

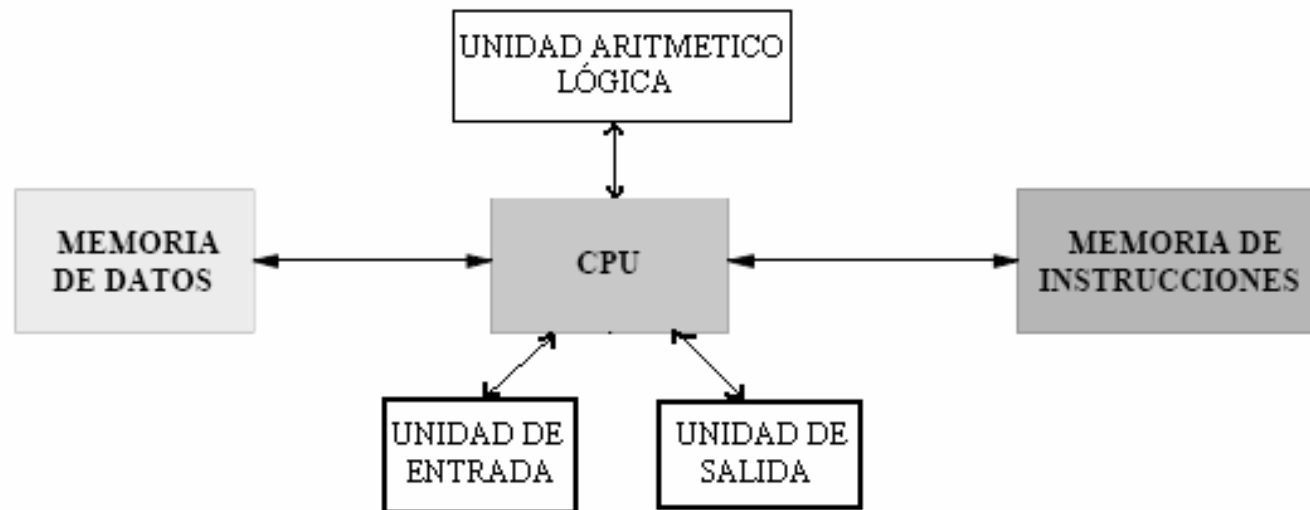
66.70 Estructura del Computador

Arquitectura del Set de Instrucciones

Instruction Set Architecture (ISA)

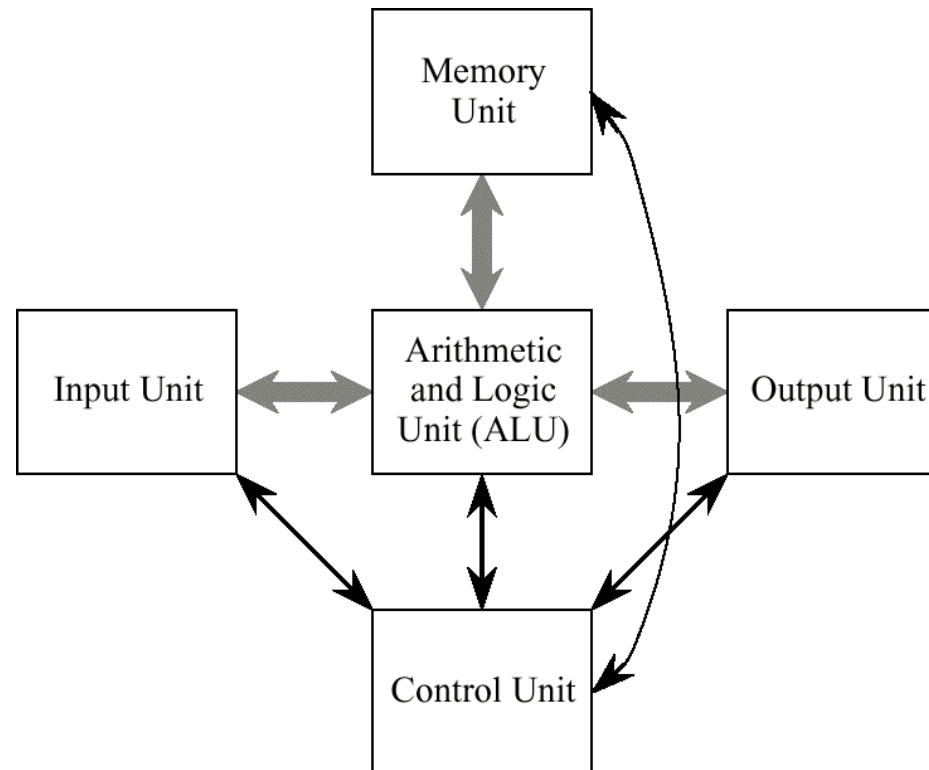
Arquitectura Harvard

- ❖ Responde a la arquitectura de las primeras computadoras
- ❖ Se aplica actualmente en sistemas pequeños (p.e.: electrodomésticos)



Arquitectura von Neumann

- ✓ Unifica el almacenamiento físico de datos y programa



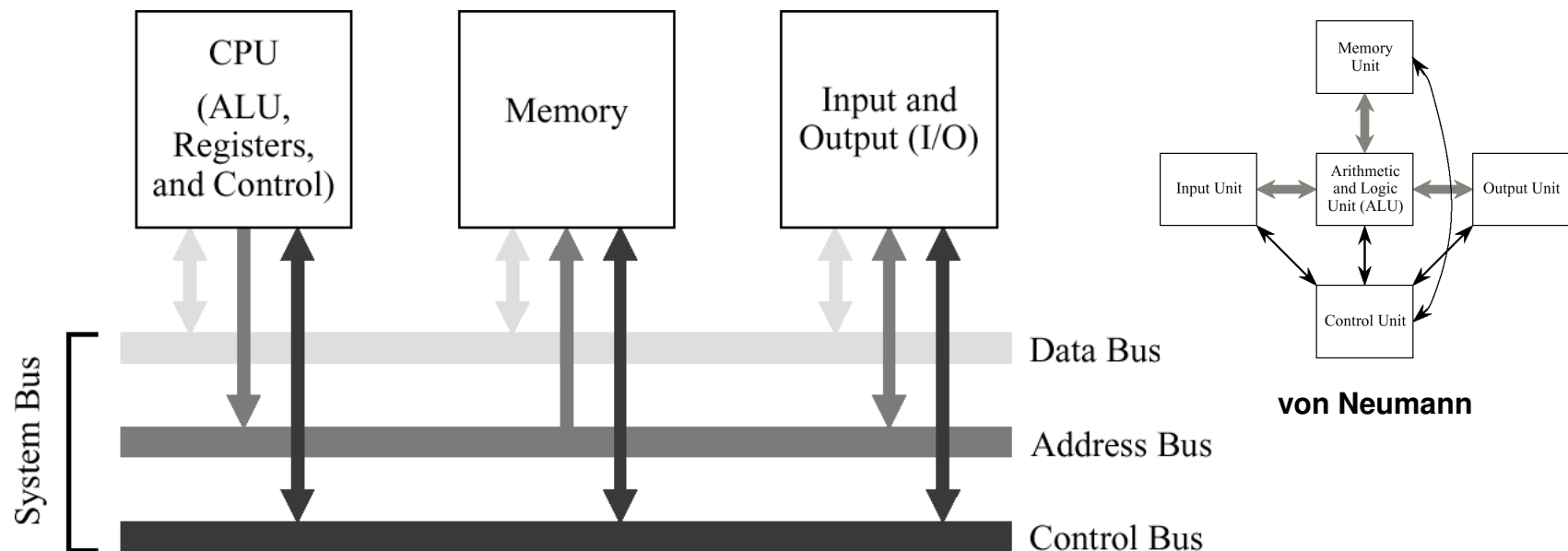
Arquitecturas von Neumann vs. Harvard

- ❖ Con la “arquitectura von Neumann” se logra modificar programas con la misma facilidad que modificar datos, permitiendo:
 - ✓ Fácil reprogramación, versatilidad característica de las computadoras actuales
 - => Compiladores, sistemas operativos, ...

