

66.70 Estructura del Computador

Arquitectura del Set de Instrucciones

Desde el código de alto nivel al código de máquina



Independiente
del hardware

símbolos significativos para humanos
(Add, jmp, mov, etc.)

Secuencia de 0s y 1s

Add r0, r1, r2 =>

0110101110101101

Desde el código de alto nivel al código de máquina

```
#include "stdio.h"
int main(void)
{
    char operando = 26;
    char resultado;

    resultado = operando + 50;
    printf("Resultado de la suma: %i\n", resultado);
    return(0);
}
```

Este programa se ejecutará en hardware...

¿Dónde se guardan físicamente las variables?

¿Cómo accedo a los periféricos?

¿Dónde se guarda físicamente el programa?



En Assembler necesito conocer
cómo está organizada la memoria

Accesos a memoria RAM

- ✓ 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)
- ✓ Los procesadores en general tienen instrucciones para acceder simultáneamente 1, 2, 4 o más bytes (palabras)
- ✓ Palabras de mas de 8 bits son guardadas como una serie de bytes

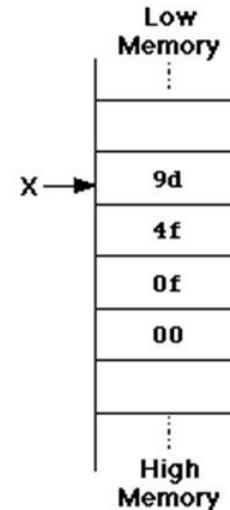
Guardar palabras de varios bytes

- Necesito guardar la variable “x” en RAM
- La memoria es direccionable por bytes

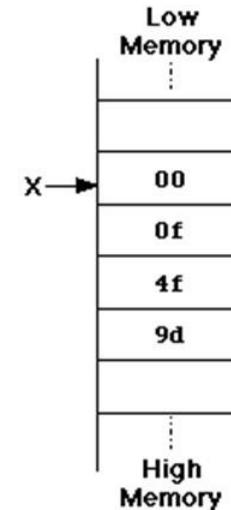
$x=000F4F9Dh$



Little-Endian



Big-Endian



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

El byte menos significativo en la dirección más baja
Little-Endian

- Orden de los 4 bytes

El byte menos significativo en la dirección más alta
Big-Endian

- Que sea uno u otro caso depende del procesador

Intel vs. Motorola

Asignación del espacio direccional

¿Qué significa espacio direccional?

¿Qué lo determina?

¿Dónde se guardan los programas?

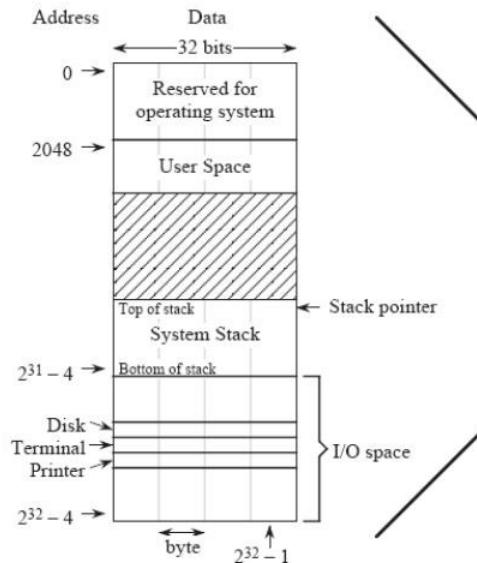
¿Dónde se guardan las variables de un programa ?

¿Dónde se guarda el stack?

¿Dónde se guarda el sistema operativo?

Mapa de memoria de un sistema

- Representa la asignación dada al **espacio de direcciones**
- El tamaño del espacio direccionable es específico del procesador (**¿porqué?**)
- El mapa de memoria es específico de un sistema (tipo de computadora)
- Dos sistemas basadas en el mismo procesador no tienen necesariamente el mismo mapa de memoria
- Direcciones de los dispositivos de entrada/salida



- **El rango máximo posible depende del procesador**
- **La función asignada a cada segmento depende del sistema**

Arquitectura ARC

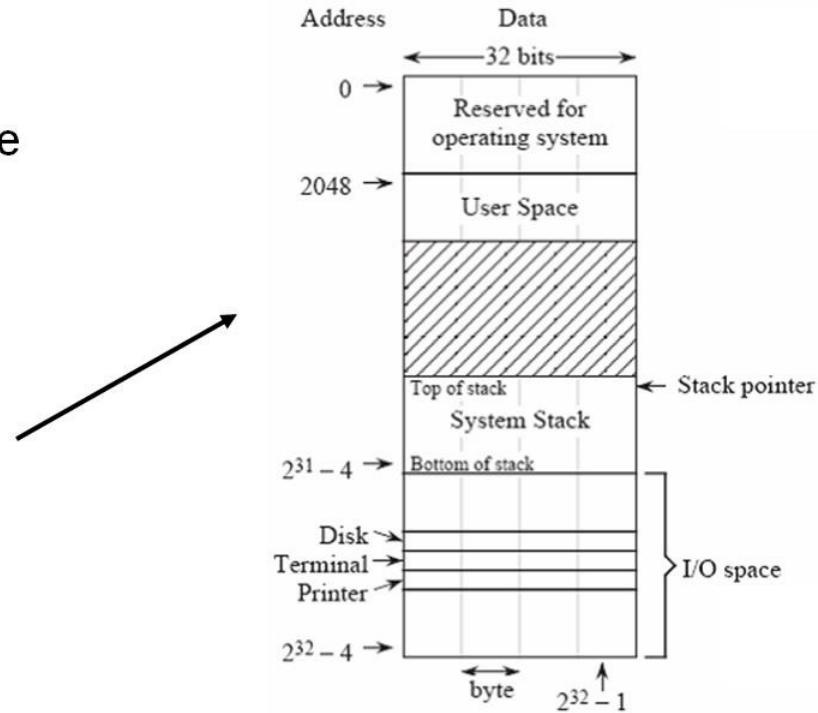
(A Risc Computer)

