

Descripción funcional de un microprocesador

Sistemas con Microprocesadores

Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

<http://www.microprocesadores.unt.edu.ar/procesadores>

Cronograma

Actividad	Inicio	Descripción	Completa
Presentación	24/08	Presentación y reglamento	✓
Tema 1	24/08	Estructura de las computadoras	✓
Tema 2	26/08	Proyecto con un microcontrolador	✓
Tema 3	31/08	Descripción funcional del ISA RV32IM	←
Tema 4	07/09	Programación en lenguaje ensamblador	
Tema 5	18/09	Diseño del camino de datos de un procesador	
Tema 6	30/09	Diseño de la unidad de control del procesador	
Tema 7	05/10	Tecnologías y sistema de memoria	
Tema 8	09/10	Dispositivos de entrada/salida	
Tema 9	19/10	Entradas/salidas digitales (GPIO)	
Tema 10	28/10	Generalidades de las interrupciones	
Tema 11	04/11	Interrupciones y excepciones en RISC-V	
Tema 12	13/11	Manejo de señales con temporizadores	
Tema 13	18/11	Transferencias de datos	
Proyectos	30/11	Seminarios de proyectos	

Temas que veremos

- ▶ **ISA del RISC-V**
 - ▶ Características generales
 - ▶ Formatos de instrucciones
 - ▶ Modos de direccionamiento
 - ▶ Conjunto de Instrucciones
 - ▶ Pseudoinstrucciones
 - ▶ Directivas al ensamblador

¿Que es un ISA?

- ▶ Es toda la información que se debe conocer para programar un procesador
- ▶ Es un nivel de abstracción que permite separar el desarrollo de software del hardware subyacente
- ▶ Es independiente de la implementación, por lo que diferentes procesadores pueden utilizar el mismo ISA

Componentes de un ISA

- ▶ Celdas de almacenamiento
 - ▶ Registros de propósito general y especiales
 - ▶ Organización de la memoria
 - ▶ Acceso a los dispositivos de entrada/salida
- ▶ Formatos de las instrucciones
 - ▶ Tamaño y significado de los campos
 - ▶ Modos de direccionamiento soportados

Componentes de un ISA

- ▶ Naturaleza del ciclo de la instrucción
 - ▶ Tareas que el procesador completa automáticamente independientemente de la instrucción que ejecuta
- ▶ Conjunto de instrucciones
 - ▶ Definición de cada instrucción disponible
 - ▶ Que formatos utiliza cada una
 - ▶ Como afectan al almacenamiento

¿Porqué RISC-V?

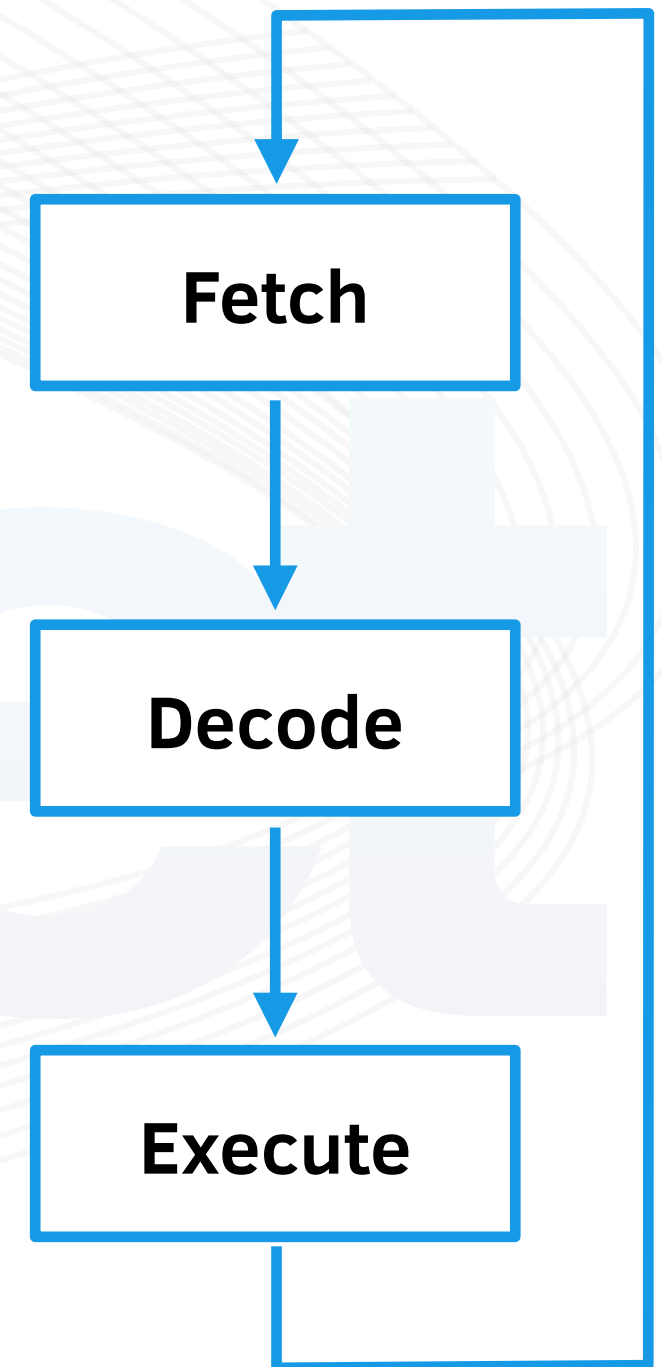
- ▶ Es un ISA novedoso, abierto y gratuito, que es propiedad de una fundación
- ▶ Esta diseñado a partir de los aciertos y errores de las máquinas previas
- ▶ Está pensado para todo el espectro de procesadores
- ▶ Es modular, por lo que define un conjunto de extensiones que pueden o no implementarse