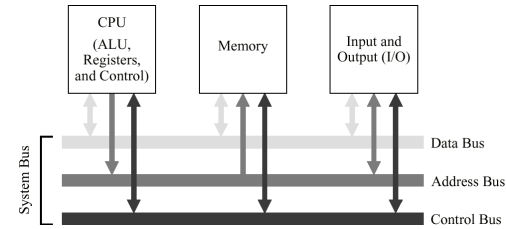
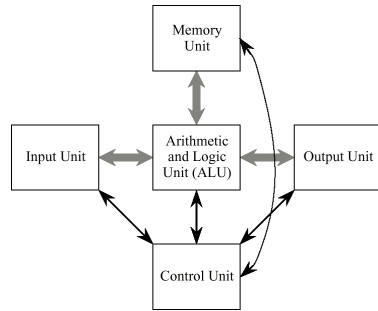
- ❖ Alternativamente la “arquitectura Harvard” (datos y programa almacenados en ubicaciones físicas diferentes) es aplicada ventajosamente en sistemas electrónicos pequeños
 - ✓ Dispositivos automáticos que cumplen una única función
 - ✓ Pequeño volumen de memoria de datos y de programa

Modelo bus de sistema

- ❖ La “arquitectura von Neumann” caracteriza a las computadoras actuales, pero se le ha incorporado un refinamiento: el **bus de sistema**
- ❖ Su propósito es reducir el número de interconexiones entre el CPU y sus subsistemas



Bus: {
▪ conjunto de cables individuales (1 bit/cable) agrupados por función
▪ por lo general bidireccionales



<h2>Modelo von Neumann</h2>	<h2>Modelo Bus de sistema</h2>
<ul style="list-style-type: none"> • Entradas y salidas separadas • Control y ALU separadas • <i>Intercomunicación:</i> Caminos separados 	<ul style="list-style-type: none"> • Unidad de entrada/salida • CPU= Control + ALU • <i>Intercomunicación:</i> Camino compartido = bus de sistema

El problema de la compatibilidad del software


Problema:

- Nueva máquina => desarrollar nuevo software

Solución:

- Compatibilidad “hacia arriba”
- Niveles de abstracción

Niveles de abstracción

- 
- *Usuario: no necesita saber programación*
 - *Lenguaje de alto nivel: datos e instrucciones del lenguaje, no importa cómo y dónde son manejados (el compilador se encarga)*
 - *Código de máquina (Assembler)*
 - *Diseño de lógica cableada o bien microprogramada*
 - *Unidades funcionales (registros, ALU ...)*
 - *Compuertas lógicas*
 - *Transistores, diodos, cables, etc.*

❖ **Esto permite generar productos con compatibilidad “hacia arriba”**

✓ **La línea IBM 360 fue la primera en garantizar compatibilidad hacia arriba en su código binario**

Niveles de abstracción

❖ Punto de vista del programador de computadoras

Arquitectura del set instrucciones (ISA)

- ✓ Conjunto de instrucciones
- ✓ Hardware accesible al programador
- ✓ El concepto ISA apareció con la familia *IBM-360*
- ✓ **No** tiene en cuenta ciclos de reloj, detalles de implementación del hardware, etc

❖ Punto de vista del diseñador del sistema

- ✓ Visión del conjunto (todos los niveles de abstracción)
- ✓ Optimización por velocidad, costos, cap. almacenamiento ...
- ✓ Soluciones de compromiso entre los distintos niveles de abstr.

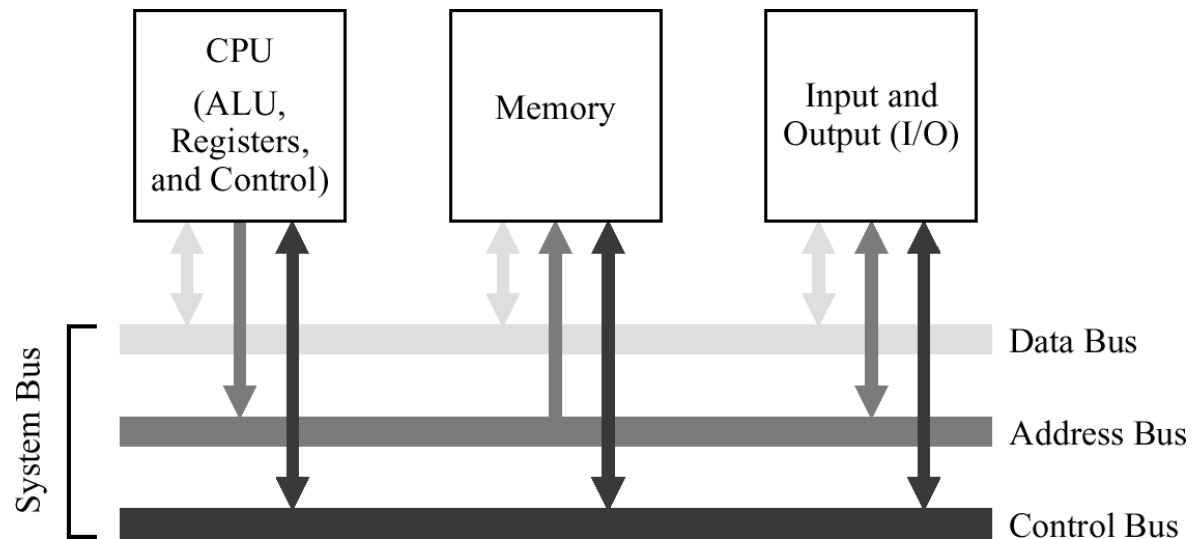
Programación a nivel código de máquina

- *Lenguajes de alto nivel son independientes del hardware*
- *Un compilador lo convierte en lenguaje de máquina*
- *Lenguaje Assembler es propio de cada procesador*
- *Lenguaje Assembler está compuesto por símbolos significativos para un ser humano (Add, jmp, mov)*
- *Un programa ensamblador convierte el lenguaje Assembler en código de máquina (serie de 0's y 1's)*

"Add r0, r1, r2" -> 0110101110101101

Ejecutar un programa bajo el modelo del bus de sistema

- *Programa (código máquina) es traído desde disco rígido a memoria*
- *CPU lee instrucciones y datos desde memoria*
- *Ejecuta cada instrucción en la secuencia programada y copiando resultados a memoria o E/S*



Memoria

Tamaños comunes para tipos de datos

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011
	11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011
	11001110 11101110 01111000 00110101
	00001011 10100110 11110010 11100110
	10100100 01000100 10100101 01010001