➤ Memoria

- ❖ Datos de 32 bits direccionables por byte
- ❖ Espacio de direcciones: 2^{32}
- ❖ Big-endian
- ❖ Mapa de memoria especificado por

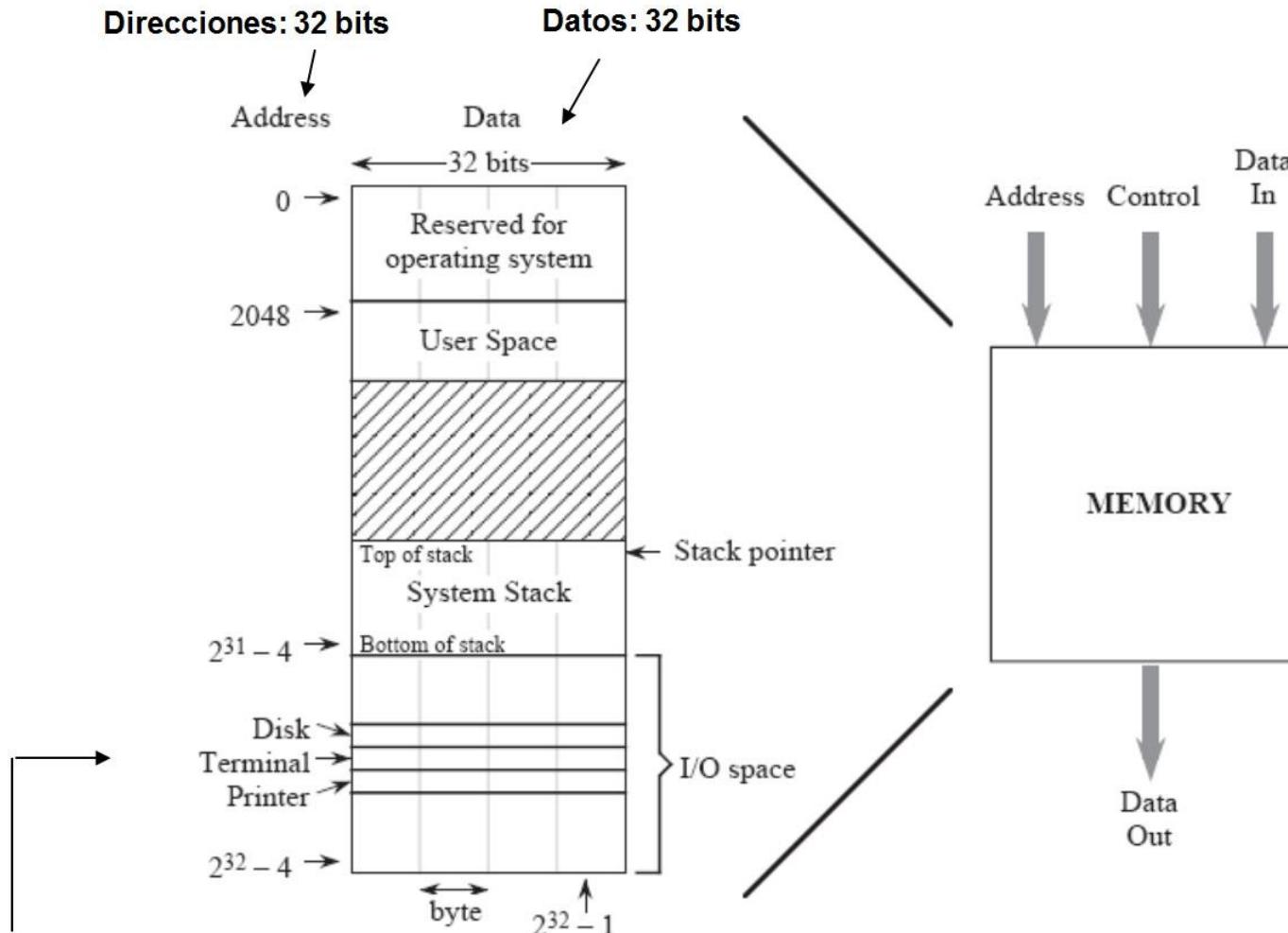
➤ Set de instrucciones

- ❖ Es un subconjunto de SPARC
- ❖ Todas las instrucciones ocupan 32 bits
- ❖ 32 registros de 32 bits
- ❖ Program Status Register (PSR) guarda los flags de ALU
- ❖ Sólo dos instrucciones acceden a memoria principal
 - (1) leer memoria a registro (2) escribir desde registro a memoria.



