Segundos parciales resueltos Estructura del Computador II

PARCIAL 1

Indicar la oración correcta. *

- ✓ Un programa ensamblador convierte el programa que está escrito en lenguaje simbólico a lenguaje de máquina, para su almacenamiento en el disco rígido.
- ✗ Un programa intérprete convierte el programa que está escrito en lenguaje simbólico a assembler, para su almacenamiento en memoria principal.
- ✗ Un programa compilador convierte el programa que está escrito en lenguaje de máquina a lenguaje de bajo nivel, para su almacenamiento en el disco rígido.
- ✗ Un programa intérprete convierte el programa que está escrito en lenguaje de alto nivel a lenguaje de alto nivel, para su almacenamiento en memoria principal.

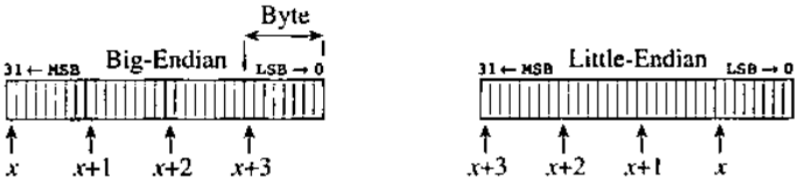
En Little Endian el byte menos significativo va en la dirección más _____ .

En Big Endian el byte _____ significativo se encuentra en la dirección más baja.

En Little Endian, el byte menos significativo va en la dirección más baja.

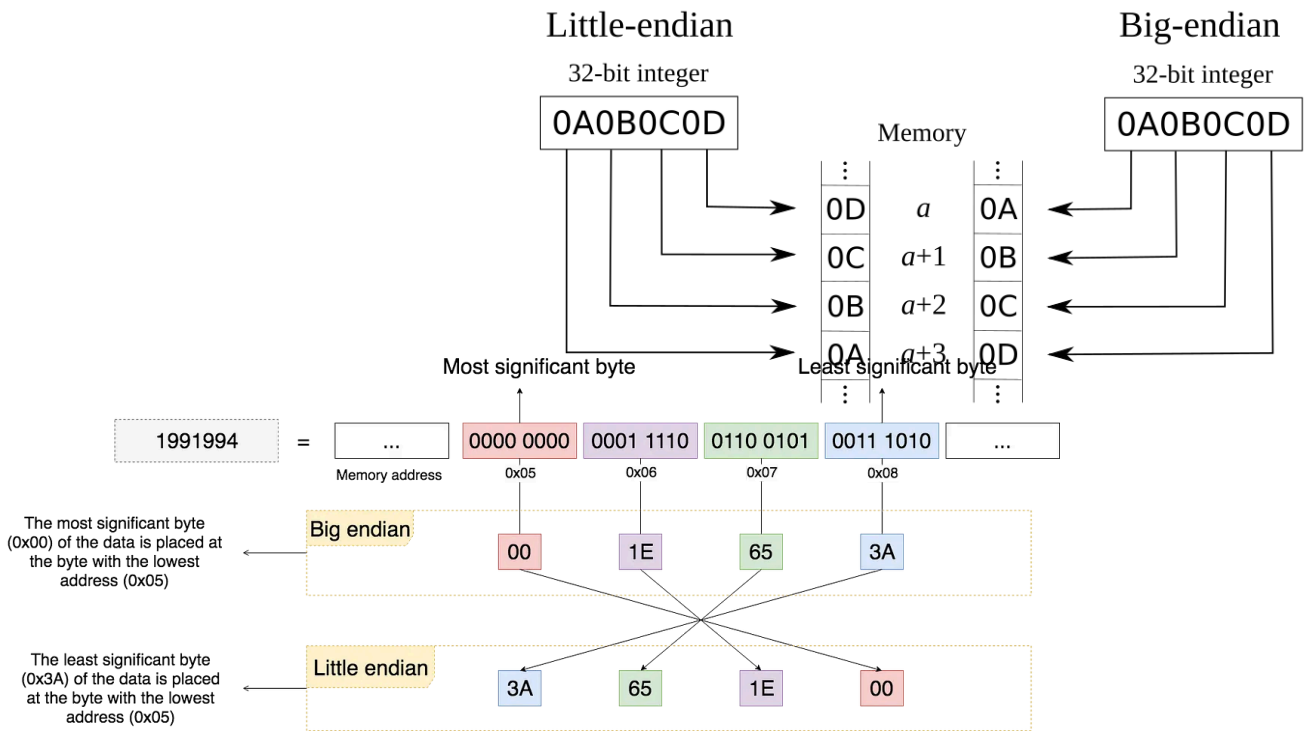
En Big Endian el byte más significativo va en la dirección más baja.