Convención de nombres

- ▶ El formato de los ISA es RV[###][abc..xyz]
- ▶ RV indica que es un ISA RISC-V
- ▶ [###] indica el ancho de palabra del procesador, puede ser 32, 64 o 128
- ▶ [abc...xyz] indica el conjunto de extensiones soportado
- ▶ RV32I es la única extensión obligatoria y es el núcleo base para el manejo de enteros

Ciclo de la instrucción

- ▶ El procesador es una maquina de estados finito que repite permanentemente una secuencia
- ▶ Busca una instrucción en memoria
- ▶ Determina las acciones necesarias para que la instrucción genere el resultado esperado
- ▶ Ejecuta las acciones determinadas en el paso anterior para obtener el resultado esperado
- ▶ Es equivalente a un interprete del ISA



Que se especifica en una instrucción

add x3, x2, x1

- ▶ ¿Qué operación realizar?
 - Op-code: add, load, branch, etc

add x3, **x2**, **x1**

- ▶ ¿Dónde están los operandos?
 - En registros del CPU, en memoria, E/S
 - Pueden ser parte de la instrucción.

add **x3**, x2, x1

- ▶ ¿Dónde se guarda el resultado?
 - En un registro, memoria o E/S

add x3, x2, x1
j **lazo**

- ▶ ¿Como continua el programa?
 - Lugar de memoria donde buscar la siguiente instrucción

Modos de direccionamiento

- ▶ Para una misma instrucción los operandos se puede especificar de diferentes formas
- ▶ En una suma un operando puede ser una constante o puede ser una variable, que puede estar en un registro o en memoria
- ▶ El modo de direccionamiento es el soporte en hardware para que la instrucción busque el operando en diferentes lugares

Directivas al ensamblador

- ▶ Las directivas al ensamblador son comandos que se insertan en el código fuente
- ▶ No se traducen a código objeto como las instrucciones porque sirven para controlar el proceso de traducción
- ▶ En general permiten gestionar la memoria, definir símbolos y organizar la estructura del programa
- ▶ Suelen comenzar con un punto

Clases de instrucciones

- ▶ Movimientos de datos
 - ▶ Permiten transferir datos entre memoria y los registros
 - ▶ Tiene siempre un destino y una fuente
 - ▶ En las cargas la fuente es la memoria y el destino un registro
 - ▶ En los almacenamientos la fuente es un registro y el destino la memoria
 - ▶ Tambien existe un movimiento entre registros
 - ▶ En RV32I no existen transferencias entre memoria

Clases de instrucciones

- ▶ **Procesamiento**
 - ▶ Calculan un resultado usando operaciones matemáticas o lógicas
 - ▶ Tienen siempre un destino y, según la operación, una o dos fuentes
 - ▶ En RV32I el destino y una fuente son siempre registros, pero la otra fuente puede ser un registro o una constante
- ▶ **Control de flujo**
 - ▶ Alteran el flujo normal de ejecución del programa
 - ▶ Este cambio en la ejecución puede ser condicional o incondicional

Memoria y registros

- ▶ RV32I es una arquitectura tipo load-store
- ▶ Las únicas operaciones que se pueden hacer con la memoria son:
 - Leer una instrucción
 - Leer un dato y copiarlo en un registro
 - Escribir un dato desde un registro
- ▶ El procesador tiene un registro especial (PC) que señala la dirección de memoria donde debe buscar la siguiente instrucción del programa

Registros

- ▶ El procesador tiene 32 registros de propósito general (x0 a x31) que el programador puede usar libremente
- ▶ El primero de estos registros (x0) tiene siempre almacenado la constante 0 y no se puede escribir
- ▶ Los otros 31 registros (x1 a x31) pueden almacenar variables
- ▶ Las instrucciones que transforman datos solo pueden operar sobre los registros

Memoria

- ▶ La memoria funciona como una gran tabla unidimensional (vector)
- ▶ En cada dirección de memoria se almacena un numero de 8 bits
- ▶ Para almacenar números de 16 o 32 bits se utilizan 2 o 4 direcciones consecutivas
- ▶ En una misma operación se pueden leer datos de 8, 16 o 32 bits siempre que están alineados
 - ▶ Si es de 16 bits debe comenzar en una dirección par
 - ▶ Si es de 32 bits en una dirección múltiplo de 4

Modos de direccionamiento

- ▶ Una de los objetivos de diseño en el ISA es minimizar cantidad de modos de direccionamiento disponibles
 - ▶ Los modos complejos interfieren con las técnicas para acelerar la ejecución de las instrucciones en los procesadores complejos
 - ▶ Los modos complejos se pueden emular usando secuencias de instrucciones mas simples
- ▶ En consecuencia, en RV32I, los programas suelen ser mas lagos que en otros ISA (por ejemplo ARM Thumb2)
 - ▶ Solo soporta tres modos de direccionamiento

Direcccionamiento inmediato

- ▶ El operando es una constante
- ▶ Se distingue porque el mnemónico termina con una letra "i" de inmediato

li x1,100 # x1 ← 100

- ▶ ¿Ventajas del direccionamiento inmediato?

Direcccionamiento de registro

- ▶ El operando es una variable y se encuentra en un registro del procesador
- ▶ En este procesador es el modo de direccionamiento mas utilizado

add x1, x1, x2 # $x1 \leftarrow x1 + x2$

- ▶ ¿Ventajas del direccionamiento de registros?

Direccionamiento indexado

- ▶ El operando es una variable y se encuentra en memoria
- ▶ La dirección de memoria del operando se obtiene al sumar un registro con una constante

`lw x2, 100(x1)    # x2 ← M[x1 + 100]`

- ▶ El registro se llama puntero base y la constante desplazamiento
- ▶ ¿Ventajas del direccionamiento indexado?