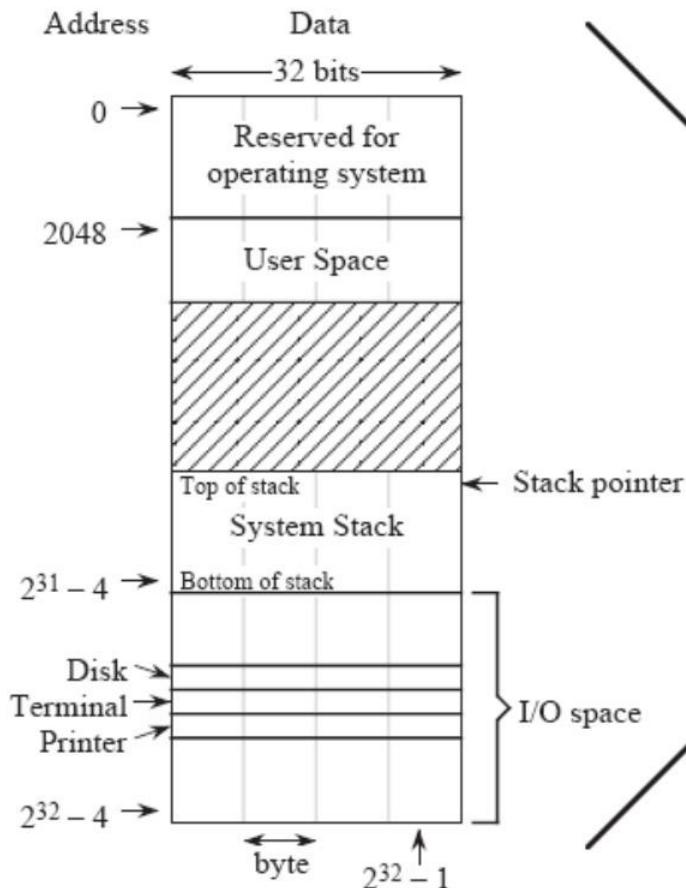
El sistema ARC

Mapa de memoria



**Dispositivos de entrada/salida
mapeados en memoria**

Mapa de memoria



Cómo se vería el mapa de memoria en caso de tener 1 Gb instalado?

y con 3 Gb?

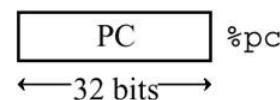
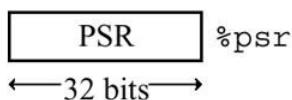
Algunas de las instrucciones ARC

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

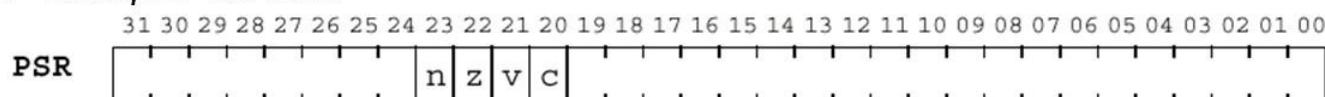
Quién define el set de instrucciones de un procesador

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		

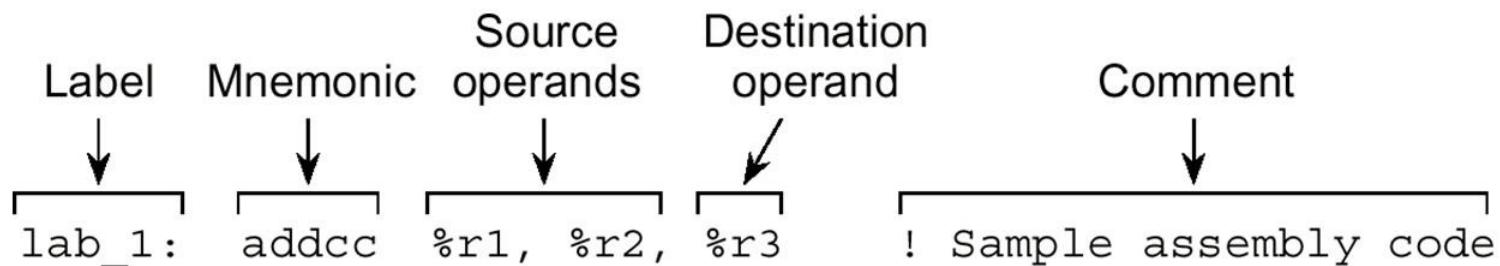


- Registros de uso predefinido: %r14 stack pointer , %r15 direcc.retorno de procedimiento
- *%r0 siempre en cero*



Quién define los registros accesibles al programador?

Sintaxis



- Distingue mayúsculas de minúsculas
- Números: default -> Base 10
hexadecimal -> Si empieza con “0x” o finaliza con “h”

¿Quién define esta sintaxis?

Directivas al ensamblador (ARCtools)

- Indican al ensamblador como procesar una sección de programa
- -Las instrucciones son específicas de un procesador
 - Las pseudo-instrucciones o directivas 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 C que suma dos números en memoria

```
int main()
{
    long x = 15;
    long y = 9;
    long z = 0;

    z = x + y;

    return(0);
}
```

Queremos escribirlo en Assembler

- ¿Dónde se guardan físicamente las variables?
- ¿Como definimos el tipo de variable?
- ¿Dónde se guarda físicamente el programa?
- ¿Qué instrucciones utilizar?

