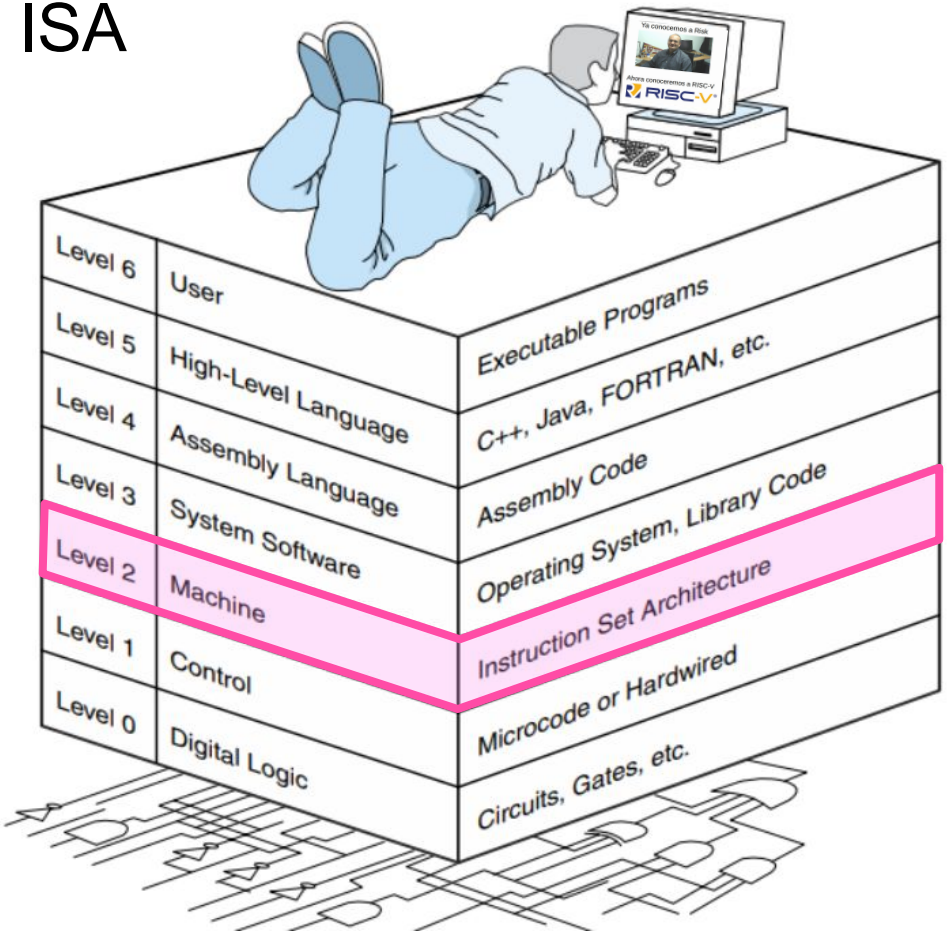
0000 0000 0010 | 01010 | 000 | 01010 | 0010011

0000 0000 0010 0101 0000 0101 0001 0011 = 0x00250513


RISC-V®

ÍNDICE DE LA CLASE

Introducción: ISA



Introducción: Instruction Set Architecture

- En simples palabras, una ISA es el conjunto de instrucciones que tiene un procesador y es usado por les programadores para escribir sus programas.

- En estas clases y en particular en el parcial, vamos a trabajar con la ISA de RISC-V.

- Pero, ¿esto no es ya lo que venimos haciendo?

Introducción: Instruction Set Architecture

Instrucción	CodOp	Formato	Acción
ADD Rx, Ry	00001	A	Rx ← Rx + Ry
ADC Rx, Ry	00010	A	Rx ← Rx + Ry + flag_C
SUB Rx, Ry	00011	A	Rx ← Rx - Ry
AND Rx, Ry	00100	A	Rx ← Rx and Ry
OR Rx, Ry	00101	A	Rx ← Rx or Ry
XOR Rx, Ry	00110	A	Rx ← Rx xor Ry
CMP Rx, Ry	00111	A	Modifica *flags* de Rx - Ry
MOV Rx, Ry	01000	A	Rx ← Ry
STR [M], Rx	10000	D	Mem[M] ← Rx
LOAD Rx, [M]	10001	D	Rx ← Mem[M]
STR [Rx], Ry	10010	A	Mem[Rx] ← Ry
LOAD Rx, [Ry]	10011	A	Rx ← Mem[Ry]

ISA de OrgaSmall

ISA de Orga1

Codop	Operación	Descripción	Condición de Salto
1111 0001	JE	Igual / Cero	Z
1111 1001	JNE	Distinto	not Z
1111 0010	JLE	Menor o igual	Z or (N xor V)
1111 1010	JG	Mayor	not (Z or (N xor V))
1111 0011	JL	Menor	N xor V
1111 1011	JGE	Mayor o igual	not (N xor V)
1111 0100	JLEU	Menor o igual sin signo	C or Z
1111 1100	JGU	Mayor sin signo	not (C or Z)
1111 0101	JCS	Carry / Menor sin signo	C
1111 0110	JNEG	Negativo	N
1111 0111	JVS	Overflow	V