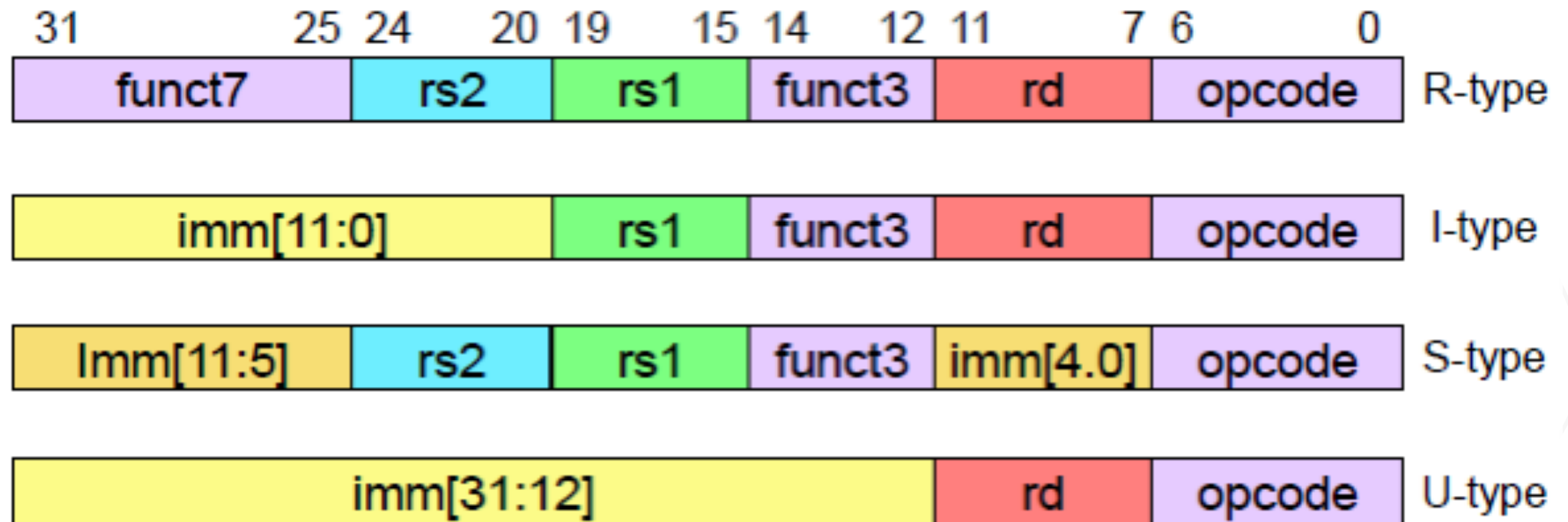
Formato de las instrucciones

- ▶ Las instrucciones de RV32I se codifican siempre utilizando 32 bits
- ▶ La instrucción se divide en campos que brindan la misma información que la instrucción en ensamblador
- ▶ Puede haber mas de un formato para la misma instrucción dependiendo de los modos de direccionamiento que soporte

Formato de las instrucciones

- ▶ En RV32I se definen 4 formatos base:
- ▶ **Tipo R:** utilizado para las instrucciones aritméticas
- ▶ **Tipo I:** utilizado para las instrucciones con constantes y las cargas de datos
- ▶ **Tipo S:** utilizado para las operaciones de almacenamiento
- ▶ **Tipo U:** utilizado para las instrucciones con constantes largas

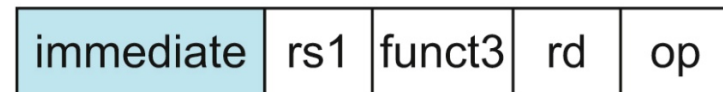
Formato de las instrucciones



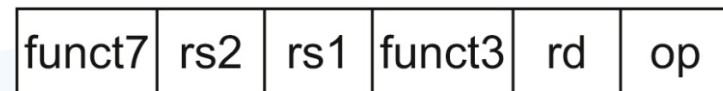
- ▶ Todos utilizan un opcode de 7 bits
- ▶ El opcode define grupos de instrucciones que se distinguen por el campo de función