Programa ARC que suma dos números en memoria

```
int main(void)
{
    long x = 15;
    long y = 9;
    long z = 0;

    z = x + y;

    return(0);
}
```

```
.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 dos números en memoria

```
int main(void)
{
    long x = 15;
    long y = 9;
    long z = 0;

    z = x + y;

    return(0);
}
```

Assembler

```
.begin
.org 2048
prog1: ld      [x] , %r1
        ld      [y] , %r2
        addcc  %r1, %r2, %r3
        st      %r3, [z]
        jmp1   %r15 + 4, %r0
x:      15
y:      9
z:      0
.end
```

RAM

	Dir. 2048	Dir. 2052
c2002814h		
c4002818h		
86804002h		
c420281ch		
81c3e004h		
0000000fh		
5		
0		

Programa C que suma los elementos de un array

```
long arrayA[5]={25, 10, 31, -5, 7};  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ] ;  
    }  
    return(0);  
}
```

Programa que suma los elementos de un array

```
long arrayA[5]={25, 10, 31, -5, 7};  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ];  
    }  
    return(0);  
}
```

! Este programa suma LENGTH números
! Uso de registros: %r1 - Indice de arrayA
!
! %r3 - Suma parcial
!
! %r5 - Guarda un elemento de arrayA

.begin
.org 2048 ! El programa empieza en la direcc. 2048
main: andcc %r3, %r0, %r3 ! Pone a 0 el %r3
 ld [length], %r1 ! %r1 = length
for: andcc %r1, %r1, %r0 ! Chequea elementos restantes
 be done ! Si no hay mas elementos termina
 addcc %r1, -4, %r1 ! Actualiza índice al array
 ld %r1,[arrayA],%r5
 addcc %r3, %r5, %r3 ! Suma un nuevo elemento a r3
 ba for ! Vuelve al for
done: jmpl %r15 + 4, %r0 ! Vuelve al proceso invocante
length: 20 ! 5 numeros (20 bytes)
arrayA: 25 ! Contenido de ArrayA
 -10
 31
 -5
 7
.end

Versión 1

Programa que suma los elementos de un array

```
long arrayA[5] ;  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ];  
    }  
    return(0);  
}
```

Versión 2

! Este programa suma LENGTH números
! Uso de registros: %r1 - Indice de arrayA
!
! %r3 - Suma parcial
!
! %r5 - Guarda un elemento de arrayA

.begin
.org 2048 ! El programa empieza en la direcc. 2048
main: andcc %r3, %r0, %r3 ! Pone a 0 el %r3
 ld [length], %r1 ! %r1 = length
for: andcc %r1, %r1, %r0 ! Chequea elementos restantes
 be done ! Si no hay mas elementos termina
 addcc %r1, -4, %r1 ! Actualiza índice al array
 ld %r1,[arrayA],%r5
 addcc %r3, %r5, %r3 ! Suma un nuevo elemento a r3
 ba for ! Vuelve al for
done: jmpl %r15 + 4, %r0 ! Vuelve al proceso invocante
length: 20 ! 5 numeros (20 bytes)
arrayA: .dwb 5 ! Contenido de ArrayA
.end

Programa que suma los elementos de un array

```
long arrayA[5] ;  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ];  
    }  
    return(0);  
}
```

Versión 3

```
! Este programa suma LENGTH números  
! Uso de registros: %r1 - Indice de arrayA  
!  
!           %r3 - Suma parcial  
!  
!           %r5 - Guarda un elemento de arrayA  
  
.begin  
length .equ      20  
        .org 2048          ! El programa empieza en la direcc. 2048  
main:  andcc    %r3, %r0, %r3 ! Pone a 0 el %r3  
       add      %r0, length, %r1      ! %r1 = length  
for:   andcc    %r1, %r1, %r0 ! Chequea elementos restantes  
       be done          ! Si no hay mas elementos termina  
       addcc    %r1, -4, %r1 ! Actualiza índice al array  
       ld      %r1,[arrayA],%r5  
       addcc    %r3, %r5, %r3 ! Suma un nuevo elemento a r3  
       ba      for          ! Vuelve al for  
done:  jmpl    %r15 + 4, %r0 ! Vuelve al proceso invocante  
arrayA: .dwb     5          ! Reserva espacio para ArrayA  
.end
```

Cual es más eficiente: versión 2 o versión 3?

Hay alguna limitación en el valor posible de la constante length?

Programa que suma los elementos de un array

```
long arrayA[5] ;  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ];  
    }  
    return(0);  
}
```

! Este programa suma LENGTH números
! Uso de registros: %r1 - Indice de arrayA
!
! %r3 - Suma parcial
!
! %r5 - Guarda un elemento de arrayA
!
! %r2 - Guarda la dirección de arrayA

.begin

length .equ 20
.org 2048 ! El programa empieza en la direcc. 2048

main: andcc %r3, %r0, %r3 ! Pone a 0 el %r3
add %r0, length, %r1 ! %r1 = length
add %r0, arrayA, %r2 !! %r2 = `puntero a arrayA

for: andcc %r1, %r1, %r0 ! Chequea elementos restantes
be done ! Si no hay mas elementos termina
addcc %r1, -4, %r1 ! Actualiza índice al array
ld %r1, %r2, %r5
addcc %r3, %r5, %r3 ! Suma un nuevo elemento a r3
ba for ! Vuelve al for

done: jmpl %r15 + 4, %r0 ! Vuelve al proceso invocante

arrayA: .dwb 5 ! Reserva espacio para ArrayA

.end

Versión 4

¿Qué limitación tiene la dirección de memoria posible para arrayA?

Programa que suma los elementos de un array

```
long arrayA[5] ;
long SumaParcial;

int main ()
{
    int i;
    SumaParcial=0;
    for ( i = 0; i < 6; i++ )
    {
        SumaParcial=SumaParcial+ a[ i ];
    }
    return(0);
}
```

```
! Este programa suma LENGTH números
! Uso de registros: %r1 - Indice de arrayA
!
!           %r3 - Suma parcial
!
!           %r5 - Guarda un elemento de arrayA
!
!           %r2 - Guarda la dirección de arrayA