Memoria

- ✓ Forma una estructura de datos organizada tipo tabla
- ✓ Cada renglón de la tabla es identificado por su “dirección”
- ✓ Cada dato es agrupado físicamente de a 8 bits (=1 byte)
- ✓ Palabras de mas de 8 bits son guardadas como serie de bytes
- ✓ Los procesadores en general tienen instrucciones para acceder directamente a palabras de 1 byte o más

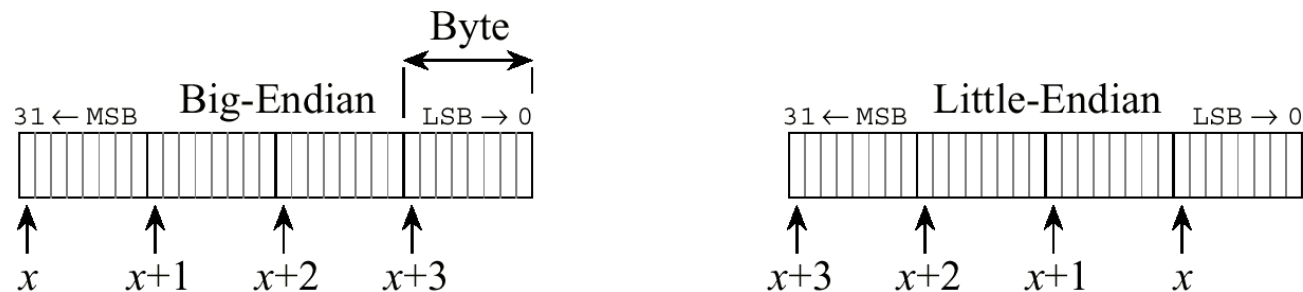
Formatos

Little-Endian y Big-Endian

- La memoria es direccionable por bytes

- Palabras multi-byte
 - El byte menos significativo en la dirección más baja
Little-Endian
 - El byte menos significativo en la dirección más alta
Big-Endian

- La dirección de la palabra multibyte es la dirección más baja

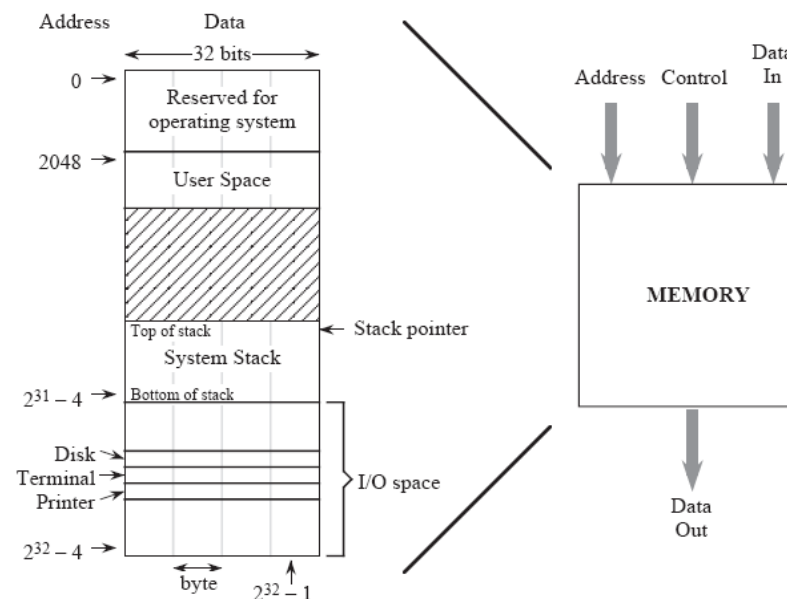


x : dirección de memoria de la palabra de 4 bytes

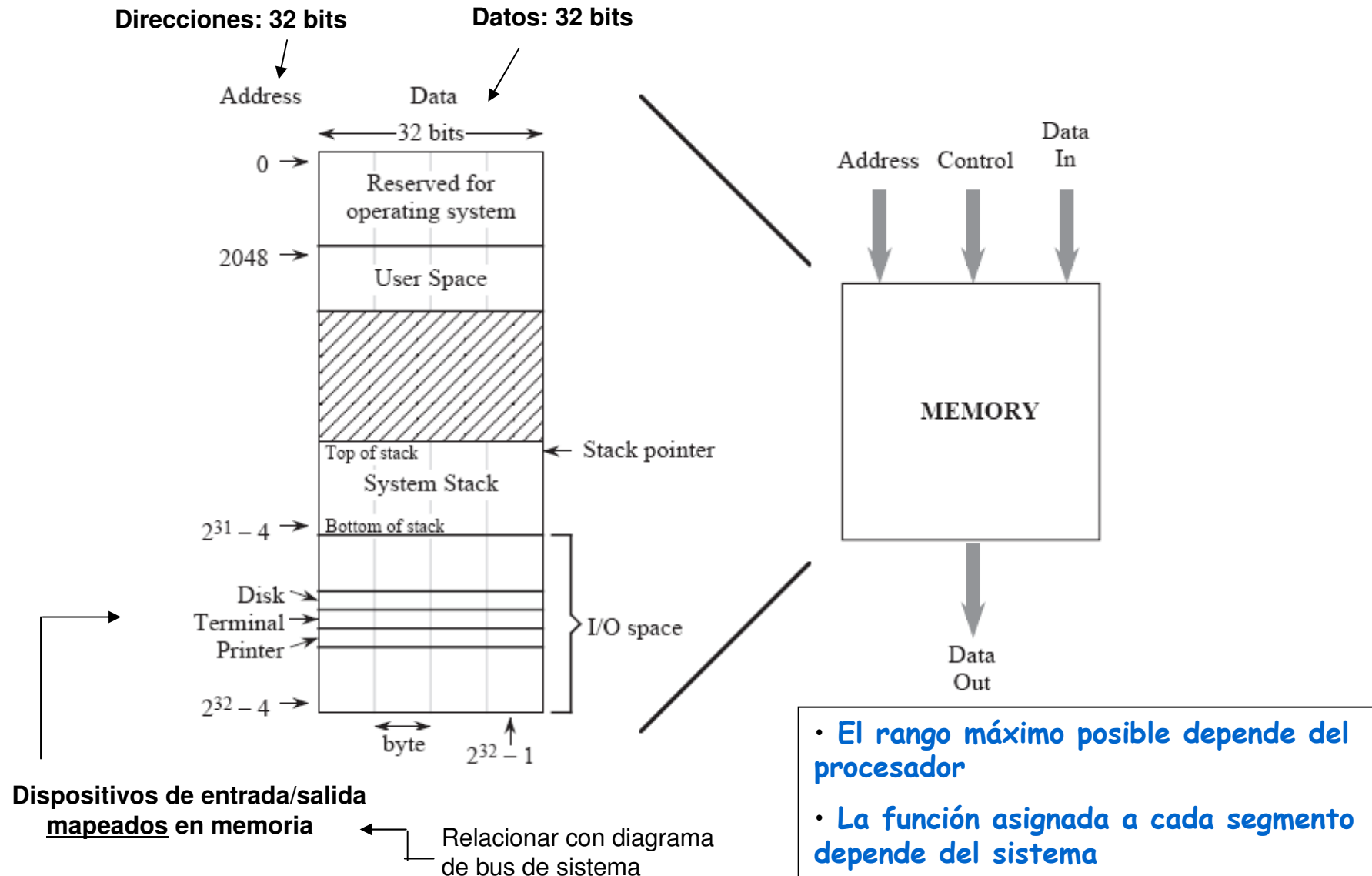
- Una u otra opción depende del procesador elegido para armar el sistema
 - (Motorola vs. Intel)