Modos de direccionamiento

1. Immediate addressing



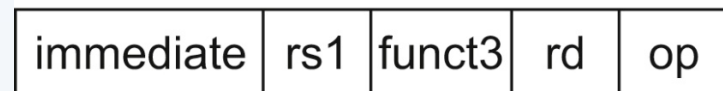
2. Register addressing



Registers

Register

3. Base addressing



Memory

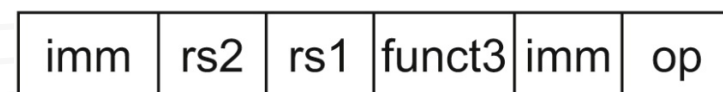
Register



Byte Halfword

word

4. PC-relative addressing



Memory

PC

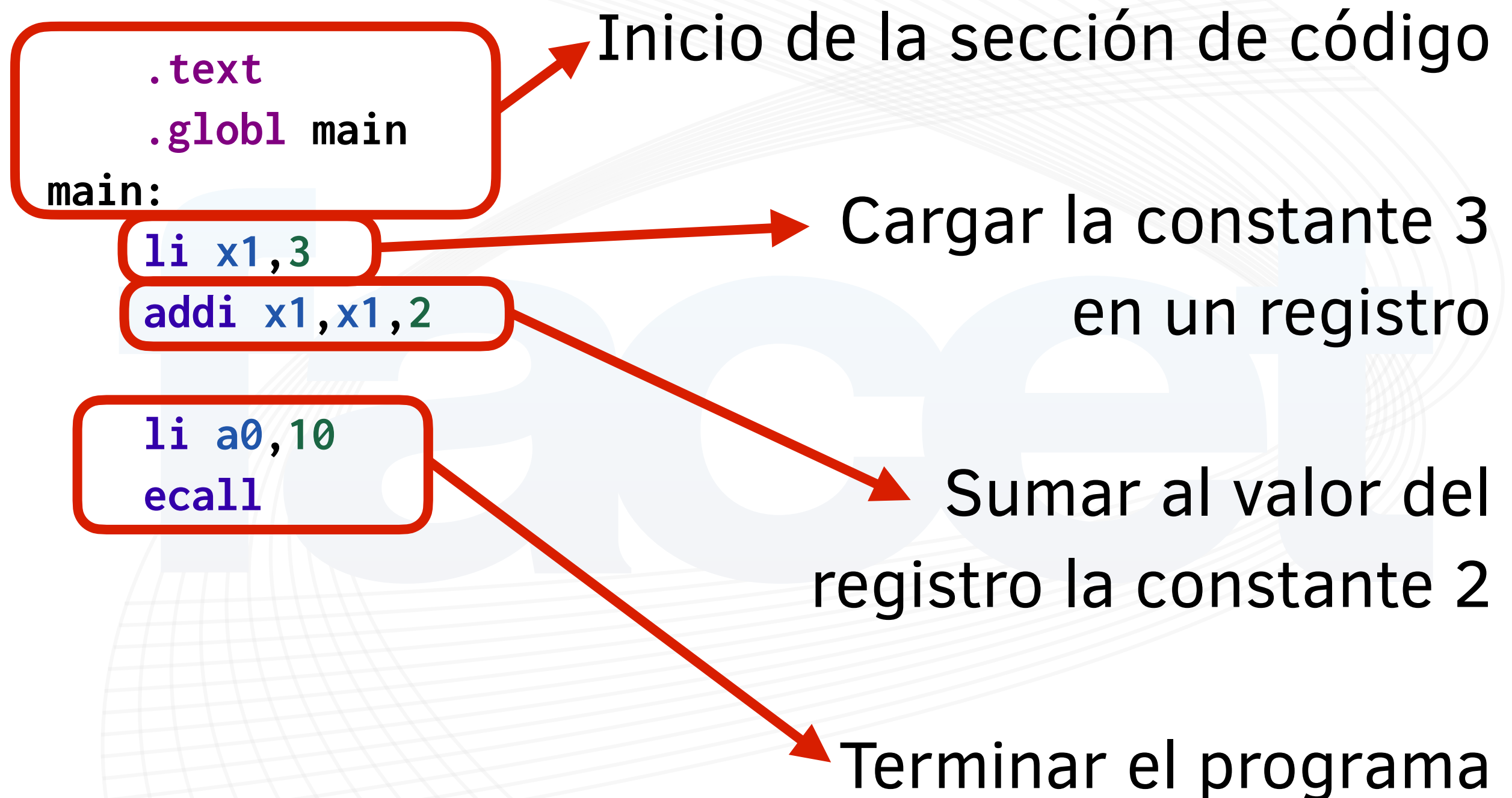


Word

Directivas al ensamblador

- ▶ **.section .text / .data**
 - ▶ Determina la memoria de destino para código o datos
- ▶ **.align**
 - ▶ Realinea el código o dato siguiente a una dirección múltiplo de 4.
- ▶ **.globl etiqueta**
 - ▶ Declara la etiqueta como global (visible por el resto del proyecto)
- ▶ **.word / .half / .byte**
 - ▶ Almacena constantes en memoria en 4, 2 y 1 bytes
- ▶ **.space n**
 - ▶ Reserva una cantidad de memoria igual a n bytes.
- ▶ **.string "cadena"**
 - ▶ Almacena la cadena como una constante en memoria

Sumando dos números



Funcionamiento del simulador

The image displays the Venus simulator interface, which is used for simulating MIPS assembly code. The interface is divided into several sections:

- Editor:** Contains the assembly code being simulated. The code is as follows:

```
1      .data
2  numero_a: .word 2
3  numero_b: .word 3
4  resultado: .word 0
5
6      .text
7  .globl main
8  main:  lw t0,numero_a      # Se carga el primer operando desde memoria
9        lw t1,numero_b      # Se carga el primer operando desde memoria
10       add t0, t0, t1        # Se calcula el resultado como la suma de los operandos
11       la t1,resultado      # Se carga la dirección para almacenar el resultado
12       sw t0,0(t1)          # Se almacena el resultado en la dirección indicada
13  fin:   li a0,10           # Selección del servicio de sistema para terminar el programa
14        ecall              # Termina el programa y retorna al sistema operativo
```
- Simulator:** Displays the execution state, including the PC, Machine Code, Basic Code, and Original Code. The current instruction being executed is highlighted in red:

PC	Machine Code	Basic Code	Original Code
0x0	0x10000297	auipc x5 65536	main: lw t0,numero_a # Se carga el primer operando desde memoria
0x4	0x0002A283	lw x5 0(x5)	main: lw t0,numero_a # Se carga el primer operando desde memoria
0x8	0x10000317	auipc x6 65536	lw t1,numero_b # Se carga el primer operando desde memoria
0xc	0xFFC32303	lw x6 -4(x6)	lw t1,numero_b # Se carga el primer operando desde memoria
0x10	0x006282B3	add x5 x5 x6	add t0, t0, t1 # Se calcula el resultado como la suma de los operandos
0x14	0x10000317	auipc x6 65536	la t1,resultado # Se carga la dirección para almacenar el resultado
0x18	0xFF430313	addi x6 x6 -12	la t1,resultado # Se carga la dirección para almacenar el resultado
- Registers:** Displays the current values of the registers. The registers are listed on the right side of the interface, with their current values shown in hexadecimal.
- Console Output:** Displays the output of the program, which is currently empty.