.begin
length .equ      20
          .org 2048           ! El programa empieza en la direcc. 2048
main:   andcc    %r3, %r0, %r3 ! Pone a 0 el %r3
        add      %r0, length, %r1 ! %r1 = length
        sethi    arrayA, %r2   !
        srl      %r2, 10.%r2 ! %r2 = `puntero a arrayA
for:    andcc    %r1, %r1, %r0 ! Chequea elementos restantes
        be done   ! Si no hay mas elementos termina
        addcc    %r1, -4, %r1 ! Actualiza índice al array
        ld       %r1, %r2, %r5
        addcc    %r3, %r5, %r3 ! Suma un nuevo elemento a r3
        ba      for         ! Vuelve al for
done:   jmpl    %r15 + 4, %r0 ! Vuelve al proceso invocante
arrayA: .dwb     5           ! Reserva espacio para ArrayA
.end
```

Versión 5

Necesario para leer datos de un periféricos. Porqué?
También necesario para pasar punteros a una subrutina.

Programa que suma los elementos de un array

```
long arrayA[5]={25, 10, 33, -5, 7};  
long SumaParcial;  
  
int main ()  
{  
    int i;  
    SumaParcial=0;  
    for ( i = 0; i < 6; i++ )  
    {  
        SumaParcial=SumaParcial+ a[ i ];  
    }  
    return(0);  
}
```

! 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
.org 2048
.equ 3000
ld [length], %r1
ld [address], %r2
andcc %r3, %r0, %r3
andcc %r1, %r1, %r0

a_start
loop:
elements
be done
addcc %r1, -4, %r1
addcc %r1, %r2, %r4
ld %r4, %r5
addcc %r3, %r5, %r3
ba loop
jmpl %r15 + 4, %r0

done:
length: 20
address: a_start
a:
.org a_start
25
-10
31
-5
7
.end

! Start program at 2048
! Address of array a
! %r1 → length of array a
! %r2 → address of a
! %r3 → 0
! Test # remaining

! Finished when length=0
! Decrement array length
! Address of next element
! %r5 → Memory[%r4]
! Sum new element into r3
! Repeat loop.
! Return to calling routine
! 5 numbers (20 bytes) in a

! Start of array a
! length/4 values follow

SUBRUTINAS

¿Qué es una subrutina?

Subrutinas vs Branch

¿Como se llega a una subrutina

¿Como termina una subrutina?

¿Como se intercambian argumentos con una subrutina?

LLAMADO A SUBRUTINAS

Parámetros por registros

! Programa que suma dos numeros

begin

.org 2048

.

.

ld [x], %r1
ld [y], %r2
call sbr_add
st %r3, [z]

.

.

jmpl %r15+4,%r0

! Subrutina sbr_add

! suma el contenido de %r1 al de r2 y

! devuelve el resultado en %r3

sbr_add: addcc %r1, %r2, %r3
jmpl %r15 + 4, %r0

x: 15

y: 9

z: 0

.end

Este programa se cuelga indefectiblemente. Qué líneas de código agregaría para que no ocurra

LLAMADO A SUBRUTINAS

Parámetros por stack

! Programa que suma dos numeros

```
.begin
.org      2048
.
.
.
ld      [x], %r1
ld      [y], %r2
addcc  %r14, -4, %r14
st      %r1, %r14
addcc  %r14, -4, %r14
st      %r2, %r14
call    sbr_add
ld      %r14,%r3
addcc  %r14, 4, %r14
st      %r3, [z]
.
.
.
jmpl   %r15+4,%r0
```

! Subrutina Sbr_add

! Le llegan dos numeros por stack

! devuelve su suma por stack

```
sbr_add: ld      %r14, %r8
          addcc  %r14, 4, %r14
          ld      %r14, %r8
          addcc  %r8, %r9, %r10
          st      %r10, %r14
          jmpl   %r15 + 4, %r0
```

x: 15

y: 9

z: 0

.end

(Pasar parámetros por registro) VS. (Pasar parámetros por stack)

Qué líneas de código agregaría para que no se cuelgue?

LLAMADO A SUBRUTINAS

Parámetros por área reservada en memoria

! Programa que suma dos numeros

```
.begin
.org 2048
.
.
st  %r1, [x]
st  %r2, [x+4] | Copia operandos al área
addcc %r0, x , %r5 | Pasa puntero al área
call sbr_add
st  [x+8], %r3
.
.
jmpl %r15+4,%r0
```

! Subrutina Sbr_add

! Por %r5 le llega el puntero al area

! de memoria donde estan los arg. de entr y de salida

```
sbr_add: ld  %r5, %r8
        ld  %r5 + 4, %r19
        addcc %r8, %r9, %r10
        st   %r10, %r5 + 8
        jmpl %r15 + 4, %r0
x:    .dwb 3
.end
```

LLAMADO A SUBRUTINAS

Parámetros por área reservada en memoria

! Programa que suma dos numeros

```
.begin
.org 2048
.
.
st %r1, [x]
st %r2, [x+4]
sethi x , %r5
srl %r5,10, %r5
call sbr_add
st [x+8], %r3
.
.
jmpl %r15+4,%r0
```

Forma alternativa del programa anterior

Copia operandos al área

Pasa puntero al área

! Subrutina Sbr_add

! Por %r5 le llega el puntero al area

! de memoria donde estan los arg. de entr y de salida

sbr_add: Id %r5, %r8

