Compiladores

Procesos y lenguajes.-

Procesos principales

Compilación: es el proceso de traducción de un programa escrito en lenguaje de alto nivel a otro programa en lenguaje ensamblador que es funcionalmente equivalente.

Ensamblado: es el proceso de traducción de un programa en lenguaje ensamblador a un programa en lenguaje máquina que es funcionalmente equivalente.

Enlazado (*linking*): proceso que consiste en unir en un programa único a los distintos módulos que fueron ensamblados de forma separada.

Carga: implica el traslado del programa a memoria y su preparación para ser ejecutado.

Compilación:

var1=var2+8

Análisis lexicográfico: proceso en el cual se reconocen los identificadores var1 y var2, definiciones como la de la constante “8” y los delimitadores “=” y “+”.

Analizador (*parser*): reconoce que la sentencia utilizada es del tipo:

identificador “=” expresión

Análisis sintáctico: se analiza la expresión anterior, de forma de detectar formas del tipo:

identificador “+” constante

Análisis de nombres...

Análisis de tipos...

Asignación de acciones y generación de código: Es la acción de asociar las sentencias de programa con la secuencia apropiada del lenguaje ensamblador.

para el ejemplo anterior:

```
ld    [var2],%r0,%r1    !cargar la variable var2 en un registro
add   %r1,8,%r2         !calcular resultado de la expresión
st    %r2,%r0,[var1]    !asignación
```

¿Cómo convierte el compilador locación en memoria, datos, instrucciones aritméticas y sentencias?

Compilación: almacenamiento de variables en memoria

Solo las **variables globales** (o estáticas en C) tienen direcciones conocidas al momento de la compilación.

Para las **variables locales** (o variables automáticas): son implementadas con una pila de tipo LIFO.

Compilación: estructuras de datos

```
struct point{
        int x,
        int y,
        int z
}

//declaración
struct point pt;
```

El programador puede referirse a cada elemento de la siguiente forma: pt.x

Lo que hace el compilador es acomodar estas tres componentes en memoria de forma consecutiva, La dirección de memoria de la estructura completa se toma como la dirección del primer elemento x.

ejemplo: cargar la el contenido de Y en %r1

ld[pt+4],%r1

¿Que diferencia tenemos para los arreglos?

Compiladores: secuencias de control

La sentencia if-else del lenguaje C

if(expr) sentencia1 else sentencia2

//resultado del compilador

```
        subcc      %r1,%r2,%r0
        bne Over
        !código a ejecutar para la sentencia 1 verdadera
        ba End
        Over
        End
```

Compiladores: la sentencia while

while(%r1 != %r2) %r1 = %r1+1

//resultado del compilador

```
        ba TEST
TRUE:   add %r1,1,%r1
TEST:   subcc %r1,%r2,%r0
        bne TRUE
```

¿Que harian con la sentencia do-while?¿y con la for?

Ensamblado

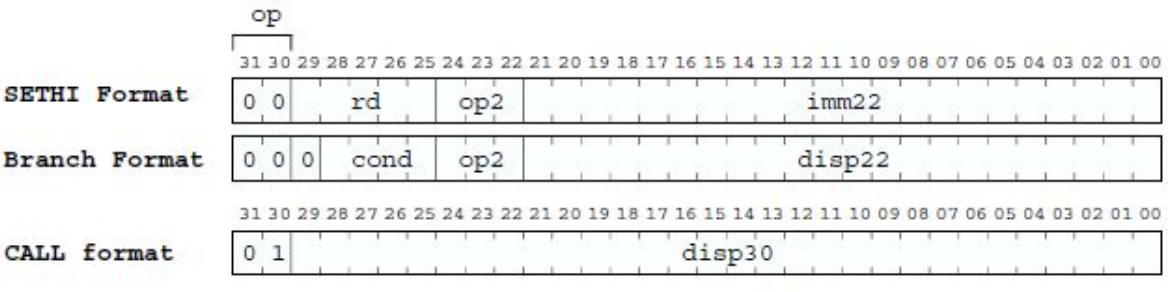
Prestaciones que debe cumplir un ensamblador:

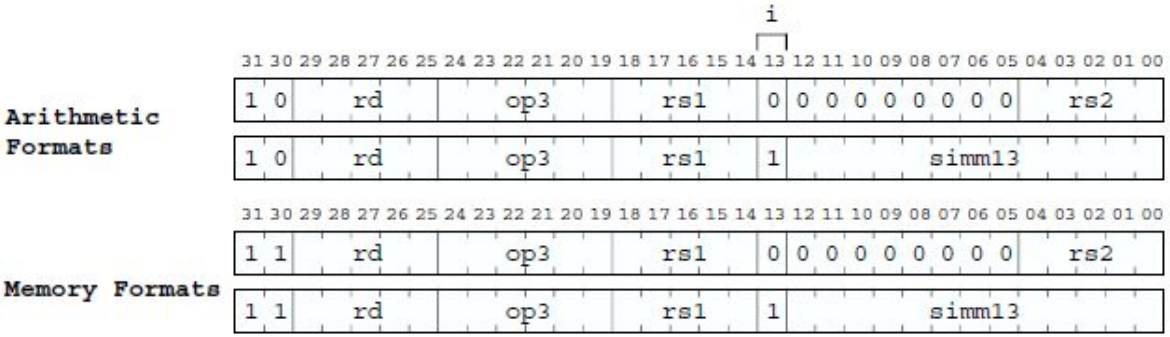
- Permitir que el programador especifique la ubicación de las variables y programas al momento de la ejecución.
- Ofrecer al programador inicializar los valores de los datos en memoria antes de la ejecución del programa
- Proveer expresiones nemotécnicas para todas las instrucciones del lenguaje máquina y modos de direccionamiento
- Permitir el uso de rótulos simbólicos para identificar o representar direcciones y constantes.
- Ofrecer al programador la posibilidad de especificar la dirección del inicio de un programa