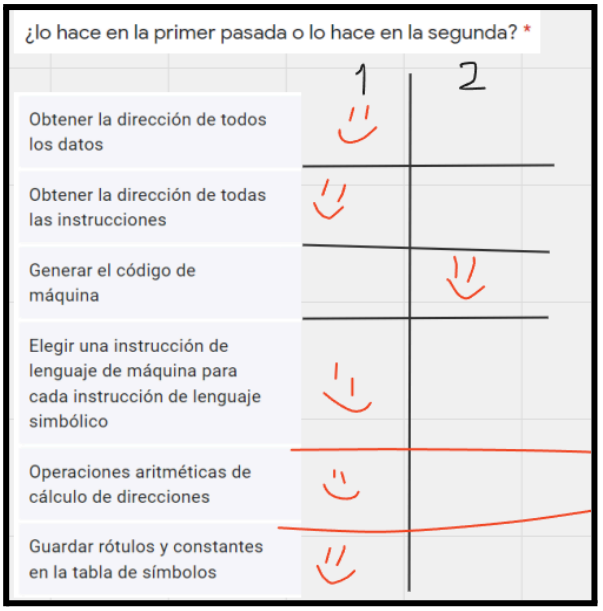
X: dirección más baja



La dirección de la palabra es *x* en ambos formatos

Figura 4.3 • Formatos de almacenamiento *big endian* y *little endian*.





La mayoría de los ensambladores recorren dos veces el texto escrito en lenguaje simbólico, por lo que se los conoce como "ensambladores de dos pasadas". En la primera pasada el ensamblador se dedica a determinar las direcciones de todos los datos e instrucciones del programa y a seleccionar qué instrucción del lenguaje de máquina debe generarse para cada instrucción del lenguaje simbólico, pero sin generar aún el código de máquina.

se asignará a dicha dirección. Durante esta pasada, el ensamblador realiza también cualquier operación aritmética necesaria e inserta las definiciones de todos los rótulos y constantes en una **tabla de símbolos**.

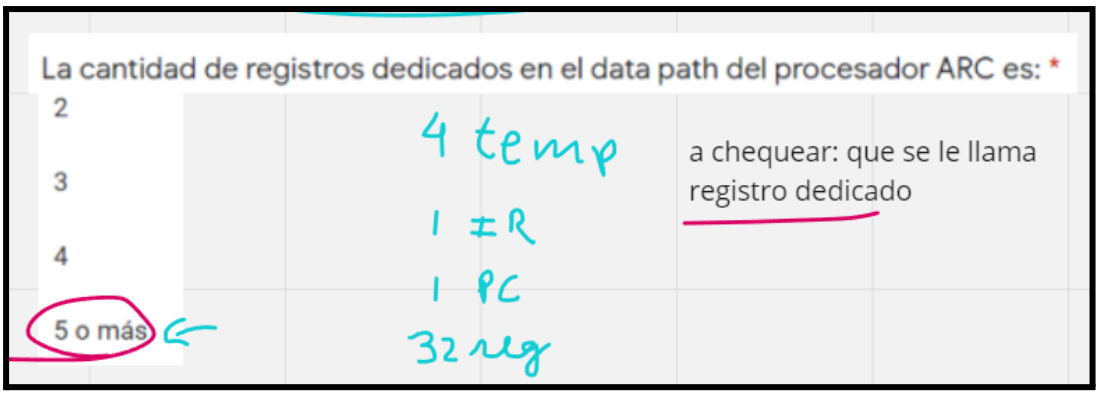
por lo que durante la segunda pasada podrá generar el código de máquina, insertando en el mismo los valores de los símbolos, ya conocidos para ese momento.