Mapa de Memoria

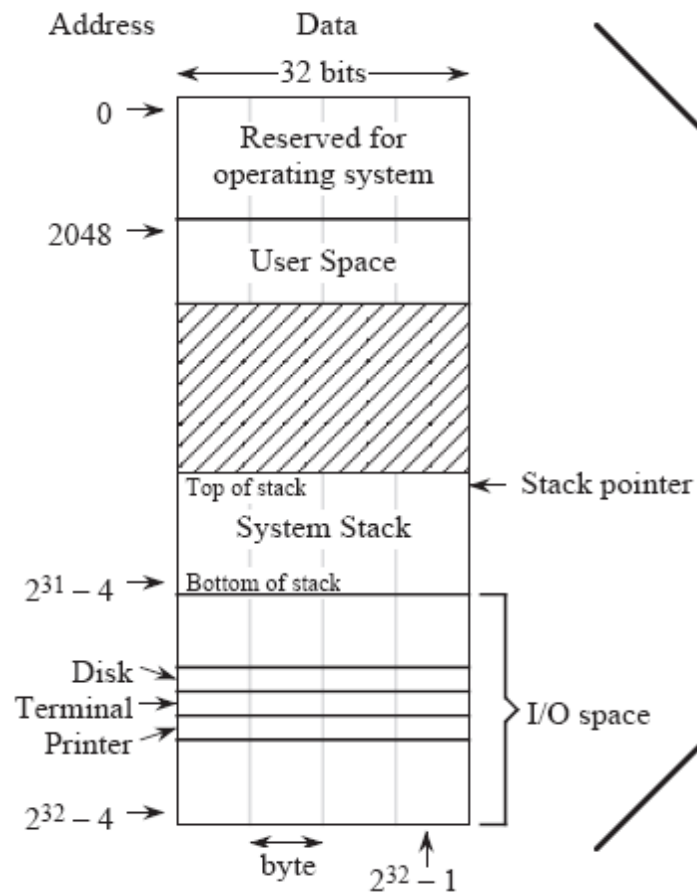
- . Grafica la forma en que se organiza el espacio de memoria
- . El espacio de memoria disponible depende del procesador
- . Una sistema (computadora) tiene un mapa de memoria específico
- . Dos sistemas basadas en el mismo procesador no tienen necesariamente el mismo mapa de memoria



Mapa de Memoria



Mapa de Memoria



Cómo se vería el mapa de memoria en caso de tener 2 Gb instalados?

Qué puede hacer una
computadora?

Qué puede hacer una computadora?

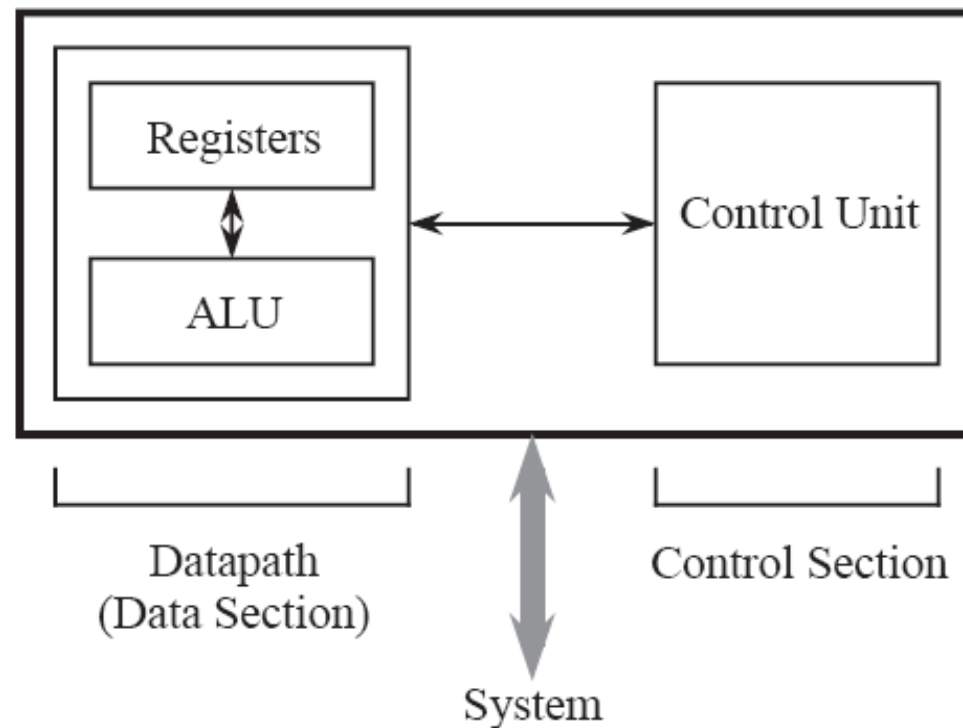
- Las tareas que implementa pueden ser muy complejas
- Los modos de operación internos son sorprendentemente limitados

Esencialmente, lo que puede hacer es:

- Ejecutar operaciones en una secuencia ordenada
- Existen dos tipos básicos de operaciones:
 - *Mover datos*
 - *Operaciones aritméticas simples y oper. lógicas*

Unidad Central de Proceso CPU

- ❖ Sección de control
- ❖ Sección de datos (*“camino de datos”* o *“datapath”*)



Sección de control

- *Función:* Controlar la ejecución de las instrucciones de programa que están almacenadas en memoria principal
- *Principales registros:*
 - ✦ **PC** (Contador de programa)
 - ✦ **IR** (Registro de instrucciones)
- *Ejecutar una instrucción de progr. significa:*
 - 1) Buscar en memoria la próxima instrucción a ser ejecutada
 - 2) Decodificar el código de operación de esa instrucción
 - 3) Traer operandos desde memoria principal, si los hubiera
 - 4) Ejecutar la instrucción y guardar los resultados
 - 5) Volver a (1)