Introducción: ISA incremental

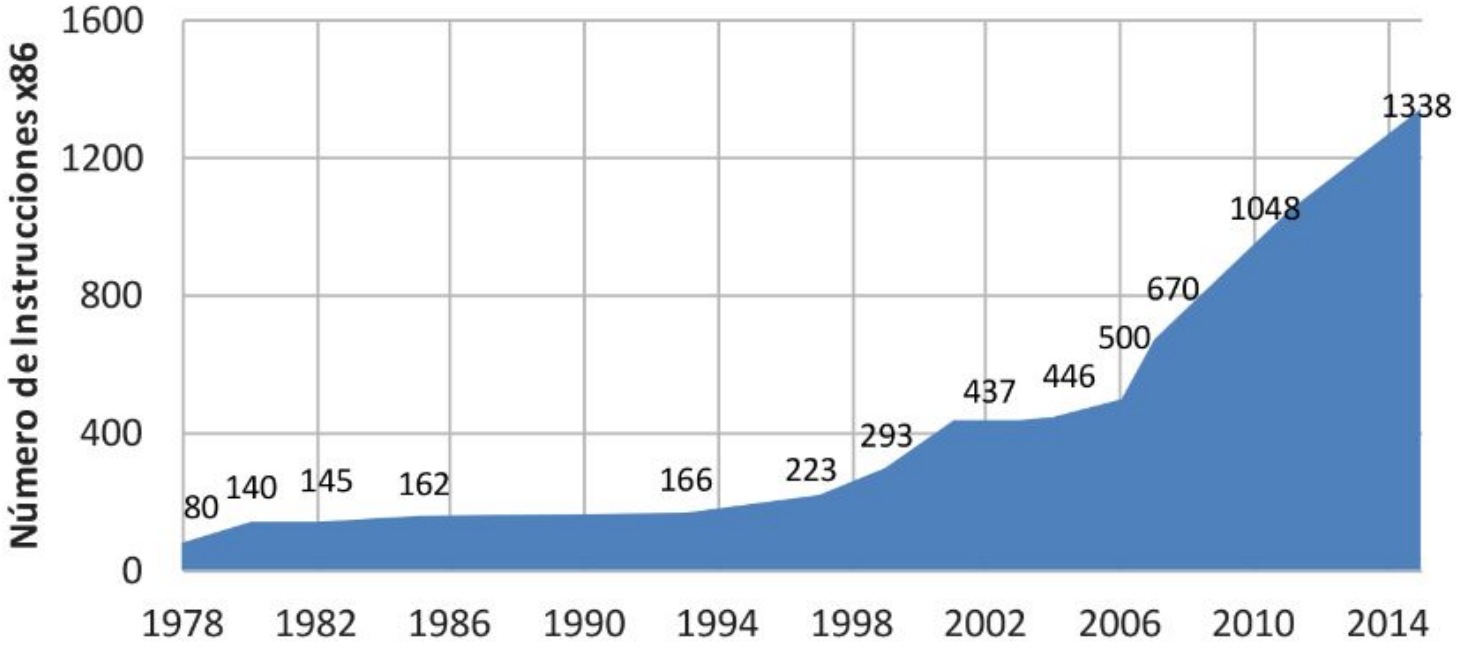
El enfoque convencional en arquitecturas es desarrollar ISAs incrementales, en los cuales los nuevos procesadores no solo implementan las nuevas extensiones, sino además todas las instrucciones de ISAs anteriores.

Su objetivo es mantener compatibilidad para que los programas ya compilados y en formato binario de décadas anteriores, aún puedan seguir funcionando.

Este tipo de arquitecturas por lo general suelen ser de tipo CISC (complex instruction set computer)

Introducción: Problemas de una ISA incremental

Por ejemplo, el desarrollo de Intel por tener una arquitectura CISC:



Introducción: Ejemplo de Intel

Introducción: ¿Quién nos puede salvar?

Acá es cuando aparece RISC-V, una arquitectura RISC (reduced instruction set computer) con una ISA reconocida por ser:

- ★ **Modular**: hardware acorde a las necesidades.
- ★ **Reciente**: RISC-V se origina en 2010, cuando la mayoría de las alternativas nacieron en los 70’s u 80’s.
- ★ **Abierta**: representando una ventaja en licencias, liberada de objetivos particulares de alguna corporación, abierta a la colaboración entre distintas entidades académicas e industriales.

Introducción: Principios básicos de RISC-V

Se desea conservar una ISA pequeña para reducir el tamaño del procesador.

Cuando la ISA es simple se reduce el tamaño del chip, reduciendo el tiempo para diseñar y validarlo. Menos documentación y más facilidad para que los clientes entiendan la ISA.

Con instrucciones simples los programas se ejecutan más rápido.

Introducción: Principios de RISC-V

Es importante tener el espacio físico (opcodes) para poder agregar instrucciones, ya que hoy en día es el único camino para mejorar el rendimiento.