```
Id    %r5 + 4, %r9
addcc %r8, %r9, %r10
st    %r10, %r5 + 8
jmpl  %r15 + 4, %r0
x:   .dwb 3
.end
```

LLAMADO A SUBRUTINAS

Comparar pasaje de parámetros

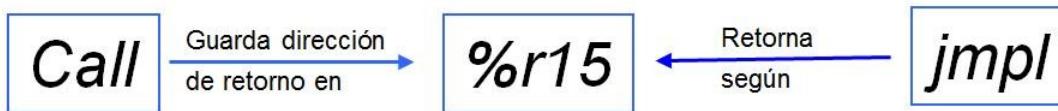
- Por registros
- Por la pila
- Por área reservada de memoria

Programas con varios módulos

- Directivas .global y .extern
- Limitación del ensamblador Arctools
- Convertir todos los programas anteriores a dos módulos

LLAMADO A SUBRUTINAS

Dirección de retorno



Problema:

Rutinas anidadas sobreescriben `%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	<code>%r15</code> se guarda en
Por registros	Registro no utilizado
Por área reservada en memoria	Incluido en área reservada
En la pila	Pila

Macros

- ❖ ¿Qué es una macro?
- ❖ ¿Para qué sirve?
- ❖ Argumentos

! Programa sin macros

```
.begin
.org      2048
.
.
.
Id      [x], %r1
Id      [y], %r2
addcc  %r14, -4, %r14
st      %r1, %r14
addcc  %r14, -4, %r14
st      %r2, %r14
call    sbr_add
Id      %r14,%r3
addcc  %r14, 4, %r14
st      %r3, [z]
.
.
```

! Programa con macros