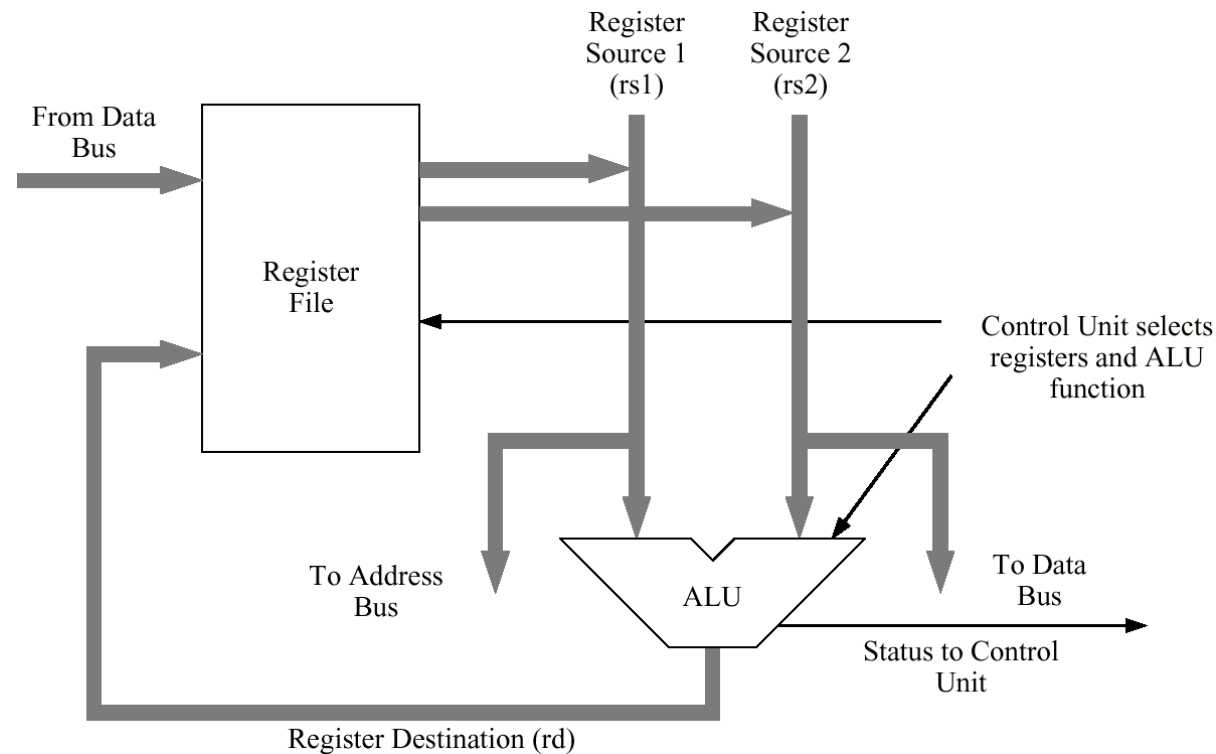


(“Ciclo de búsqueda-ejecución”, “ciclo de fetch”, “ciclo de “fetch-execute”)

Sección de datos

➤ *Función:* Realizar la tarea que involucra cada instrucción. Actúa bajo el control de la sección control. Para ello cuenta con:

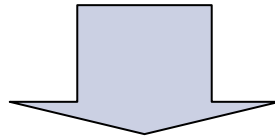
- ❖ Archivo de registros
- ❖ Unidad aritmético-lógica (ALU)



Sección de datos

Archivo de registros

- ❖ Es memoria interna al CPU organizada como RAM
- ❖ Bus de direcciones (interno y de pocos bits)
- ❖ Cada registro está implementado con circuitos muy rápidos



Programas con uso intensivo de registros son más rápidos
que los que hacen uso intensivo de memoria principal

Sección de datos

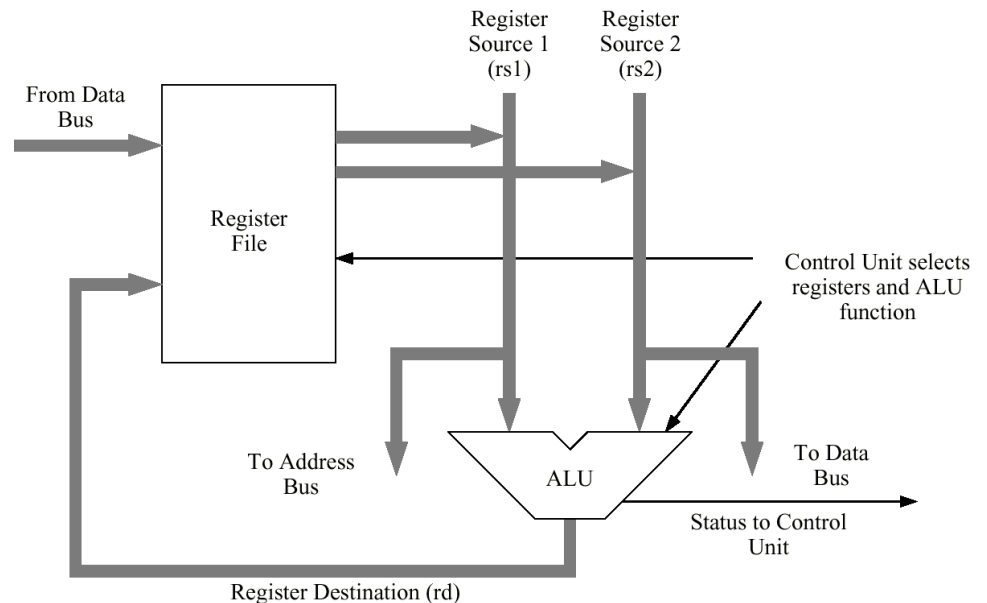
Buses

Internos a la CPU

- ❖ Entrada ALU: 2 Buses de Datos (rs1 y rs2)
- ❖ Resultado de ALU hacia registros: 1 Bus de datos (rd)

Conectados con el bus de sistema

- ❖ Desde DATA BUS al archivo de registros
- ❖ Hacia DATA BUS (rs2)
- ❖ Hacia ADDRESS BUS (rs1)



Set de instrucciones

➤ Características típicas

- ❖ Tamaño de las instrucciones (=espacio ocupado por el código de máquina)
- ❖ Tipo de operaciones admitidas
- ❖ Tipo de operandos (ubicación y tamaño)
- ❖ Tipo de resultados (ubicación y tamaño)

➤ Compatibilidad del software

- ❖ Programa compilado para PC no anda en PowerPC (Mac / IBM)
- ❖ Lenguajes de alto nivel son idénticos pero requieren recompilación
- ❖ Programa compilado para Mac no anda en IBM pSeries ¿?

Un ejemplo de ISA

Arquitectura SPARC

(Scalable Processor ARChitecture)

- ✓ Big-endian
 - ✓ Arquitectura RISC
-
- ✓ Planteada en la Universidad de Berkeley 1980/82
 - ✓ Primera implementación Sun Microsystems 1987
 - ✓ Especificaciones de uso público
 - ✓ Algunos implementadores: Sun Microsystems, Texas Instruments, Toshiba, Fujitsu, Cypress, Tatung, entre otros

Arquitectura ARC

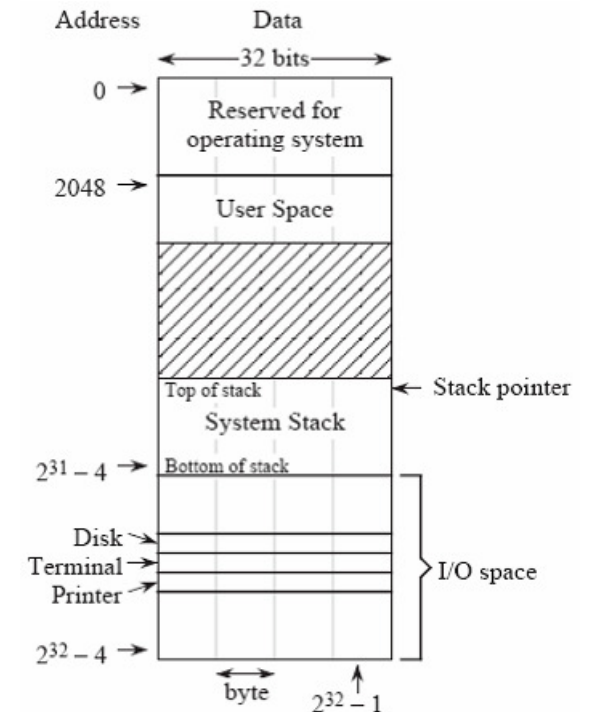
(A Risc Computer)

➤ Memoria

- ❖ Datos de 32 bits direccionables por bytes
Dirección del dato: Byte menos significativo
- ❖ Espacio de direcciones: 2^{32}
- ❖ Big-endian

➤ Set de instrucciones

- ❖ Es un subconjunto de ISA SPARC
- ❖ Todas las instrucciones ocupan 32 bits
- ❖ 32 registros de 32 bits
- ❖ **P**rogram **S**tatus **R**egister (PSR) guarda los flags de ALU
- ❖ Sólo dos instrucciones acceden a memoria principal
(1) leer memoria a registro (2) escribir desde registro a memoria.