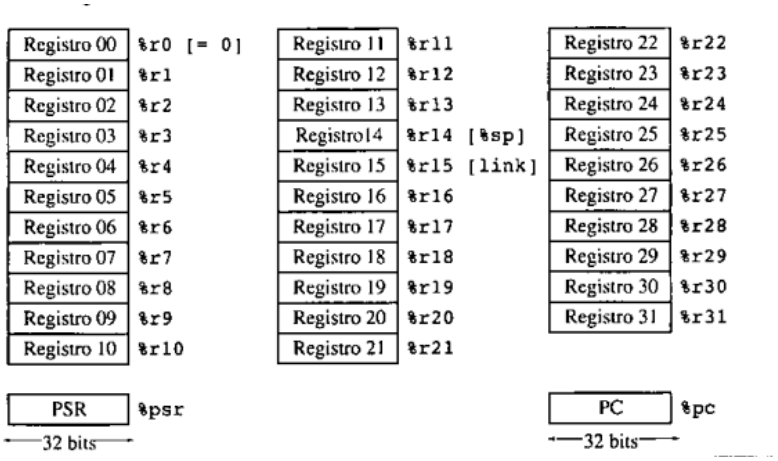
Paso a paso qué hace:

¿Qué número guarda en z el siguiente p

```
        .begin
        .org 2048
        ld [x], %r1
        ld [y], %r2
        subcc %r1, %r2, %r0
        bneg true1
        subcc %r2, %r1, %r0
        bneg true2
        ba fin
true1:  st %r2, [z]
        ba fin
true2:  st %r1, [z]
        ba fin
A:      2
B:      11
x:      9
y:      3
z:      0
fin:
        .end
```

Cargo X en %r1, X = 9
Cargo Y en %r2, Y = 3
Hago la resta %r1 - %r2 = 9 - 3 = 6
bneg si el resultado es negativo (N = 1), salto a true1, como NO es negativo, sigo
Hago la resta %r2 - %r1 = 3 - 9 = -6
bneg si el resultado es negativo (N = 1), salto a true2, como SI dio negativo, salto
true2 almacena en la etiqueta Z el contenido de %r2, por lo tanto Z = 9.





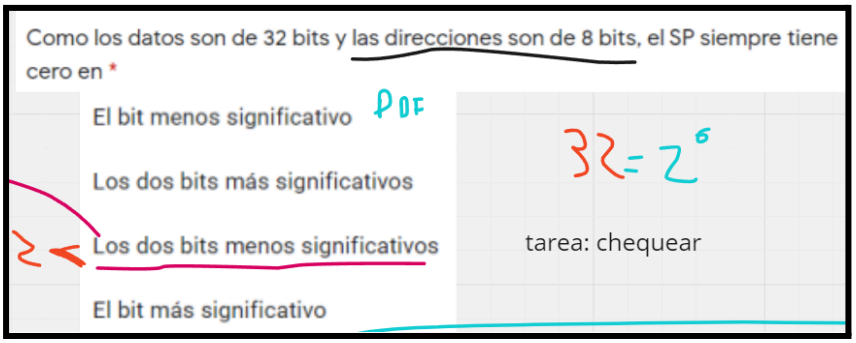
Tenemos:

- 32 registros para el uso del usuario (%r0 al %r31)
- 1 IR (registro de instrucciones)
- 1 PC (contador de programa)
- 4 registros temporales

Los registros `%r0–%r31` son accesibles directamente por el usuario. El registro `%r0` siempre contiene un 0, y esto no puede modificarse. El `%pc` es el contador de programa, que apunta a la dirección a ser leída desde la memoria principal. El usuario tiene acceso directo al contador de programa solo a través de las instrucciones `call` y `jmpl`. Los registros temporarios se utilizan para interpretar el conjunto de instrucciones de ARC, por lo que no son accesibles para el usuario. El registro `%ir` contiene la instrucción en ejecución, y tampoco es accesible al usuario.

Los que no son del usuario son registros dedicados en el data path, son 5 o más.

Es 1 solo, el %r14.



Bueno, lo entendí.

Como para poder manejarme con la pila, debo sumar y restar 4 según pusheo o popeo, para que funcione bien, el stack pointer debe ser múltiplo de 4.

La forma de asegurar que un número en binario sea múltiplo de 4 es que sus dos últimos bit sean 0.