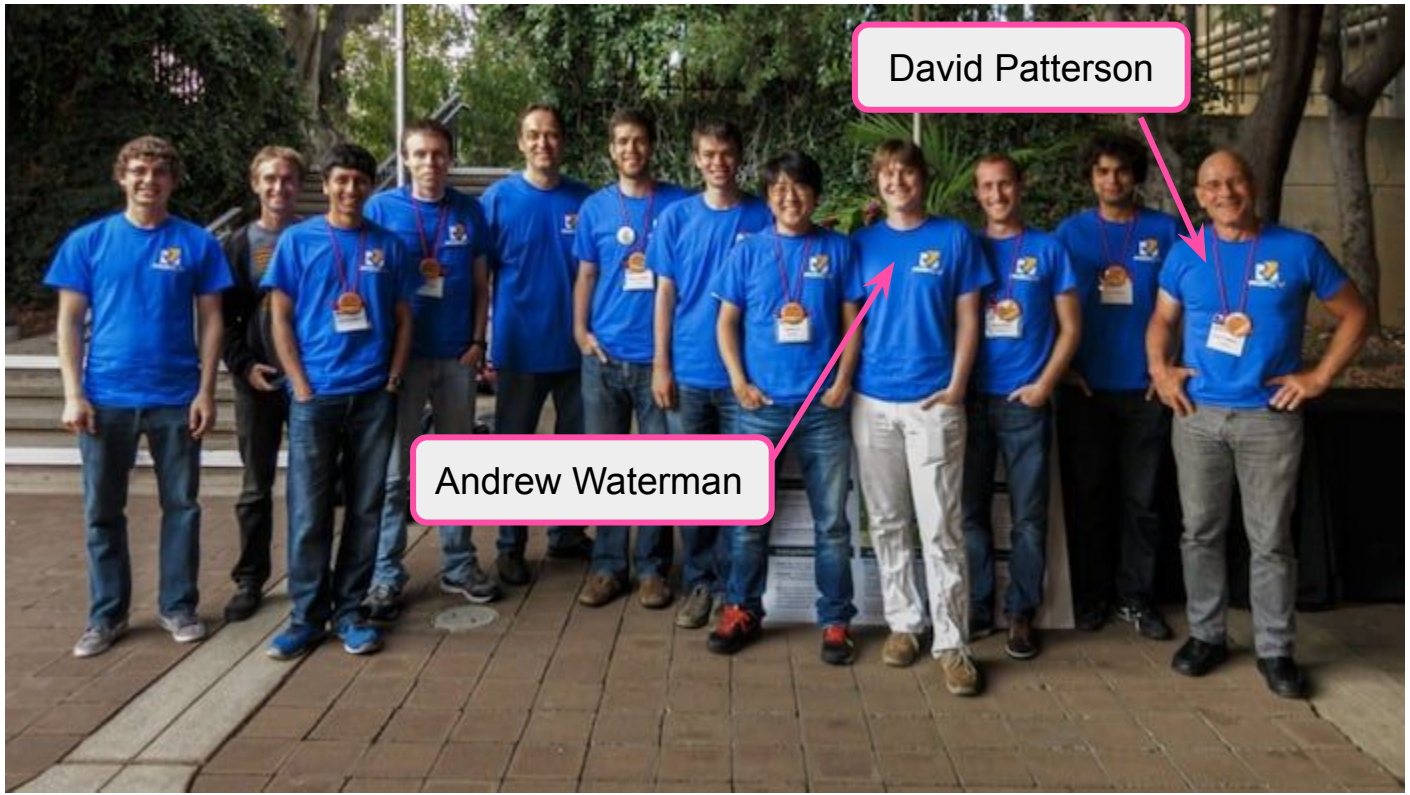
Empíricamente hablando, las ISAs con todas sus instrucciones de igual tamaño generan código más compacto que ISAs con instrucciones de tamaño variable.

Introducción: Principios de RISC-V

No solo debe ser fácil programar, sino también compilar y linkear. Además es importante tener en cuenta que los accesos a registros son más rápidos que a la memoria.

Es sabido que un programador de lenguaje máquina debe conocer la arquitectura para escribir programas correctos. No se deberían agregar funciones que ayudan a una implementación específica, así tampoco deberían agregar funciones que afecten otras implementaciones.