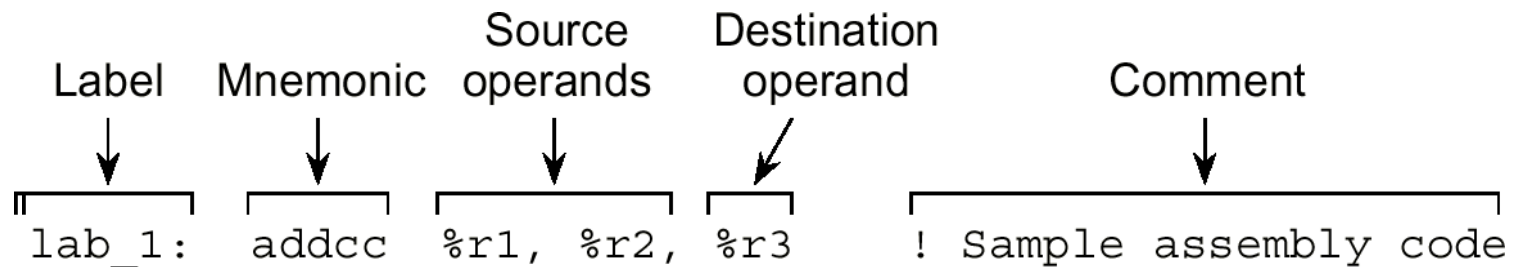
Arquitectura ARC

Instrucciones

	Mnemonic	Meaning
Memory	ld	Load a register from memory
	st	Store a register into memory
Logic	sethi	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND
	orcc	Bitwise logical OR
	orncc	Bitwise logical NOR
Arithmetic	srl	Shift right (logical)
	addcc	Add
	call	Call subroutine
Control	jmp1	Jump and link (return from subroutine call)
	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

Lenguaje assembly ARC

SINTAXIS



- No importa el encolumnado, sino el orden de izquierda a derecha
- Distingue mayúsculas de minúsculas
- Números: Default -> Base 10

Hexadecimal -> Si empieza con "0x" o finaliza con "h"

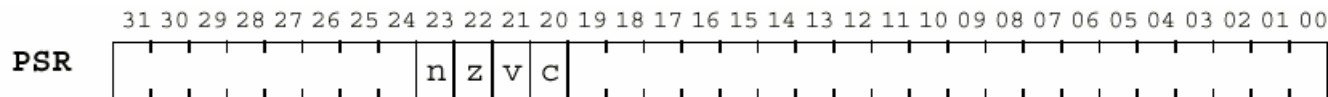
Lenguaje assembly ARC

REGISTROS ACCESIBLES AL PROGRAMADOR

Register 00	%r0 [= 0]	Register 11	%r11	Register 22	%r22
Register 01	%r1	Register 12	%r12	Register 23	%r23
Register 02	%r2	Register 13	%r13	Register 24	%r24
Register 03	%r3	Register 14	%r14 [%sp]	Register 25	%r25
Register 04	%r4	Register 15	%r15 [link]	Register 26	%r26
Register 05	%r5	Register 16	%r16	Register 27	%r27
Register 06	%r6	Register 17	%r17	Register 28	%r28
Register 07	%r7	Register 18	%r18	Register 29	%r29
Register 08	%r8	Register 19	%r19	Register 30	%r30
Register 09	%r9	Register 20	%r20	Register 31	%r31
Register 10	%r10	Register 21	%r21		

PSR	%psr	PC	%pc
← 32 bits →		← 32 bits →	

- Registros de uso predefinido: %r14 stack pointer , %r15 direcc.retorno de procedimiento
- %r0 *siempre en cero*



Lenguaje assembly ARC

Ejemplos de instrucciones

ld [FFFFFF1C9], %r3

add %r3, %r1, %r5

Etiqueta1: sub %r5, %r1, %r3

srl %r20, 2, %r2

subcc %r3, 4, %r5

bneg Etiqueta1

sra %r20, 2, %r2

be Etiqueta1

st %r1, [000001D3]

ba Etiqueta3

Pseudo-instrucciones o directivas al ensamblador

- Indican al ensamblador como procesar una sección de programa
- Las instrucciones son específicas de un procesador
Las pseudo-instrucciones son específicas de un programa ensamblador
- Algunas generan información en la memoria, otras no

Pseudo-Op	Usage	Meaning
.equ	X .equ #10	Treat symbol X as (10) ₁₆
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global Y	Y is used in another module
.extern	.extern Z	Z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition
.if	.if <cond>	Assemble if <cond> is true
.endif	.endif	End of .if construct

Programa ARC que suma dos números en memoria

```
! This programs adds two numbers

        .begin
        .org 2048
prog1:   ld      [x], %r1          ! Load x into %r1
        ld      [y], %r2          ! Load y into %r2
        addcc   %r1, %r2, %r3     ! %r3 ← %r1 + %r2
        st      %r3, [z]          ! Store %r3 into z
        jmp1    %r15 + 4, %r0     ! Return

x:       15
y:       9
z:       0
        .end
```

Programa ARC que suma los elementos de un array

```
! This program sums LENGTH numbers
! Register usage:      %r1 - Length of array a
!                      %r2 - Starting address of array a
!                      %r3 - The partial sum
!                      %r4 - Pointer into array a
!                      %r5 - Holds an element of a

        .begin                ! Start assembling
        .org 2048              ! Start program at 2048
a_start .equ 3000              ! Address of array a
        ld [length], %r1 ! %r1 ← length of array a
        ld [address], %r2 ! %r2 ← address of a
        andcc %r3, %r0, %r3 ! %r3 ← 0
loop:    andcc %r1, %r1, %r0 ! Test # remaining elements
        be done            ! Finished when length=0
        addcc %r1, -4, %r1 ! Decrement array length
        addcc %r1, %r2, %r4 ! Address of next element
        ld %r4, %r5        ! %r5 ← Memory[%r4]
        addcc %r3, %r5, %r3 ! Sum new element into r3
        ba loop            ! Repeat loop.

done:    jmp1 %r15 + 4, %r0 ! Return to calling routine

length: 20                    ! 5 numbers (20 bytes) in a
address: a_start
        .org a_start        ! Start of array a
a:       25                    ! length/4 values follow
        -10
        33
        -5
        7

        .end                ! Stop assembling
```