Movimientos de datos

- ▶ **load:** carga en un registro un dato de memoria
 - ▶ Puede ser un byte, media palabra o una palabra, con o sin signo
 - ▶ El dato se almacena en el registro siempre como una palabra extendiendo en cero, o en signo, el valor leído de memoria
- ▶ **store:** almacena en memoria el dato de un registro
 - ▶ Puede ser un byte, media palabra o una palabra
 - ▶ El dato se almacena truncando el valor del registro
 - ▶ La operación es la misma para datos con o sin signo
- ▶ El dato en memoria siempre se accede en forma indexada sumando una constante a un registro

Set de instrucciones

Cargas	Formato	Instrucción	
Load Byte	I	lb	rd,rs1,imm
Load Halfword	I	lh	rd,rs1,imm
Load Byte Unsigned	I	lbu	rd,rs1,imm
Load Halfword Unsigned	I	lhu	rd,rs1,imm
Load Word	I	lw	rd,rs1,imm

Almacenamientos	Formato	Instrucción	
Store Byte	S	sb	rd,rs1,imm
Store Halfword	S	sh	rd,rs1,imm
Sote Word	S	sw	rd,rs1,imm

Ejemplo

```
.data # Define la sección de variables
num_a: .byte 0xE1,0xE3,0x65,0x87
num_b: .space 4
```

```
.text # Define la sección de código
.globl main
```

```
main: la x3,num_a # x3 ← 0x1000.0000
      la x4,num_b # x4 ← 0x1000.0004

      lb x5,0(x3) # x5 ← 0xFFFF.FFE1
      sb x5,0(x4) # M(0x1000.0004) ← 0xE1

      lbu x5,0(x3) # x5 ← 0x0000.00E1
      sb x5,0(x4) # M(0x1000.0004) ← 0xE1

      lh x5,0(x3) # x5 ← 0xFFFF.E3E1
      sh x5,0(x4) # M(0x1000.0004) ← 0xE1
               # M(0x1000.0005) ← 0xE3

      lw x5,0(x3) # x5 ← 0x8765.E3E1
      sw x5,0(x4) # M(0x1000.0004) ← 0xE1
               # M(0x1000.0005) ← 0xE3
               # M(0x1000.0006) ← 0x65
               # M(0x1000.0007) ← 0x87
```

Dirección	Contenido
0x1000.0000	0xE1
0x1000.0001	0xE3
0x1000.0002	0x65
0x1000.0003	0x87

Dirección	Contenido
0x1000.0004	0xE1

Dirección	Contenido
0x1000.0004	0xE1

Dirección	Contenido
0x1000.0004	0xE1
0x1000.0005	0xE3

Dirección	Contenido
0x1000.0004	0xE1
0x1000.0005	0xE3
0x1000.0006	0x65
0x1000.0007	0x87

Instrucciones aritméticas

- ▶ **add**: calcula la suma de dos registros y almacena el resultado en un tercer registro
- ▶ **sub**: calcula la resta de dos registros y almacena el resultado en un tercer registro
- ▶ **addi**: calcula la suma de un registro con una constante y almacena el resultado en un segundo registro
- ▶ Siempre operan sobre datos de 32 bits
- ▶ No existe **subi**: no es necesaria

Instrucciones lógicas

- ▶ **and, or, xor**: calcula la operación lógica correspondiente, bit a bit, usando el contenido de dos registros y almacena el resultado en un tercer registro
- ▶ **andi, ori, xori**: calcula la operación lógica correspondiente, bit a bit, usando el contenido de un registro y una constante, y almacena el resultado en un segundo registro
- ▶ Siempre operan sobre datos de 32 bits
- ▶ La constante de 12 bits siempre se extiende en signo

Mascaras

- ▶ Las operaciones lógicas se utilizan para manipular una parte de un dato sin alterar el resto
- ▶ Se opera una variable de valor desconocido con una constante (o variable) de valor conocido

	AND								OR								XOR							
	d ₃		d ₂		d ₁		d ₀		d ₃		d ₂		d ₁		d ₀		d ₃		d ₂		d ₁		d ₀	
Variable	a		b		c		d		a		b		c		d		a		b		c		d	
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Constante	0		1		0		1		0		1		0		1		0		1		0		1	
Resultado	0	0	0	1	0	0	0	1	0	1	1	1	0	1	1	1	0	1	1	0	0	1	1	0
	0		b		0		d		a		1		c		1		a		1-b		c		1-d	

Aritméticas con constantes

- ▶ **lui**: carga un registro con una constante de 20 bits.
 - ▶ La constante se completa con ceros en la parte menos significativa para obtener el valor de 32 bits que se carga en el registro.
 - ▶ Combinada con **addi** o **ori** permite cargar una constante arbitraria de 32 bits en un registro utilizando dos instrucciones.
- ▶ **auipc**: suma una constante de 20 bits al pc y almacena el resultado en un registro.
 - ▶ La constante se completa con ceros en la parte menos significativa para obtener el valor de 32 bits que se suma al PC.
 - ▶ Combinada con **addi** o **ori** permite cargar una constante una dirección de memoria arbitraria de 32 bits en un registro utilizando dos instrucciones
 - ▶ Combinada con las instrucciones de salto permite alcanzar cualquier dirección de memoria en dos instrucciones

Set de instrucciones

Aritméticas	Formato	Instrucción
Add	R	add rd,rs1,rs2
Add Immediate	I	addi rd,rs1,imm
Subtract	R	sub
Load Upper Immediate	U	lui rd,imm
Add Upper Immediate to PC	U	auipc rd,imm

Lógicas	Formato	Instrucción
XOR	R	xor rd,rs1,rs2
XOR Immediate	I	xori rd,rs1,imm
OR	R	or rd,rs1,rs2
OR Immediate	I	ori rd,rs1,imm
AND	R	and rd,rs1,rs2
AND Immediate	I	andi rd,rs1,imm

Set de instrucciones

Corrimientos	Formato	Instrucción
Shift Left Logical	R	sll rd,rs1,rs2
Shift Left Logical Immediate	I	slli rd,rs1,shamt
Shift Right Logical	R	srl rd,rs1,rs2
Shift Right Logical Immediate	I	srl rd,rs1,shamt
Shift Right Arithmetic	R	sra rd,rs1,rs2
Shift Right Arithmetic Immediate	I	srai rd,rs1,shamt