Introducción: algunos pioneros en este proyecto



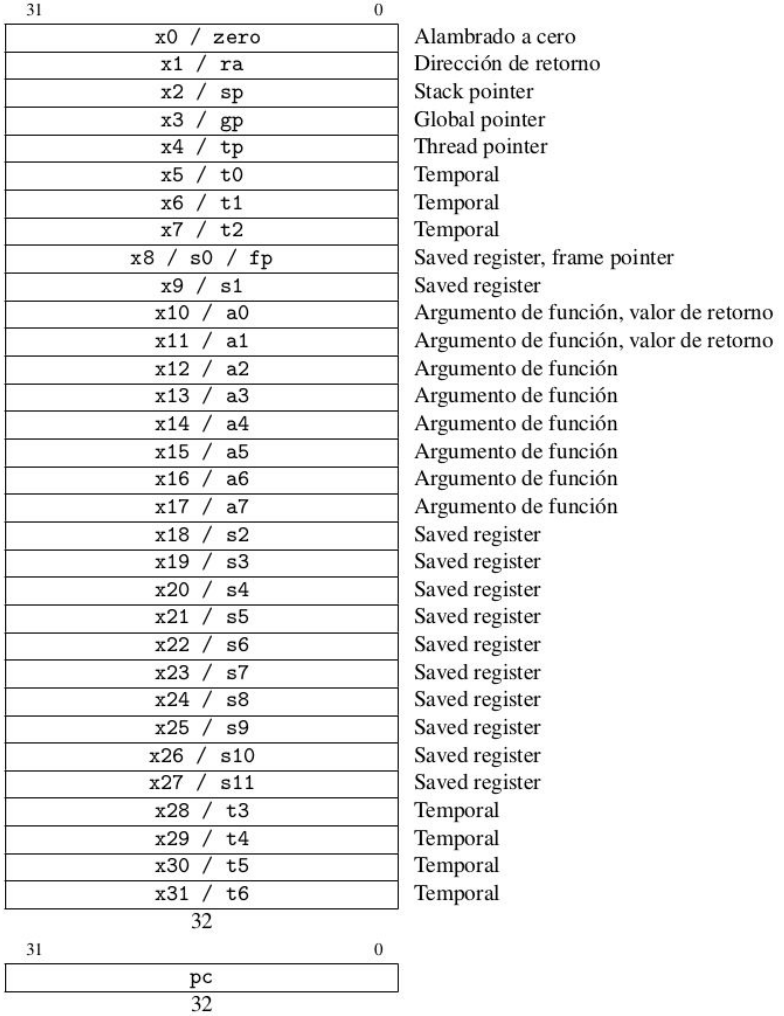
RV32I: ISA RISC-V base para números enteros

RISC-V nos proporciona un diseño modular, nosotros veremos en detalle el módulo RV32I que es el módulo base de operaciones con números enteros. Algunas características importantes son:

- ➔ Instrucciones de datos, aritméticas, lógicas, shifts, y saltos.
- ➔ Todas las instrucciones son de **32 bits**.
- ➔ Palabras de **32 bits**, direccionamiento a **byte**.
- ➔ **32 registros** de uso general más uno dedicado al PC.
- ➔ Todas las operaciones son entre registros y con inmediatos de dos tipos (con signo y sin signo). **No permite operaciones registro-memoria ni memoria-memoria** (operaciones aritméticas/lógicas).
- ➔ Los inmediatos son extendidos en signo a 32 bits.
- ➔ Instrucciones de 6 formatos diferentes:
  - ◆ Operaciones entre registros: ***tipo R***
  - ◆ Operaciones de almacenamiento (store): ***tipo S***
  - ◆ Operaciones con inmediatos y carga: ***tipo I***
  - ◆ Operaciones con inmediatos largos: ***tipo U***
  - ◆ Saltos condicionales (branches): ***tipo B***
  - ◆ Saltos incondicionales (jump): ***tipo J***

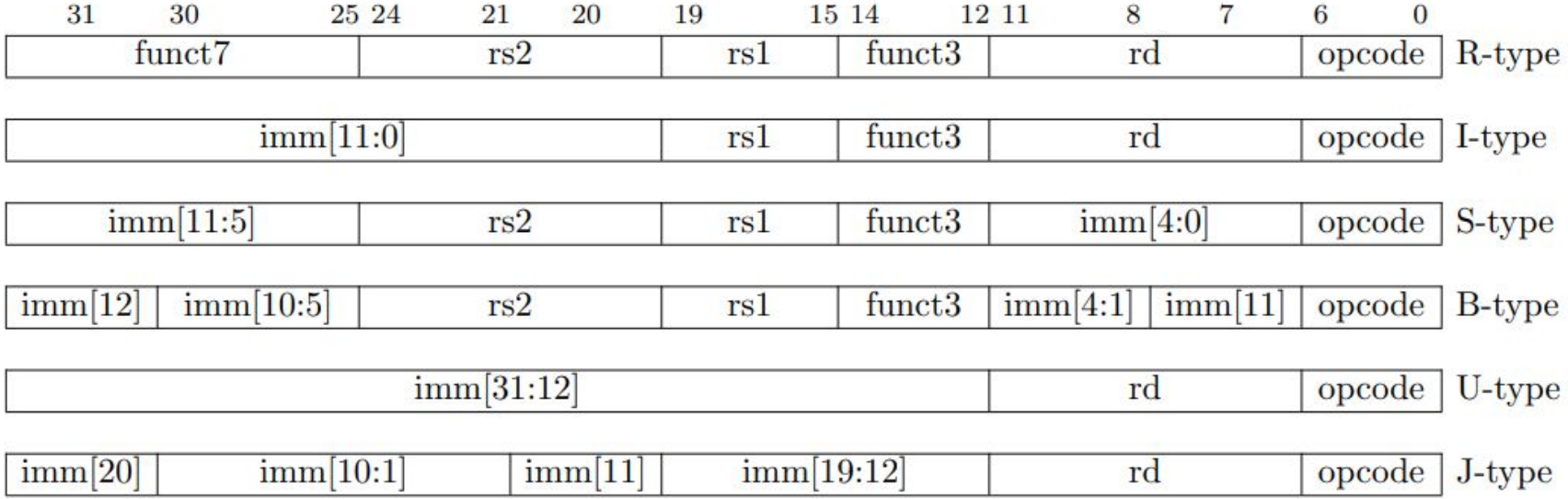
RV32I: Registros

- Total de 32 registros de 32 bits
- Un registro específico para el PC
- Un registro inmutable alambrado a 0 (x0)