```
.begin
.org      2048
.
.
.
Id      [x], %r1
Id      [y], %r2
push   %r1
push   %r2
call   sbr_add
pop    %r3
st     %r3, [z]
.
.
```

Macros

! Programa: suma dos numeros SUBRUTINA
.begin

.org 2048

.

.

ld [x], %r1

ld [y], %r2

call sbr_add

st %r3, [z]

.

.

jmpl %r15+4,%r0

! Subrutina Sbr_add

! suma el contenido de %r1 al de r2 y

! devuelve el resultado en %r3

sbr_add: addcc %r1, %r2, %r3

jmpl %r15 + 4, %r0

x: 15

y: 9

z: 0

.end

! Programa: suma dos numeros MACRO
.begin

.macro mcr_add Reg1, Reg2, Reg3

addcc Reg1, Reg2, Reg3

.endmacro

.org 2048

.

ld [x], %r1

ld [y], %r2

mcr_add %r1, %r2, %r3

st %r3, [z]

.

.

jmpl %r15+4,%r0

x: 15

y: 9

z: 0

.end

Macros vs Subrutinas

Macro

Se accede en tiempo de ensamblado (expansión de macros) convirtiéndola en su código equivalente..

Sus parámetros son interpretados por el ensamblador y reemplazados por lo que corresponda en cada lugar que es invocada

Su código de máquina está repetido tantas veces como la macro fue invocada

Subrutina

Se accedida por un CALL en tiempo de ejecución y termina con un JMPL en tiempo de ejecución

Sus parámetros (valores) le llegan en tiempo de su ejecución (pila o registros)

Su código de máquina está localizado en un lugar específico y único en memoria

Comparar:

- *Uso de memoria*
- *Velocidad de ejecución*

Localización de variables

- ❖ Registros
- ❖ “Segmento de datos” (RAM)
- ❖ Stack (RAM)
- ❖ Comparar
 - Velocidad de acceso
 - Alcance
 - Vida

CÓDIGO DE MÁQUINA

Formato de instrucciones

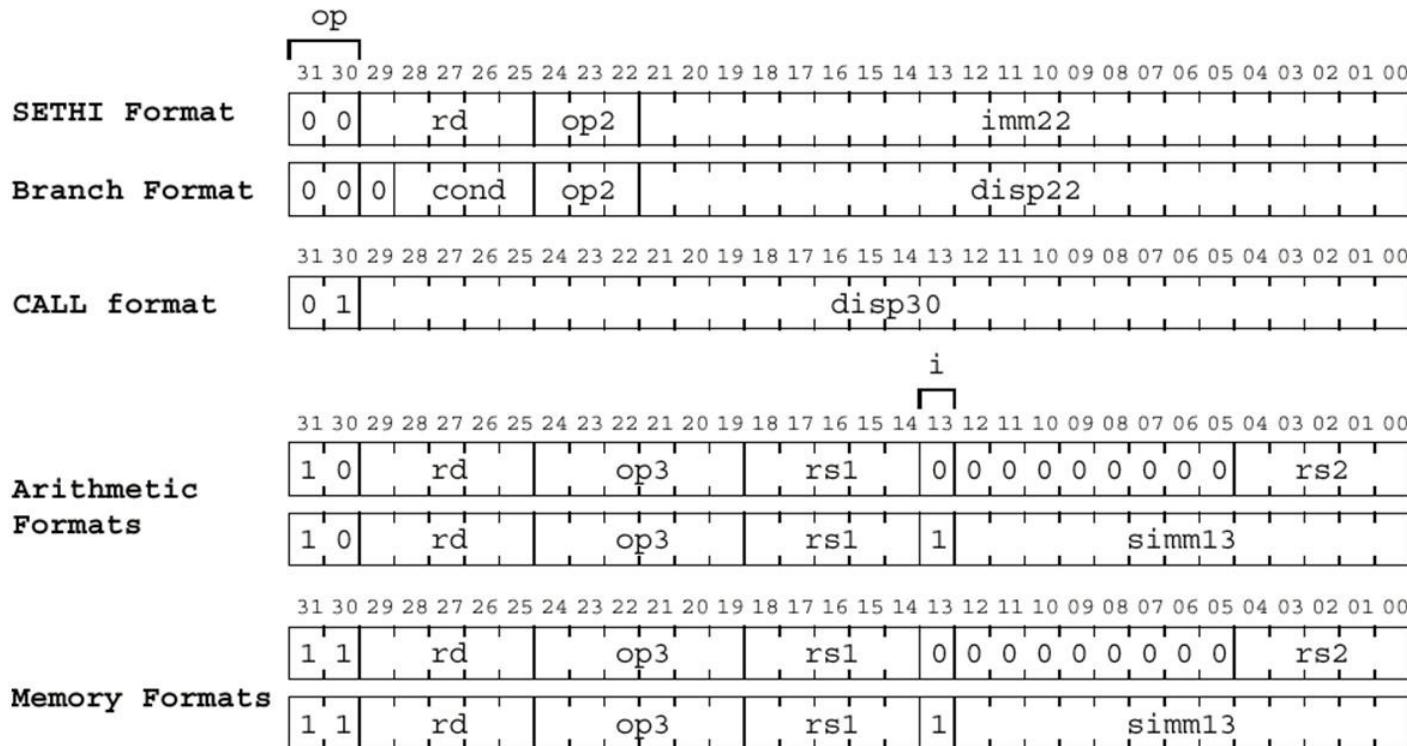
- ❖ Todas las instrucciones ocupan 32 bits
- ❖ No todas las instrucciones siguen el mismo formato
- ❖ Se definen grupos de bits a los que se da un significado
- ❖ Todas las instrucciones ARC pueden agruparse en 5 formatos:
(respecto de su código de máquina)
 - Formato de la instrucción Sethi
 - Formato 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
00	SETHI/Branch
01	CALL
10	ArithmetiC
11	Memory

op2	Inst.
010	branch
100	sethi

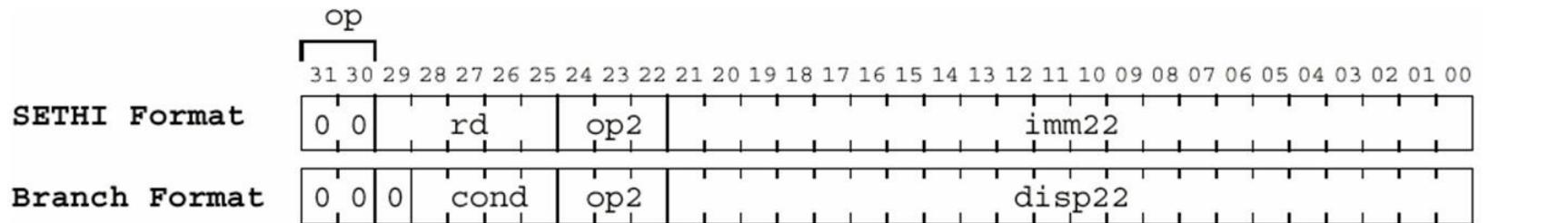
op3 (op=10)
010000 addcc
010001 andcc
010010 orcc
010110 orncc
100110 srl
111000 jmp1

op3 (op=11)
000000 ld
000100 st

cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba

CÓDIGO DE MÁQUINA

Formatos de instrucción



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

op2=100 => SETHI

rd=registro de destino
operando= const. *imm22*

→ cond=n, z, v, c
Desplaz. = constante *disp22*.

0001 be
0101 bcs
0110 bneg
0111 bvs
1000 ba

Medido en palabras de 4 bytes

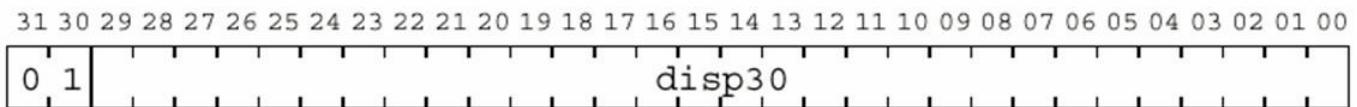
4 x disp22 (2 shift left)

PC=PC + (4 × sign_ext(*disp22*)

CÓDIGO DE MÁQUINA

Formatos de instrucción

CALL format

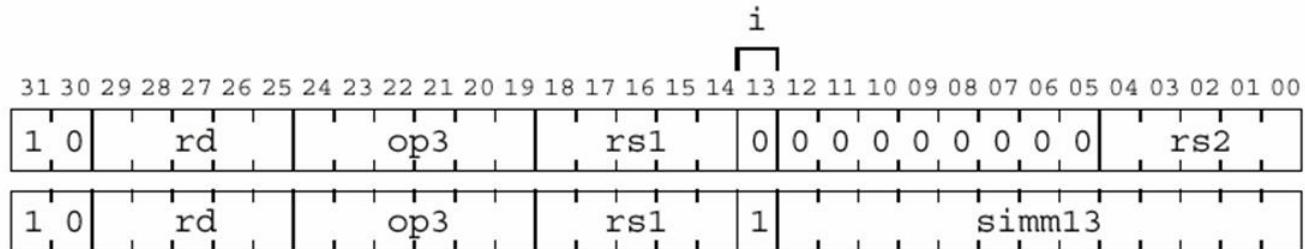


CALL op=01 desplazamiento → constante *disp30*

CÓDIGO DE MÁQUINA

Formatos de instrucción

Arithmetic
Formats



ARITMETICA op=10 op3

010000 addcc

1er. registro origen = rs1

010001 andcc

2do reg. origen = rs2

si $i=0$

010010 orcc

2do reg. origen = constante $simm13$

si $i=1$

010110 orncc

Reg. destino = rd

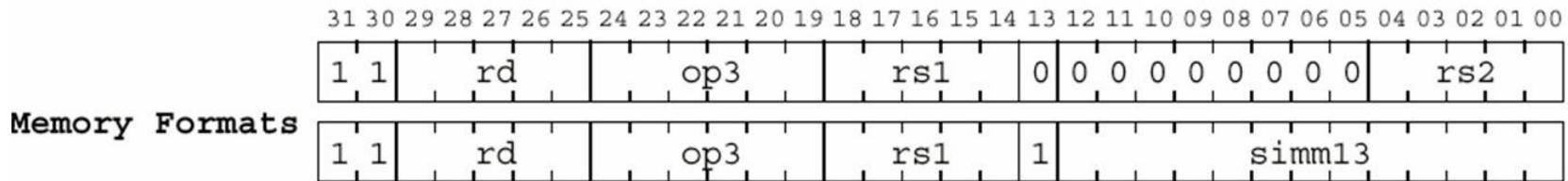
100110 srl

111000 jmpl

$simm13$ se interpreta extendiendo el signo a 32 bits

CÓDIGO DE MÁQUINA

Formatos de instrucción



ACC. MEMORIA: $op=11 \quad op3$

000000 ld rd=reg.destino

000100 st rd=reg.origen

direcc. de memoria (ld o st)

= $rs1 + rs2$

si $i = 0$

= $rs1 + simm13$ (constante)

si $i = 1$

Ejemplo

! This program adds two numbers