Comparaciones	Formato	Instrucción
Set Less Than	R	slt rd,rs1,rs2
Set Less Than Immediate	I	slti rd,rs1,imm
Set Less Than Unsigned	R	situ rd,rs1,rs2
Set Less Than Immediate Unsigned	I	sltiu rd,rs1,imm

Set de instrucciones

Saltos Condicionales	Formato		Instrucción
Branch Equal	I	beq	rd,rs1,imm
Branch Not Equal	I	bne	rd,rs1,imm
Branch Less Than	I	blt	rd,rs1,imm
Branch Greater or Equal	I	bge	rd,rs1,imm
Branch Less Than Unsigned	I	bltu	rd,rs1,imm
Branch Greater or Equal Unsigned	I	bgeu	rd,rs1,imm

Saltos Incondicionales	Formato		Instrucción
Jump And Link	J	jal	rd,imm
Jump And Link Register	I	jalr	rd,rs1,imm

Set de instrucciones (Extensión M)

Corrimientos	Formato	Instrucción	
Multiply	R	mul	rd,rs1,rs2
Multiply High	R	mulh	rd,rs1,rs2
Multiply High Unsigned	R	mulhu	rd,rs1,rs2
Multiply High Signed with Unsigned	R	mulhsu	rd,rs1,rs2
Divide	R	div	rd,rs1,rs2
Divide Unsigned	R	divu	rd,rs1,rs2
Remainder	R	rem	rd,rs1,rs2
Remainder Unsigned	R	remu	rd,rs1,rs2

Pseudo-instrucciones

- ▶ Son “instrucciones” que están disponibles para el programador, pero que en realidad no son parte del ISA
- ▶ Suelen facilitar en gran medida la lectura y la escritura de programas, y de compiladores, sin incrementar los costos
- ▶ El ensamblador las reemplaza por una o dos instrucciones reales del ISA que generan el mismo resultado
- ▶ Son combinaciones ingeniosas de instrucciones normales, y muchas de ellas aprovechan el registro x0 como operando

Set de pseudo-instrucciones

Cargas	Pseudoinstrucción	Instrucción
Move	<code>mv rd,rs</code>	<code>addi rs,rs,0</code>
Load Immediate	<code>li rd,immediate</code>	<code>lui rd,immediate[31:12]</code> <code>addi rd,rd,immediate[11:0]</code>
Load Address	<code>la rd,symbol</code>	<code>auipc rd,symbol[31:12]</code> <code>addi rd,rd,symbol[11:0]</code>
Load Local Address	<code>lla rd,symbol</code>	<code>auipc rd,symbol[31:12]</code> <code>addi rd,rd,symbol[11:0]</code>
Load Byte	<code>lb rd,symbol</code>	<code>auipc rd,symbol[31:12]</code> <code>lb rd,symbol[11:0](rd)</code>
Load Half	<code>lh rd,symbol</code>	<code>auipc rd,symbol[31:12]</code> <code>lh rd,symbol[11:0](rd)</code>
Load Word	<code>lw rd,symbol</code>	<code>auipc rd,symbol[31:12]</code> <code>lw rd,symbol[11:0](rd)</code>

Set de pseudo-instrucciones

Almacenamientos	Pseudoinstrucción	Instrucción
Store Byte	sb rd,symbol,rt	auipc rt,symbol[31:12] sb rd,symbol[11:0](rt)
Store Half	sh rd,symbol,rt	auipc rt,symbol[31:12] sh rd,symbol[11:0](rt)
Store Word	sw rd,symbol,rt	auipc rt,symbol[31:12] sw rd,symbol[11:0](rt)

Aritméticas	Pseudoinstrucción	Instrucción
No Operation	nop	addi x0,x0,0
Negative	neg rd,rs	sub rd,x0,rs
Not	not rd,rs	xori rd,rs,-1