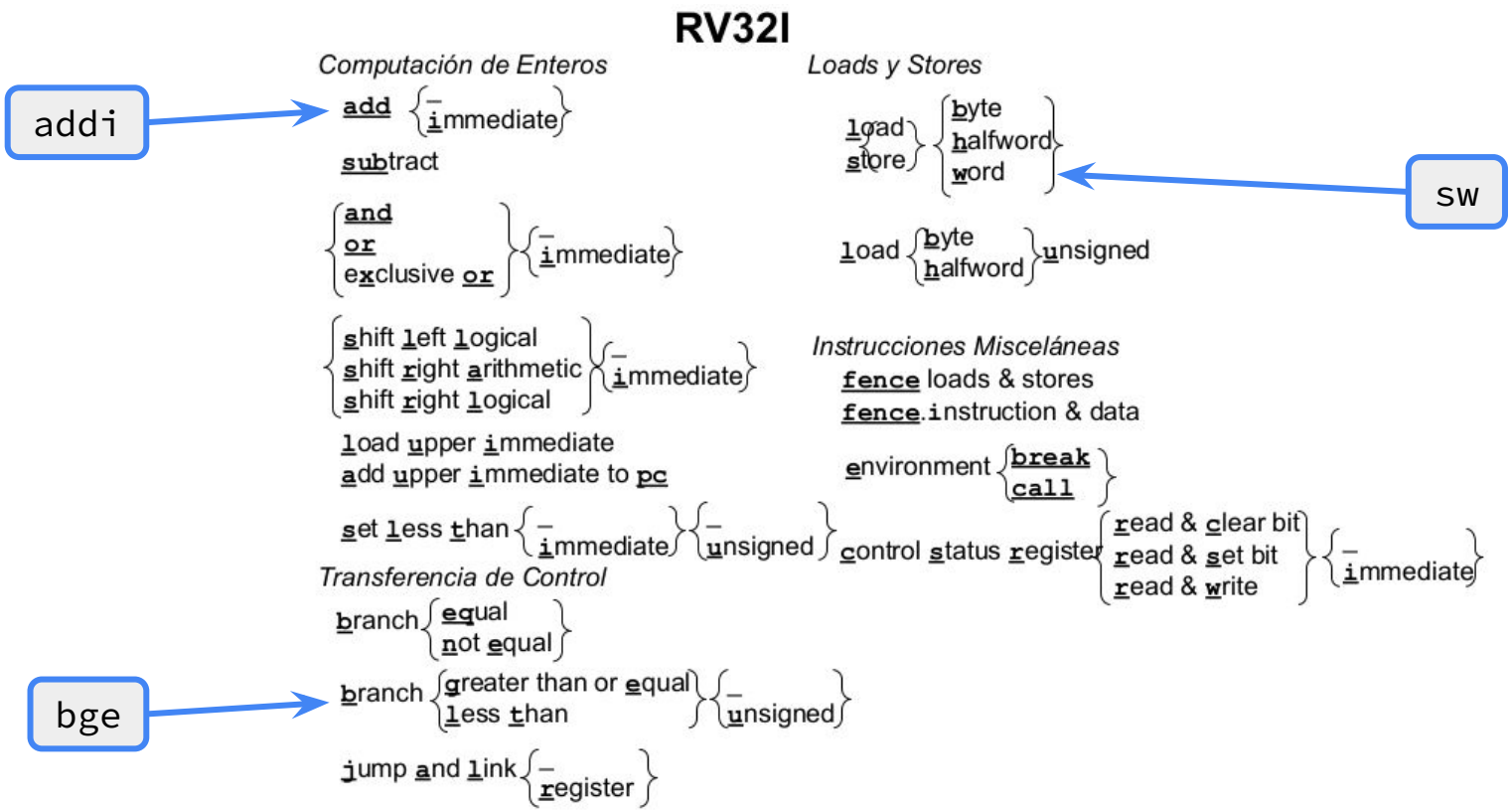


Formato de instrucción

Existen 4 tipos base de formato (R, I, S y B) y dos que varían el formato de cómo se usa el inmediato (U y J).



RV32I: Instrucciones



RV32I: Instrucciones

Arithmetic Operation

Mnemonic	Instruction	Type	Description
ADD rd, rs1, rs2	Add	R	rd ← rs1 + rs2
SUB rd, rs1, rs2	Subtract	R	rd ← rs1 - rs2
ADDI rd, rs1, imm12	Add immediate	I	rd ← rs1 + imm12
SLT rd, rs1, rs2	Set less than	R	rd ← rs1 < rs2 ? 1 : 0
SLTI rd, rs1, imm12	Set less than immediate	I	rd ← rs1 < imm12 ? 1 : 0
SLTU rd, rs1, rs2	Set less than unsigned	R	rd ← rs1 < rs2 ? 1 : 0
SLTIU rd, rs1, imm12	Set less than immediate unsigned	I	rd ← rs1 < imm12 ? 1 : 0
LUI rd, imm20	Load upper immediate	U	rd ← imm20 << 12
AUIP rd, imm20	Add upper immediate to PC	U	rd ← PC + imm20 << 12