CÓDIGO DE MÁQUINA

Formato de instrucciones

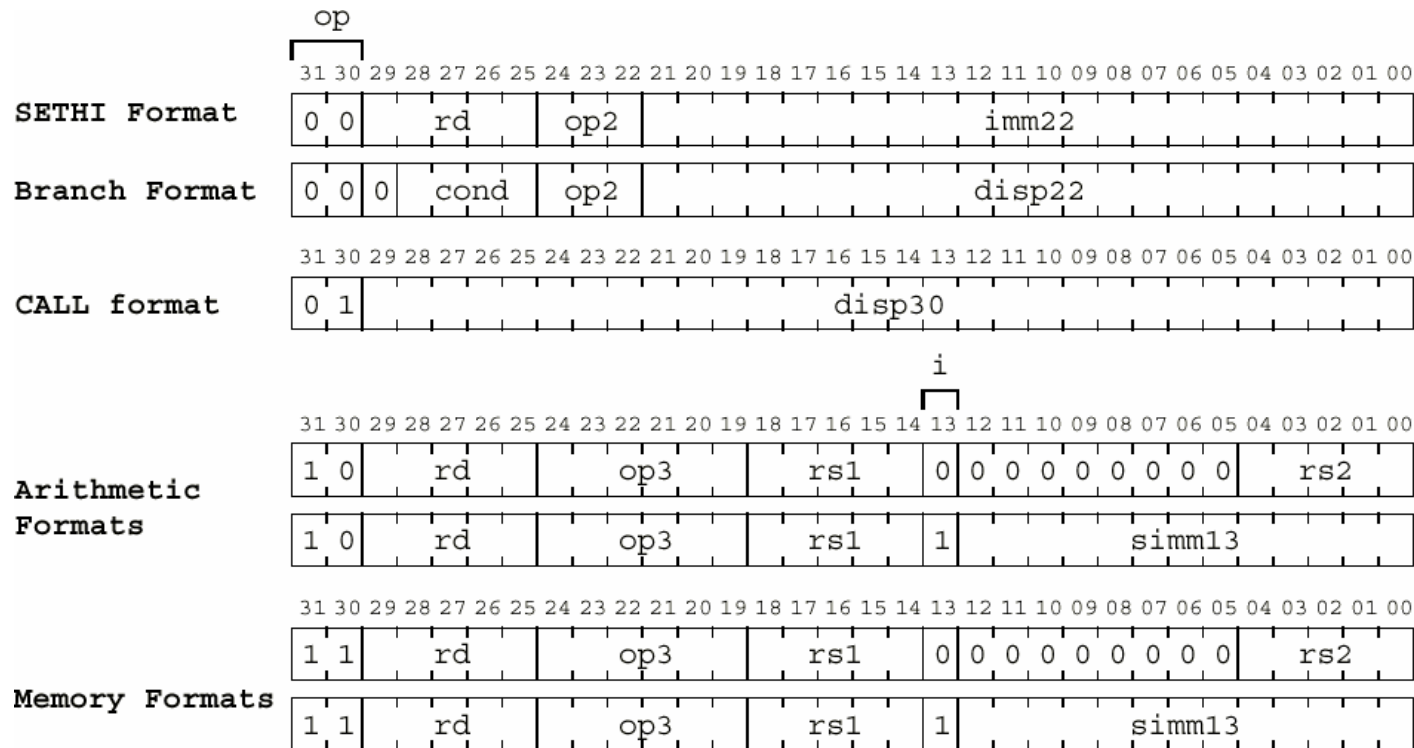
- ❖ Todas las instrucciones ocupan 32 bits
- ❖ No todas las instrucciones siguen el mismo formato
- ❖ Se definen grupos de bits a los que se da un significado
- ❖ Todas las instrucciones ARC pueden agruparse en 5 formatos:
(respecto de su código de máquina)
 - Formato de la instrucción Sethi
 - Formato de instrucciones tipo Branch
 - Formato de la instrucción Call
 - Formato de instrucciones aritméticas
 - Formato de instrucciones de acceso a memoria



Estos 5 formatos **no** están relacionados con los 5 tipos de instrucción en transparencia anterior!

CÓDIGO DE MÁQUINA

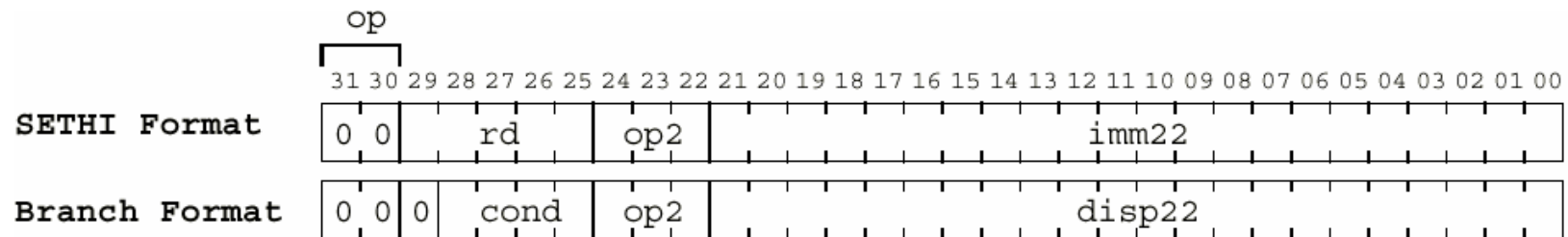
Cinco formatos de instrucción



op	Format	op2	Inst.	op3 (op=10)	op3 (op=11)	cond	branch
00	SETHI/Branch	010	branch	010000 addcc	000000 ld	0001	be
01	CALL	100	sethi	010001 andcc	000100 st	0101	bcs
10	Arithmetic			010010 orcc		0110	bneg
11	Memory			010110 orncc		0111	bvs
				100110 srl		1000	ba
				111000 jmpl			

CÓDIGO DE MÁQUINA

Formatos de instrucción



SETHI / BRANCH: op=00 op2=010 => BRANCH

bit29=0

cond=n, z, v, c

Desplaz. = constante *disp22*.

0001 be
0101 bcs
0110 bneg
0111 bvs
1000 ba

op2=100 => SETHI

rd=registro de destino

operando= const. *imm22*

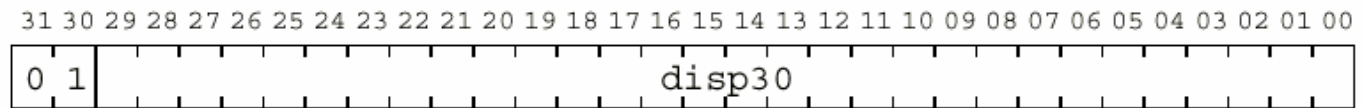
Medido en palabras de 4 bytes

4 x *disp22* (2 shift left)