Esta propiedad es útil en la programación de bajo nivel, especialmente cuando se trabaja con la alineación de memoria, ya que garantiza que los múltiplos de 4 estén alineados adecuadamente en arquitecturas que requieren alineación en direcciones múltiplos de 4.

en la interpretación de las instrucciones, tal como se verá en la sección 6.2.4. El contador de programa solo puede contener valores que sean múltiplos de 4, por lo que los dos bits menos significativos de %pc pueden ser conectados eléctricamente a cero.

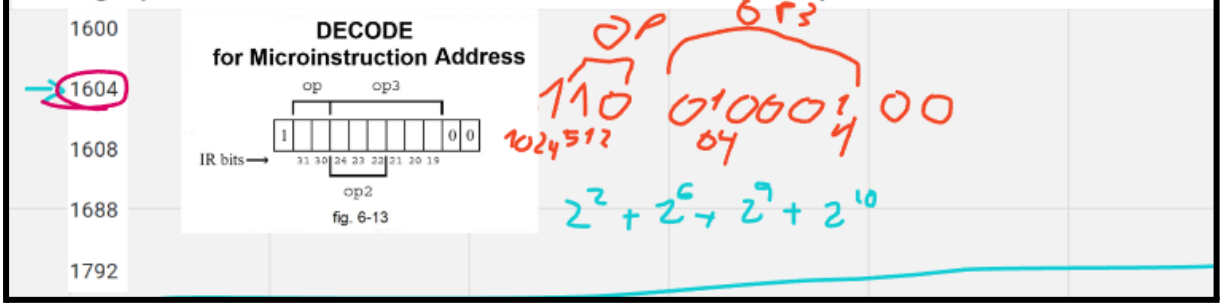
¿Cómo se podría implementar en una sola instrucción mov %r3, %r1 (mover el contenido del registro %r3 al registro %r1)? *

subcc %r0, %r3, %r1

xnor %r3, %r0, %r1

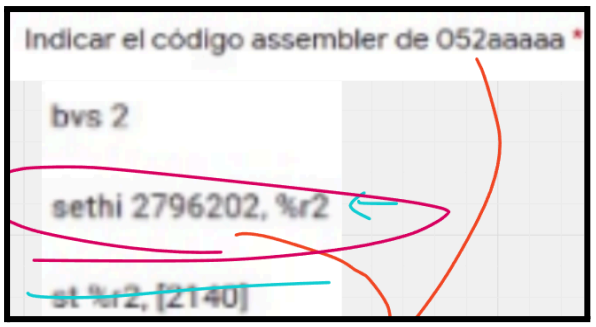
or %r0, %r3, %r1

Indicar la dirección dentro de la ROM de Control dónde se encuentra el micro código para la instrucción andcc si el OP es 10 y el op3 es 010001 *