Logical Operations

Mnemonic	Instruction	Type	Description
AND rd, rs1, rs2	AND	R	rd ← rs1 & rs2
OR rd, rs1, rs2	OR	R	rd ← rs1 | rs2
XOR rd, rs1, rs2	XOR	R	rd ← rs1 ^ rs2
ANDI rd, rs1, imm12	AND immediate	I	rd ← rs1 & imm12
ORI rd, rs1, imm12	OR immediate	I	rd ← rs1 | imm12
XORI rd, rs1, imm12	XOR immediate	I	rd ← rs1 ^ imm12
SLL rd, rs1, rs2	Shift left logical	R	rd ← rs1 << rs2
SRL rd, rs1, rs2	Shift right logical	R	rd ← rs1 >> rs2
SRA rd, rs1, rs2	Shift right arithmetic	R	rd ← rs1 >> rs2
SLLI rd, rs1, shamt	Shift left logical immediate	I	rd ← rs1 << shamt
SRLI rd, rs1, shamt	Shift right logical imm.	I	rd ← rs1 >> shamt
SRAI rd, rs1, shamt	Shift right arithmetic immediate	I	rd ← rs1 >> shamt

Fuente: https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

Instrucción: ADDI (add immediate)

rd = registro destino
rs1 = registro fuente 1

extensiones

efecto

formato

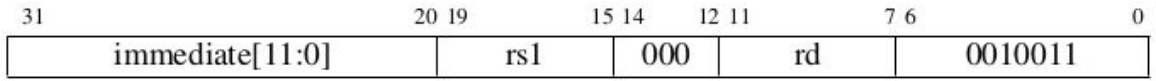
addi rd, rs1, immediate

`x[rd] = x[rs1] + sext(immediate)`

Add Immediate: Tipo I, RV32I y RV64I.

Suma el *inmediato* sign-extended al registro x[*rs1*] y escribe el resultado en x[*rd*]. Overflow aritmético ignorado.

Formas comprimidas: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm




```
addi x1, x2, 17
```

```
addi x1, x2, 2048
```

```
x1 = x2 + sext(17)
```

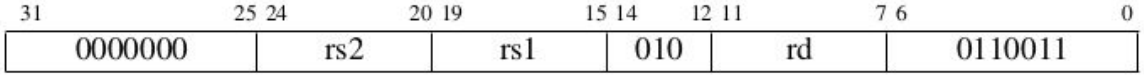
```
No funciona… Por qué?
```

Instrucción: SLT (set if less than)

slt rd, rs1, rs2 $\quad x[rd] = x[rs1] <_s x[rs2]$

Set if Less Than. Tipo R, RV32I y RV64I.

Compara x[*rs1*] con x[*rs2*] como números de complemento a dos, y escribe 1 en x[*rd*] si x[*rs1*] es menor, ó 0 si no.




```
slt x5, x6, x7
```

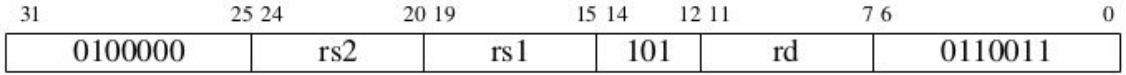
```
if x6 < x7:
    x5 = 1
else:
    x5 = 0
```

Instrucción: SRA (shift right arithmetic)

sra rd, rs1, rs2 `x[rd] = x[rs1] >>s x[rs2]`

Shift Right Arithmetic. Tipo R, RV32I y RV64I.

Corre el registro x[*rs1*] a la derecha por x[*rs2*] posiciones de bits. Los bits liberados son reemplazados por copias del bit más significativo de x[*rs1*], y el resultado es escrito en x[*rd*]. Los cinco bits menos significativos de x[*rs2*] (o seis bits para RV64I) forman la cantidad a correr; los bits altos son ignorados.




```
rs1 = -8      // Binary: 11111111111111111111111111111000 (32-bit signed integer)
rs2 = 2       // Binary: 00000000000000000000000000000010 (nro. de bits a correr)

sra rd, rs1, rs2

rd = rs1 >> rs2
rd = -8 >> 2
rd = -2       // Binary: 11111111111111111111111111111110
```

Instrucciones de acceso a memoria/datos

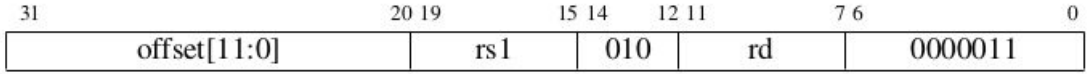
- Palabras de 32 bits (4 bytes)
- Tenemos load y store para palabra, media palabra y byte
- Los datos pueden ser *signed* o *unsigned*

lw rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

Load Word. Tipo I, RV32I y RV64I.

Carga cuatro bytes de memoria en la dirección x[*rs1*] + *sign-extend*(*offset*) y los escribe en x[*rd*]. Para RV64I, el resultado es extendido en signo.