CÓDIGO DE MÁQUINA

Formatos de instrucción

CALL format

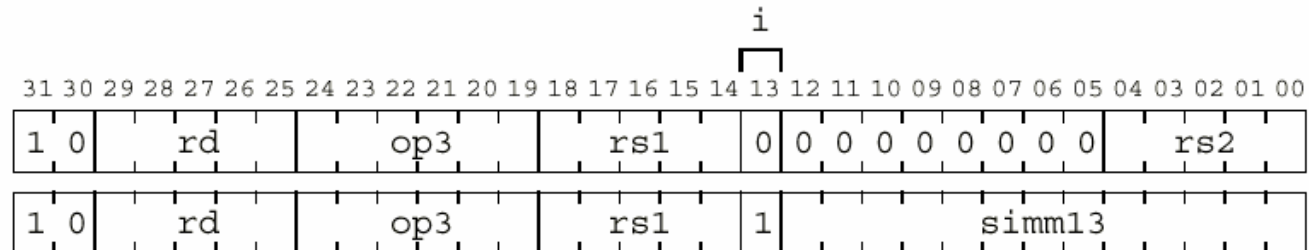


CALL op=01 desplazamiento → constante *disp30*

CÓDIGO DE MÁQUINA

Formatos de instrucción

Arithmetic
Formats



ARITMETICA op=10

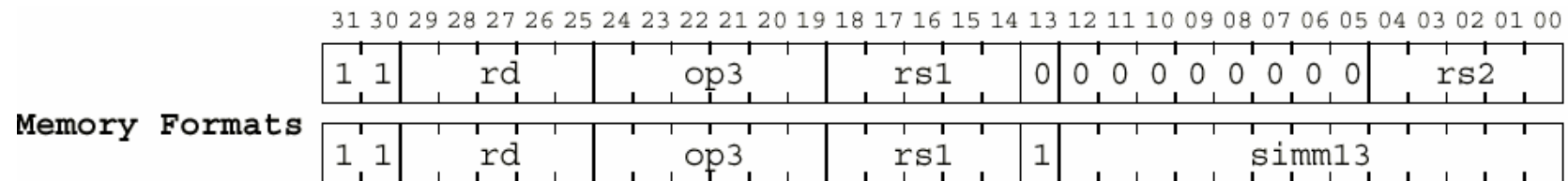
op3
 010000 addcc
 010001 andcc
 010010 orcc
 010110 orncc
 100110 srl
 111000 jmp1

1er. registro origen = rs1
 2do reg. origen = rs2 **si i=0**
 2do reg. origen = constante *simm13* **si i=1**
 Reg. destino = rd

simm13 se interpreta extendiendo el signo a 32 bits

CÓDIGO DE MÁQUINA

Formatos de instrucción



ACC. MEMORIA: $op=11$

$op3$
 000000 ld rd=reg.destino
 000100 st rd=reg.origen

direcc. de memoria (ld o st)

= $rs1 + rs2$

= $rs1 + \text{simm13 (constante)}$

si i = 0

si i = 1

RISC vs CISC

(Introducción)

- ARC accede a memoria sólo con *ld* y *st* (RISC), otras instrucciones son registro-registro
- Otros procesadores tienen extensos set de instrucciones que incluyen operaciones con varios modos de acceder operadores en memoria (CISC)
- Procesador RISC tiene más registros
- Procesador RISC formado por hardware más sencillo => más rápido
=> menor tamaño
- Programas RISC tienen más instrucciones
- Programas RISC requiere más memoria para almacenarlo
- Compiladores RISC son más complejos

MODOS DE DIRECCIONAMIENTO

Ejemplos

Modo	Sintaxis	
Inmediato	$\#K$	Constante incluida en la instrucción
Por registro	Rn	El registro tiene el dato
Directo o absoluto	M	Dirección de memoria incluida en la instrucción
Indirecto	(M)	Direcc de memoria donde esta el puntero al dato (poco usado, lento)
Indirecto x Registro	(Rn)	El registro tiene el puntero al dato
Indexado x Registro	$(Rm+Rn)$	Un registro da la dirección inicial el otro un incremento (arrays)
Base + Desplaz.	$(Rb+Rn)$	Desplazamiento a partir de una direcc base (reg base). Cada programa tiene su base
Base+Desplaz. Indexado	$(Rb+Rn+Rn)$	Reg. base es del programa, Desplaz indica inicio de array, Reg indice indica el elemento del array

→ • Cuáles están en el ISA ARC?

Para cada uno { • Datos al tiempo de compilación?
• Datos al tiempo de ejecución?

LLAMADO A SUBROUTINAS

Parámetros por registros

```
! Calling routine
```

```
⋮
```

```
ld    [x], %r1
```

```
ld    [y], %r2
```

```
call  add_1
```

```
st    %r3, [z]
```

```
⋮
```

```
x: 53
```

```
y: 10
```

```
z: 0
```

Ventajas

- Simple
- Rápido

Desventajas

- Muchos argumentos => muchos registros
- Muchas rutinas anidadas => muchos registros

```
! Called routine
```

```
! %r3 ← %r1 + %r2
```

```
add_1: addcc  %r1, %r2, %r3
```

```
        jmp1  %r15 + 4, %r0
```

LLAMADO A SUBROUTINAS

Parámetros por área reservada en memoria

Copia operandos al área

Pasa puntero al área

Forma alternativa:

```
ld [addr_de_x], %r15
```

- Más simple
- Más lento

! Calling routine	! Called routine
:	! x[2] ← x[0] + x[1]
:	
st %r1, [x]	add_2: ld %r5, %r8
st %r2, [x+4]	ld %r5 + 4, %r9
sethi x, %r5	addcc %r8, %r9, %r10
srl %r5, 10, %r5	st %r10, %r5 + 8
call add_2	jmp1 %r15 + 4, %r0
ld [x+8], %r3	
:	
! Data link area	
x: .dwb 3	

Ventajas

- Tamaño limitado a memoria disponible
- Sólo se pasa un registro puntero en el llamado

Desventajas

- Rutinas recursivas
- El tamaño del área reservada debe conocerse en tiempo de ensamblado

LLAMADO A SUBROUTINAS

Parámetros en el stack

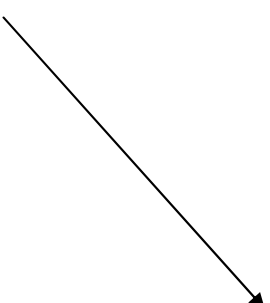
```
! Calling routine
:
%sp .equ %r14
addcc %sp, -4, %sp
st    %r1, %sp
addcc %sp, -4, %sp
st    %r2, %sp
call  add_3
ld    %sp, %r3
addcc %sp, 4, %sp
:
```

Ventajas

- Tamaño limitado a memoria disponible
- La ocupación de memoria es dinámica
- Funciona bien con rutinas anidadas o recursivas

Desventajas

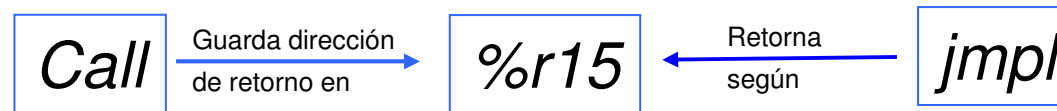
- Es lento:
 - Acceso a memoria para pasar parámetros
 - Acceso a memoria para leer parámetros



```
! Called routine
! Arguments are on stack.
! %sp[0] ← %sp[0] + %sp[4]
%sp .equ %r14
add_3: ld    %sp, %r8
      addcc %sp, 4, %sp
      ld    %sp, %r9
      addcc %r8, %r9, %r10
      st    %r10, %sp
      jmp1  %r15 + 4, %r0
```

LLAMADO A SUBROUTINAS

Dirección de retorno



Problema:

Rutinas anidadas sobrescriben %r15



Con cada llamado es necesario guardar el valor de %r15



Donde se guarda?: depende de la convención usada al pasar parámetros

Tipo de convención para los parámetros	<i>%r15</i> se guarda en
Por registros	Registro no utilizado
Por área reservada en memoria	Incluido en área reservada
En la pila	Pila