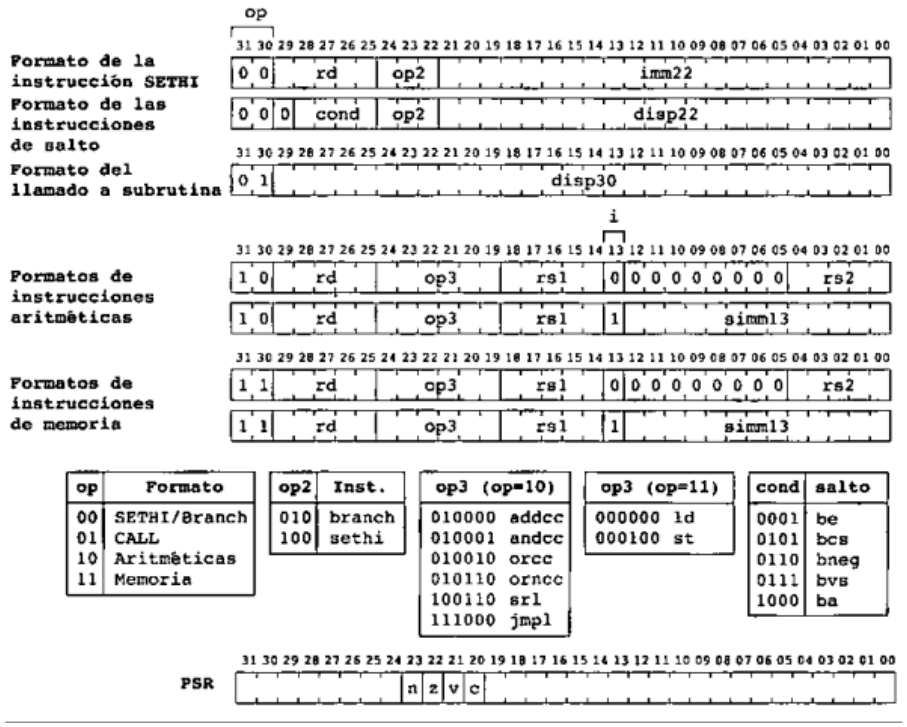
1 OP OP3 00
OP = 10
OP3 = 010001

Entonces la instrucción queda 1 10 010001 00, pasando a decimal obtengo la dirección de la instrucción: 1604.



052AAAAA

Pasamos a binario: 0000 0101 0010 1010 1010 1010 1010 1010



Miro los 2 primeros: 00 SETHI o Branch OP. Ahora los OP2: (8 9 10) 1 0 0, es SETHI.

Puedo separar entonces:

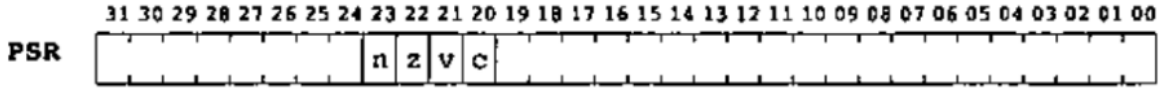
00 00010 (2) 100 10101010101010101010 rd = %r2, cte = un número feo. Mi instrucción será SETHI cte, %r2

PARCIAL 2

El registro %psr se actualiza luego de cada operación que realiza la ALU.

Seleccione una:

- Verdadero
- Falso



El campo cond identifica el tipo de salto condicional, que se basa en los códigos de condición (n, z, v y c) del registro de estado PSR, de acuerdo con lo que puede observarse en la parte inferior de la figura 4.10. El resultado de la ejecución de cualquier instrucción

Falso. Cómo se modifica con los flags, hay operaciones de la ALU que no modifican los códigos de condición.

Al intentar compilar este código en lenguaje ensamblador para el procesador AI

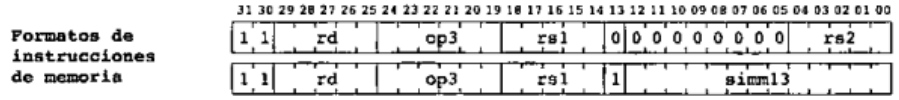
```
        .begin
        .org 2048
                X .equ 4000   ! 0xfa0
main:           sethi X, %r2
                srl %r2, 10, %r2
set:            st %r2, [dispo]
                call Rutina
                addcc %r2, -1, %r2
                bne set
fin:            jmpl %r15 + 4, %r0

        .org  a00004h
dispo:  .dwb 1
Rutina:         jmpl %r15 + 4, %r0

        .end
```

Seleccione una:

- a. No compila porque la instrucción 3 no se puede codificar
- b. No compila porque la instrucción 1 no se puede codificar

La instrucción 1 funciona bien, la constante de 4000 no supera los 22 bits.

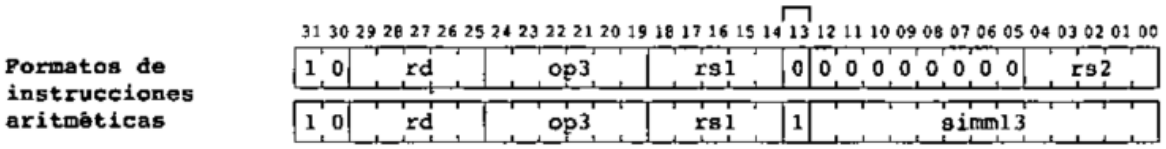
La instrucción 3 del tipo storage quiere almacenar la constante A00004 en el registro %r2, pero viendo el formato de la instrucción de memoria, la constante puede tener como máximo 13 bits, por lo tanto en este caso no se puede codificar. **Opción b)**



La compilación de la siguiente instrucción: add %r0, 2300, %r1 ! r1 <- 2300 re:

Seleccione una:

- a. Error porque la instrucción no existe
- b. Error porque 2300 es número fuera del rango de representación
- c. El compilación existosa pero la instrucción ensamblada no es la desead
- d. Compilación exitosa



2300 debe tener 13 bits, 2300 = 100011111100 cumple, así que podría compilar

En la arquitectura de ARC, no es necesario que la salida del multiplexor del bus conectado con el registro %r0.