Set de pseudo-instrucciones

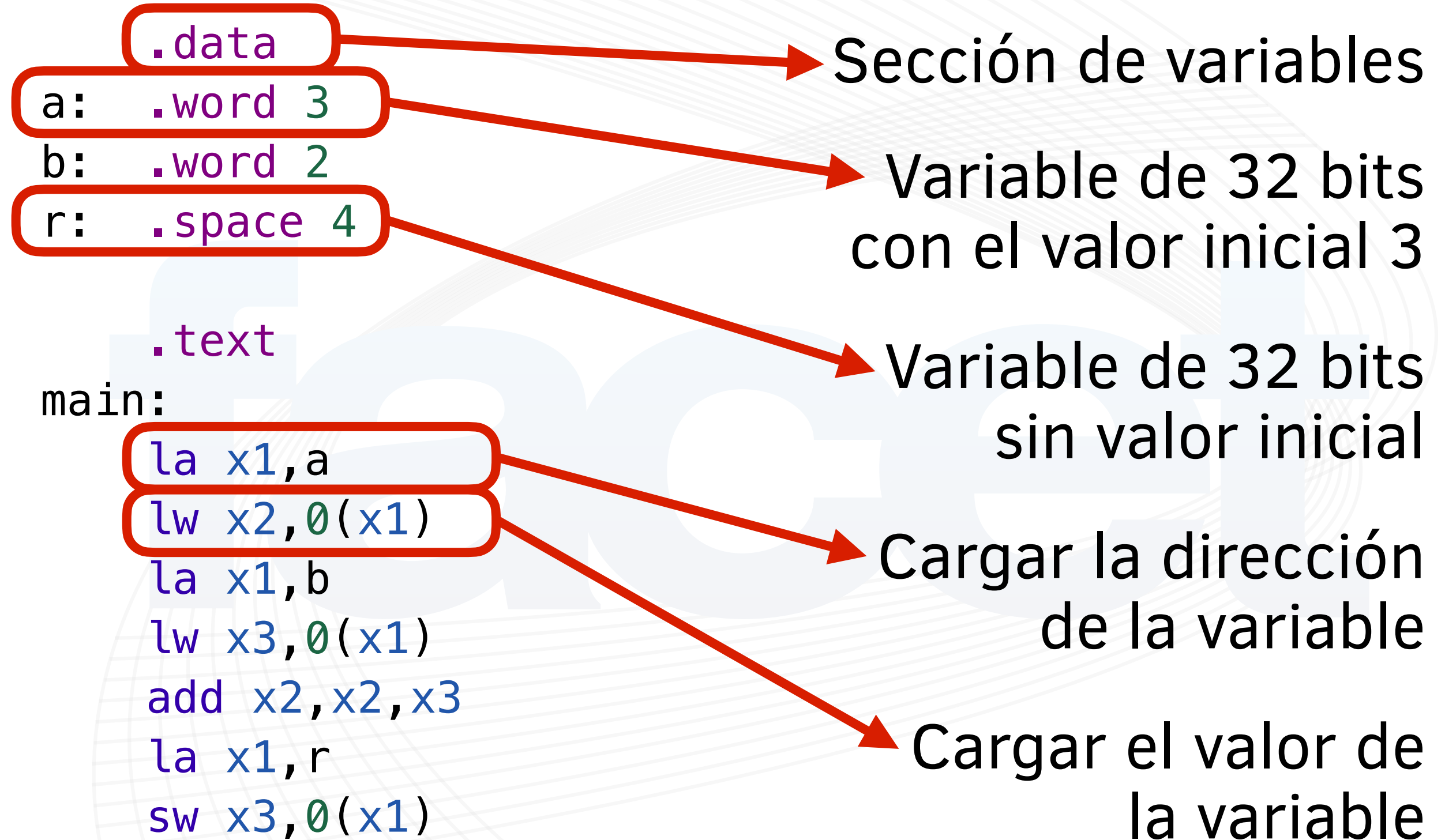
Comparaciones	Pseudoinstrucción	Instrucción
Set Equal Zero	<code>seqz rd,rs</code>	<code>sltiu rd,rs,1</code>
Set Not Equal Zero	<code>snez rd,rs</code>	<code>sltu rd,x0,rs</code>
Set Less Than Zero	<code>sltz rd,rs</code>	<code>sltu rd,x0,rs</code>
Set Greater Than Zero	<code>sltz rd,rs</code>	<code>sltu rd,rs,x0</code>

Salto incondicional	Pseudoinstrucción	Instrucción
Jump	<code>j offset</code>	<code>jal x0,offset</code>
Jump Register	<code>jr rs</code>	<code>jalr x0,rs,0</code>
Jump and Link	<code>jal offset</code>	<code>jal x1,offset</code>
Jump and Link Register	<code>jalr rs</code>	<code>jalr x1,rs,0</code>
Return	<code>ret</code>	<code>jalr x0,x1,0</code>

Set de pseudo-instrucciones

Saltos condicionales	Pseudoinstrucción	Instrucción
Branch Greater Than	bgt rs,rt,offset	blt rt,rs,offset
Branch Less or Equal	ble rs,rt,offset	bge rt,rs,offset
Branch Greater Than Unsigned	bgtu rs,rt,offset	bltu rt,rs,offset
Branch Less or Equal Unsigned	bleu rs,rt,offset	bgeu rt,rs,offset
Branch Equal Zero	beqz rs,offset	beq rs,x0,offset
Branch Not Equal Zero	bnez rs,offset	bne rs,x0,offset
Branch Less or Equal Zero	bleh rs,offset	bge x0,rs,offset
Branch Greater or Equal Zero	bgez rs,offset	bge rs,x0,offset
Branch Less Than Zero	bltz rs,offset	blt x0,rs,offset
Branch Greater Than Zero	bgtz rs,offset	blt rs,x0,offset

Sumando dos variables



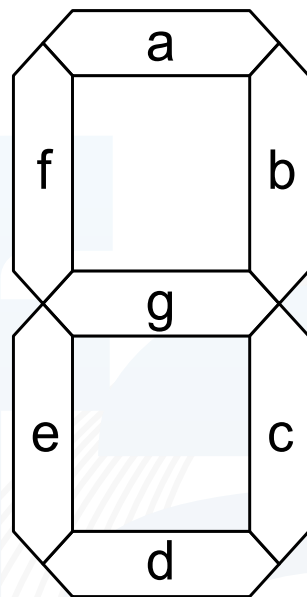
Conversor de BCD a siete segmentos

- ▶ Se debe tomar un número de varios dígitos, almacenados en un bloque de bytes en M terminado por un valor **0xFF**
- ▶ El número está almacenado en bytes en RAM a partir de la dirección **origen**
- ▶ El número está en BCD sin compactar (cada dígito decimal ocupa un byte)
- ▶ Se debe traducir cada dígito a su representación para un display de siete segmentos, generando en M un bloque que también finaliza en **0xFF**
- ▶ Los dígitos convertidos se deben almacenar a partir de la dirección **destino**

Estructura de datos y algoritmo

- ▶ Estructura de datos:
 - ▶ Bloque a convertir desde **origen** (máximo 16 bytes)
 - ▶ Bloque convertido desde **destino**
 - ▶ Tabla de Conversión
 - ▶ Carácter de terminación para ambos: **0xFF**
- ▶ Algoritmo:
 - ▶ Tomar números de origen uno por uno
 - ▶ Convertirlos mediante tabla
 - ▶ Guardarlos en destino
 - ▶ Si el carácter es **0xFF**, guarda sin convertir y termina