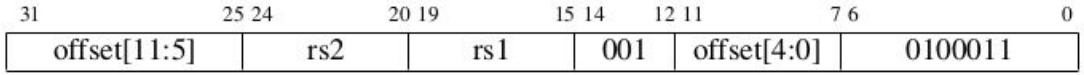
Formas comprimidas: **c.lwsp** rd, offset; **c.lw** rd, offset(rs1)



sh rs2, offset(rs1) `M[x[rs1] + sext(offset)] = x[rs2][15:0]`

Store Halfword. Tipo S, RV32I y RV64I.

Almacena los dos bytes menos significativos del registro x[*rs2*] a memoria en la dirección x[*rs1*] + *sign-extend*(*offset*).



Instrucciones de acceso a memoria/datos

- Tenemos otras instrucciones que no son exactamente instrucciones sino *pseudo-instrucciones*

li rd, immediate `x[rd] = immediate`

Load Immediate. Pseudoinstrucción, RV32I y RV64I.
Carga una constante en x[*rd*], usando tan pocas instrucciones como sea posible. Para RV32I, se extiende a **lui** y/o **addi**; para RV64I, es tan largo como **lui**, **addi**, **slli**, **addi**, **slli**, **addi**, **slli**, **addi**.

la rd, symbol `x[rd] = &symbol`

Load Address. Pseudoinstrucción, RV32I y RV64I.
Carga la dirección de *symbol* en x[*rd*]. Cuando se ensambla *position-independent code*, se extiende a un load del *Global Offset Table*: para RV32I, **auipc** rd, offsetHi luego **lw** rd, offsetLo(rd); para RV64I, **auipc** rd, offsetHi luego **ld** rd, offsetLo(rd). En cualquier otro caso, se extiende a **auipc** rd, offsetHi luego **addi** rd, rd, offsetLo.

RV32I: Instrucciones

Load / Store Operations

Mnemonic	Instruction	Type	Description
LD rd, imm12(rs1)	Load doubleword	I	rd ← mem[rs1 + imm12]
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SD rs2, imm12(rs1)	Store doubleword	S	rs2 → mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → em[rs1 + imm12]

Fuente: https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

Instrucción: LI (load immediate)

```
# Casos fáciles
li x1, 42
li x1, 0xFFFFFFFF
```

```
# Caso más complicado
li x1, 0xFFF

# Se transforma en
lui x1, 0x1
addi x1, x1, -1

# Ejecución lui
x1 = sext(0x1 << 12)
x1 = sext(0x1000)
x1 = 0x00001000

# Ejecución addi
x1 = x1 + sext(-1)
x1 = 0x00001000 + 0xFFFFFFFF
x1 = 0x00000FFF
```

0xFFF -> 0x00000FFF
Sea U = 0x00000, L = 0xFFF (upper/lower)

Cargo U = 0x00000 en la parte alta con **LUI**
Cargo L = 0xFFF en la parte baja con **ADDI**
Luego x1 = (0x00000 << 12) + 0xFFF
x1 = 0x00000000 + 0xFFF = 0x00000FFF

Pero no funciona exactamente así, nos olvidamos que ADDI hace sext() del operando inmediato. Si el valor de L es positivo, entonces la cuenta da bien porque sext(L) = L.

Pero si L es negativo, **ADDI** realiza:
sext(L) = L - 2^12 = L - 0x1000

Entonces compensamos sumando 0x1000 para cancelar ese término en la cuenta final.

0xFFF + 0x1000 -> 0x1FFF
U = 0x00001
L = 0xFFF
x1 = (0x00001 << 12) + sext(0xFFF)
x1 = 0x00001000 + 0xFFFFFFFF = 0x00000FFF

Instrucción: LA, LW (load address, load word)

```
.text:
la x1, numeros
lw x2, 0(x1)

mv a0, x2
li a7, 34
ecall

.data:
numeros: .word 0x11223344, 0x55667788, 0x99AABBCC
```

Instrucción: SW (store word)

```
.text:
li x1, 0x1000
li x2, 0x1337
sw x2, 0(x1)

lw x3, 0(x1)

mv a0, x3
li a7, 34
ecall
```

Saltos incondicionales

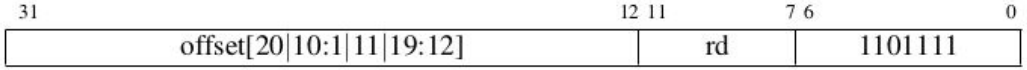
- El `jal` (jump and link) suele usarse para llamar funciones o saltos incondicionales
- Salto relativo al PC
- Hay una versión con offset guardado en registro: `jalr`

jal rd, offset `x[rd] = pc+4; pc += sext(offset)`

Jump and Link. Tipo J, RV32I y RV64I.

Escribe la dirección de la siguiente instrucción (*pc*+4) en x[*rd*], luego asigna al *pc* su valor actual más el *offset* extendido en signo. Si *rd* es omitido, se asume x1.

Formas comprimidas: **c.j** offset; **c.jal** offset

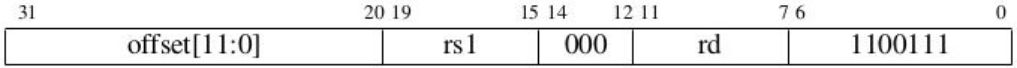


jalr rd, offset(rs1) `t=pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t`

Jump and Link Register. Tipo I, RV32I y RV64I.