Seleccione una:

- Verdadero
- –

Los demás registros adoptan un esquema similar, salvo por algunas excepciones. El registro `%r0` siempre contiene un 0, el que no puede modificarse. Por consiguiente, el registro `%r0` no tiene entradas desde el bus C, ni tampoco desde el decodificador C y, por ende, no requiere *flip flops* (véase el problema 6.11). El registro `%ir` tiene salidas adicionales

tradas del decodificador para seleccionar el registro correspondiente. La salida 0 del decodificador C no se utiliza debido a que el registro `%r0` no puede ser escrito. Los índi-

El siguiente código en C

```
unsigned int a = 3;
int b=-5;
unsigned int c= 0;
if (a>b){
        c=8;
}
else {
c=-20; }
```

a pesar de las advertencias del compilador, se ensambla como:

```
or      %r0, 3, %r4      ! r4 = a
or      %r0, -5, %r10    ! r10 = b
or      %r0, %r0, %r20   ! r20 = c
                         subcc   %r4, %r10, %r0
                         bleu    ELSE                    ! salte si es menor
signo
                         or      %r0, 8, %r20
                         ba      SIGUE
        ELSE:            or      %r0, -20, %r20
        SIGUE:  …
```

Al ejecutarse el código la variable c vale:

Seleccione una:

- a. -20
- b. 4 294 967 276
- c. 0
- d. ninguna es correcta
- e. 8

Menor o igual sin signo	BLEU rY_menorIgual_rX	(C==0) || (Z==0)

a = 3
b = -5
c = 0

a - b = 3 + 5 = 8, entonces no salta, si miramos sin signo 3 < 5
Entonces, salta a ELSE

c = -20. **Opción a)**

¿Que tarea hace un ensamblador?

Seleccione una:

- a. Aclarar la legibilidad del programa
- b. Permite la reubicación del código.
- c. facilitar la ejecución de saltos por la incorporación de labels
- d. Agilizar la escritura de programas
- e. Codificar instrucción por instrucción en forma unívoca

Opción e)

PARCIAL 3

2. Un procesador ARC al ejecutar el siguiente código:

```
                addcc %r0, 1, %r1          ! r1 <- 1
                add %r0, 3, %r27
                add %r0, %r27, %r17
                sub %r27, %r17, %r0        ! r27 - r17
                be  RETURN                 ! salte si es igual
                inc  %r1
     RETURN:    jmpl %r15+4, %r0
```

devuelve:

a. r1 en dos
b. r1 en uno
c. r1 en cero

%r1 = 1
%r27 = 3
%r17 = 3
3 - 3 = 0
Son iguales -> Voy a RETURN
Termino

%r1 = 1. **Opción b)**

3. Como el usuario programador no tiene acceso directo a los flags del PSR, no puede fijar Z=1 C=1 V=1 N=0 simultáneamente luego de una sucesión arbitraria de instrucciones de assembler.

Verdadero.

4. Un procesador ARC usa una ALU con 16 funciones distintas a las originales de Murdocca. Para tener el mismo ISA alcanza con cambiar en forma adecuada la memoria de microprograma.

5. En las operaciones aritméticas, el campo *simm13* se puede usar para operar directamente con un dato en memoria.

Falso.

7. Para determinar el endianness de una máquina se propone utilizar el siguiente código en C:

```
void main()
{       int x = 1;
        char *y = (char*)&x;
        printf("%d\n",*y);
        return;
}
```

a. si la máquina es little-endian, se imprime "1"
b. si la máquina es big-endian, se imprime "0"
c. *a* y *b* son correctas
d. ninguna de las anteriores es correcta

Imprime el valor del byte menos significativo de X

La salida es 1, siendo el byte menos significativo. Por lo tanto la opción es a).