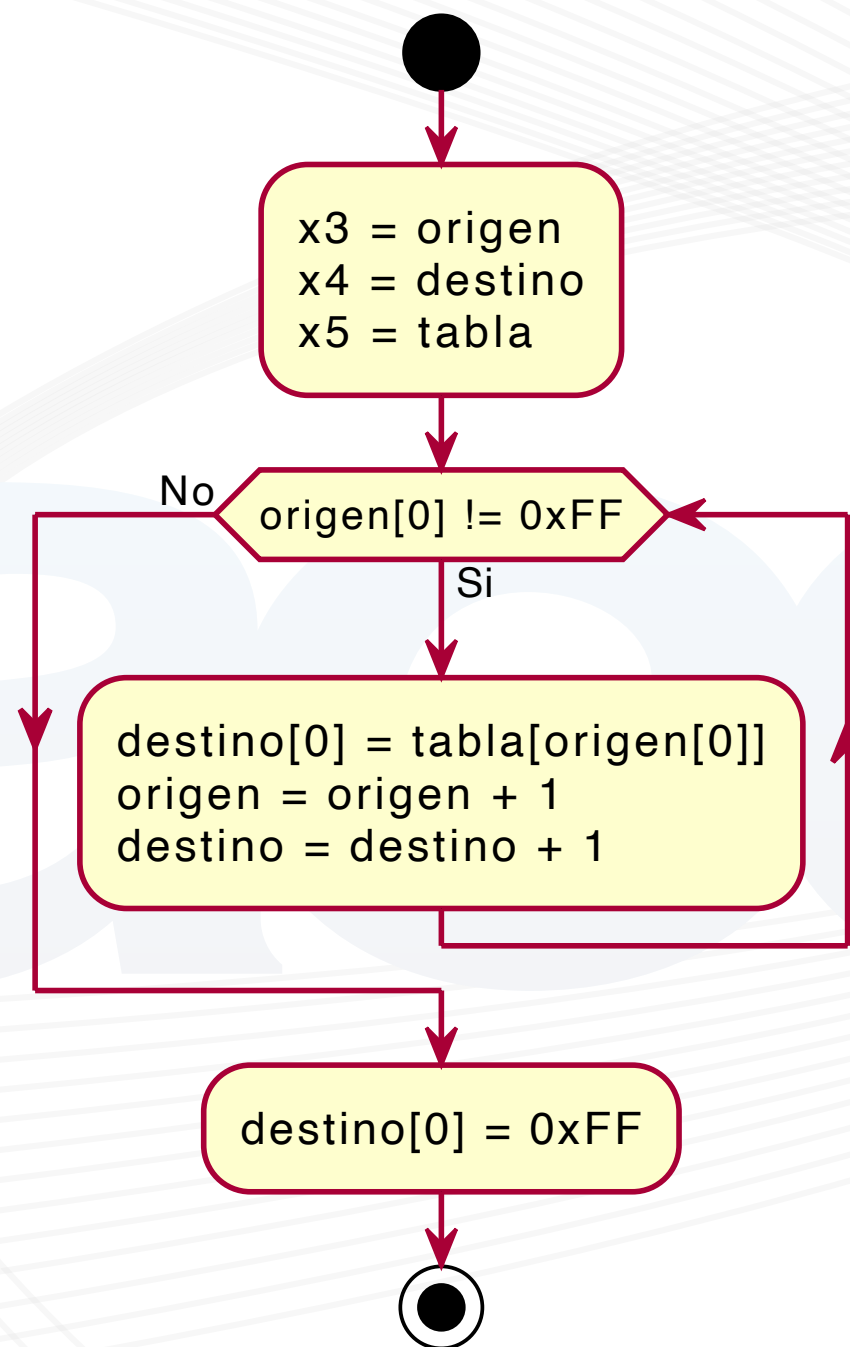
Tabla de conversión



b7	b6	b5	b4	b3	b2	b1	b0
a	b	c	d	e	f	g	nc

Digito	a	b	c	d	e	f	g	x	Valor
0	1	1	1	1	1	1	0	0	0xFC
1	0	1	1	0	0	0	0	0	0x60
2	1	1	0	1	1	0	1	0	0xDA
3	1	1	1	1	0	0	1	0	0xF2
4	0	1	1	0	0	1	1	0	0x66
5	1	0	1	1	0	1	1	0	0xB6
6	1	0	1	1	1	1	1	0	0xBE
7	1	1	1	0	0	0	0	0	0xE0
8	1	1	1	1	1	1	1	0	0xFE
9	1	1	1	1	0	1	1	0	0xF6

Diagrama de flujo



Programa

```
origen:    .data                # Define la sección de variables
          .byte 6,2,8,0xFF      # Variable inicializada de 6 bytes
          .space 12            # Completo los 20 lugares de origen
destino:   .space 16           # Variable de 20 bytes en blanco

tabla:     .byte 0xFC,0x60,0xDA,0xF2,0x66
          .byte 0xB6,0xBE,0xE0,0xFE,0xF6

          .text                # Define la sección de código
          .globl main          # Define el punto de entrada del código

main:      la    x3,origen      # Apunta un registro al bloque de origen
          la    x4,destino     # Apunta un registro al bloque de destino
          la    x5,tabla       # Apunta un registro al inicio de la tabla
          li    x6,0xFF        # Define el terminador en un registro
lazo:      lbu   x7,0(x3)       # Carga en un registro el elemento a convertir
          addi  x3,x3,1         # Mueve el puntero origen al siguiente dato
          beq   x7,x6,final     # Termina si el dato cargado es el terminador
          add   x7,x5,x7        # Calcula la dirección de la conversión en la tabla
          lbu   x7,0(x7)       # Carga la conversión desde la tabla
          sb    x7,0(x4)       # Guarda el dato convertido en destino
          addi  x4,x4,1         # Mueve el puntero destino al siguiente lugar
          j     lazo           # Repite la conversión para el siguiente dato
final:     sb    x7,0(x4)      # Guarda el ultimo dato sin convertir

          li    a0,10
          ecall                # Termina la ejecución del programa
```