Escribe x[*rs1*] + *sign-extend*(*offset*) al *pc*, enmascarando a cero el bit menos significativo de la dirección calculada, luego escribe el valor previo del *pc*+4 en x[*rd*]. Si *rd* es omitido, se asume x1.

Formas comprimidas: **c.jr** rs1; **c.jalr** rs1



Instrucción: JAL (jump and link), JALR

```
.text

main:
jal x1, calcular1

imprimir:
li a7, 34
ecall # Imprime el valor de a0 en hexa

li a0, 0
li a7, 93
ecall # Termina la ejecución

calcular1:
jal x2, calcular2
jalr x0, x1, 0 # ret a main

calcular2:
li a0, 0x1337
jalr x0, x2, 0 # ret a calcular1
```

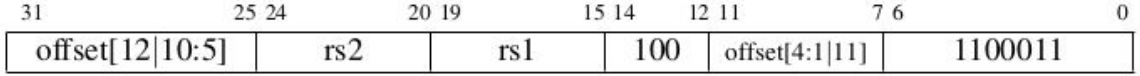
Saltos condicionales (branching)

- Para *saltos condicionales* tenemos saltos que comparan dos registros si son iguales (`beq`), distintos (`bne`), mayor igual (`bge`) o menor (`blt`). ¿Cómo se compara esto con lo que pasaba en la arquitectura de Orga1?
- Dos versiones sin signo: `bltu y bgeu`

blt rs1, rs2, offset `if (rs1 <s rs2) pc += sext(offset)`

Branch if Less Than. Tipo B, RV32I y RV64I.

Si el registro x[*rs1*] es menor que x[*rs2*], tratando los valores como números de complemento a dos, asignar al *pc* su valor actual más el *offset* sign-extended.



RV32I: Instrucciones

Branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2` `  pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2` `  pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2` `  pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `  pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `  pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `  pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

Fuente: https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8

Instrucción: BLT (branch less than)

```
.text
li x1, 42
li x2, 100
blt x1, x2, es_menor
j es_mayor

es_menor:
la a0, txt_menor
j exit

es_mayor:
la a0, txt_mayor
j exit
```

```
exit:
li a7, 4
ecall # Imprime el string en la dirección [a0]

li a0, 0
li a7, 93
ecall # Termina la ejecución

.data
txt_menor: .string "es menor"
txt_mayor: .string "es mayor"
```

- ¿?
  - ¿Cómo detectarían si el resultado de una operación es negativo?
  - ¿Cómo detectarían si el resultado de una operación es exactamente 0?
  - ¿Cómo detectarían si el resultado de una operación tuvo overflow?
  - ¿Cómo moverían valores entre registros?
  - Esquema de registros
    - ¿Stack Pointer dedicado?
    - Registro x0

Ejercicio integrador

Tenemos una imagen (array de pixeles) en formato RGB de 24 bits contiguos en memoria. Cada color tiene 8 bits. Queremos procesar la imagen y buscar el máximo valor rojo entre todos los pixeles.

```
find_max_red(&pixels, num_pixels)
```

Ejercicio integrador

```
def find_max_red(pixels, num_pixels):
    max_red = 0
    for i in range(num_pixels):
        pixel = pixels[i] # RRGGBB
        red = (pixel >> 16) & 0xFF
        if red > max_red:
            max_red = red
    return hex(max_red)

print(find_max_red([
    0x112233,
    0x445566,
    0x778899,
    0xAABBCC,
    0xDDEEFF,
], 5))
```

Ejercicio integrador

```
.text:

# Inicializamos t1 con la dirección donde están guardados los pixeles.
la t1, pixels

# Cantidad de pixels.
li t3, 5

# Inicializamos el valor máximo de rojo en 0.
li t4, 0
```

```
loop:
# while (t3-- > 0) {...}
beqz t3, exit
addi t3, t3, -1

# Cargamos 1 byte que corresponde al pixel rojo.
lb t2, 0(t1)

# Aplicamos la máscara para quedarnos solo con el byte que nos interesa.
# Esto es necesario si el valor cargado es >= 128, pues en complemento a 2
# es un número negativo, y RISC-V realiza extensión de signo al byte cargado
# para llenar el registro entero de 32 bits.
andi t2, t2, 0xFF

# Vemos si este valor de rojo es el nuevo máximo.
blt t2, t4, not_max
mv t4, t2

not_max:
# Avanzamos 3 bytes al siguiente pixel.
addi t1, t1, 3

j loop
```

```
exit:
# Imprime el rojo máximo.
mv a0, t4
li a7, 34
ecall

# Termina el programa.
li a0, 0
li a7, 93
ecall

.data:
pixels:
.byte 0x11, 0x22, 0x33 # RGB
.byte 0x44, 0x55, 0x66 # RGB
.byte 0x77, 0x88, 0x99 # RGB
.byte 0xAA, 0xBB, 0xCC # RGB
.byte 0xDD, 0xEE, 0xFF # RGB
```