```
.begin
.org 2048
main: 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
      jmpl %r15 + 4, %r0 ! Return
x:   15
y:   9
z:   0
.end
```

•	(ARCTools Version 2.1.2)	HexLoc	DecLoc	MachWord	Label	Instruction	Comment
•						.org 2048	
•		00000800	0000002048	c2002814	main:	ld [2068], %r1	! Load x into %r1
•		00000804	0000002052	c4002818		ld [2072], %r2	! Load y into %r2
•		00000808	0000002056	86804002		addcc %r1, %r2, %r3	! %r3 = %r1 + %r2
•		0000080c	0000002060	c620281c		st %r3, [2076]	! Store %r3 into z
•		00000810	0000002064	81c3e004		jmpl %r15, 4, %r0	! Return
•		00000814	0000002068	0000000f	x:		
•		00000818	0000002072	00000009	y:		
•		0000081c	0000002076	00000000	z:		

MODOS DE DIRECCIONAMIENTO

Algunos Ejemplos

Modo	
Inmediato	Constante incluida en la instrucción
Por registro	El registro tiene el dato
Directo o absoluto	Dirección de memoria incluida en la instrucción
Indirecto	Direcc de memoria dónde esta el puntero al dato (poco usado, lento)
Indirecto por Registro	El registro tiene el puntero al dato
Indexado por Registro	Un registro da la dirección inicial el otro un incremento (arrays)

- Un ISA CISC tiene varios otros modos de direccionamiento



- Cuáles están en el ISA ARC?

Para cada uno

- Datos al tiempo de compilación?
- Datos al tiempo de ejecución?

Conformación de un set instrucciones

Set de instrucciones = Conjunto de instr. + Registros disponibles

➤ Características

- ❖ 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)
- ❖ Formas de indicar la ubicación de los datos=“modos de direccionamiento”

Modelos de Arquitectura de Set de Instrucciones

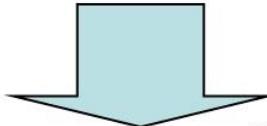
- **CISC** = *Complex Instruction Set Computer*
- **RISC** = *Reduced Instruction Set Computer*

RISC vs CISC

- Cantidad de trabajo hecho en cada instrucción
- Espacio ocupado por cada instrucción (Código de máquina)
- Tiempo de ejecución de cada instrucción (Ciclos de reloj)
- Direccionar memoria (Instrucciones que la acceden y modos de direccionar)
- La cantidad de registros disponibles
- Uso del stack (inmerso en la instrucción o implementable externo a ella)

Arquitecturas CISC

- Certo grado de complejidad en sus instrucciones
- Código binario de largo variable (entre 1 y 15 bytes)
- El número de operandos y su tipo dependen de la instrucción

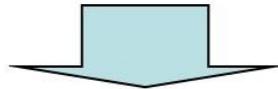


- ✓ Complica la lógica para encontrar las instrucciones en memoria
- ✓ Complica la decodificación de las instrucciones
- ✓ Complica la lógica de interconexiones dentro del procesador

Ejemplos: x86, x86-32, x86-64

Arquitecturas RISC

- Certo grado de simplificación en sus instrucciones
- En todas ellas el código binario tiene la misma longitud
- Todas están localizadas en direcciones de memoria múltiplos de cuatro
- Las únicas instrucciones que acceden a memoria son ‘guardar’ y ‘recuperar’;
- Operaciones aritméticas y lógicas sólo entre registros (modos de direccionamiento)
- Dispositivos de E/S mapeados en memoria



- ✓ Simplifica la lógica para encontrar las instrucciones en memoria
- ✓ Simplifica la decodificación de las instrucciones
- ✓ Ese hardware más simple permite optimizarlo para obtener más velocidad

¿Porqué?

Ejemplos: SPARC, MIPS, ARM, CORTEX

desde supercomputadoras hasta tablets y smart phones

RISC vs CISC

- ¿Con cuál de ellos debería ser más fácil implementar compiladores?
- ¿Con cual de ellos se genera mayor cantidad de instrucciones de Assembler?
- ¿Cuál requiere mayor memoria para almacenar el programa?