- Ofrecer cierto grado de cálculo al momento del ensamblado.
- Incluir un mecanismo que permita en lenguaje simbólico definir variables en un módulo y utilizarlas en otro módulo distinto
- Proveer la expansión de macrorutinas

Ensamblador: 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
```





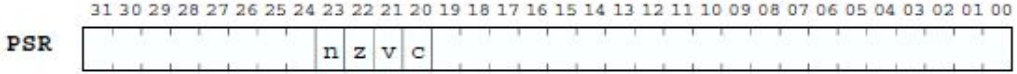
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

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

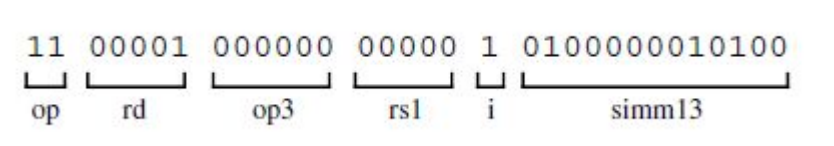
op3 (op=11)	
000000	ld
000100	st

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



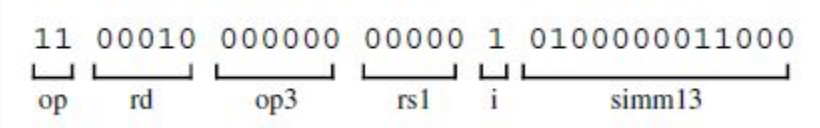
El ensamblador recorre al programa escrito con un contador de posición, el cual similar a como opera el PC va incrementandose de cuatro en cuatro dado que las instrucciones son de 32 bits.
Dicho contador es inicializado en cero y el .org le asigna lugo un valor específico.

ld [X],%r1



resultado de la segunda pasada del ensamblador para la primer y segunda linea que va representada en memoria.

ld [Y],%r2



ambas difieren en el campo de simm13, dado que las direccoines de X e Y son distintas

```
ld [x], %r1          1100 0010 0000 0000 0010 1000 0001 0100
ld [y], %r2          1100 0100 0000 0000 0010 1000 0001 1000
addcc %r1,%r2,%r3    1000 0110 1000 0000 0100 0000 0000 0010
st %r3, [z]          1100 0110 0010 0000 0010 1000 0001 1100
jmpl %r15+4, %r0     1000 0001 1100 0011 1110 0000 0000 0100
15                   0000 0000 0000 0000 0000 0000 0000 1111
9                    0000 0000 0000 0000 0000 0000 0000 1001
0                    0000 0000 0000 0000 0000 0000 0000 0000
```

Ensamblador: rótulo definidos luego de ser utilizado

```
                .
                .
                .
              call   sub_r       ! Subroutine is invoked here
                .
                .
                .
sub_r:        st     %r1, [w]    ! Subroutine is defined here
                .
                .
                .
```

```
! 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:     jmpl  %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
```

Symbol	Value
a_start	3000
length	—

(a)

Symbol	Value
a_start	3000
length	2092
address	2096
loop	2060
done	2088
a	3000

(b)

Tareas finales para el ensamblador

- Nombre y tamaño del módulo; datos, códigos, pilas, etc.
- Dirección del símbolo de comienzo.
- Información acerca de símbolos y globales y externos
- Información acerca de rutinas de biblioteca.
- Información de reubicación: el ensamblador puede especificar cuales direcciones pueden reubicarse y cuales no.

Ubicación de programas en memoria

Es responsabilidad del ensamblador indicar que símbolos son los reubicables y cuáles no. En la mayoría de los casos podemos suponer que como programadores no nos interesa en que posiciones de memoria se está corriendo nuestro código.

Para esta tarea el ensamblador utiliza un **diccionario de reubicación** el cual queda en el módulo ensamblado. Luego este diccionario será utilizado por el *enlazador (linker)* o/y el *cargador (loader)*.

Enlace y carga

El enlazador combina módulos que fueron ensamblados por separado llamados **módulos objeto** en un único programa o **módulo de carga.**

El trabajo del linker es resolver todas las referencias globales y externas y reubicar las direcciones de los diferentes módulos.

El **módulo de carga** puede ser cargado a memoria por medio de un **cargador (loader)** el cual también puede necesitar modificar la direcciones si el programa se carga en una dirección distinta a la utilizada por el *linker*.

Enlazador: resolución de referencias externas

La directiva ***.global***: le indica al ensamblador que debe señalizar al símbolo como disponible para otros módulos objeto durante la fase de enlace.

La directiva ***.extern:*** identifica un rótulo utilizado en el módulo actual pero que fue definido en otro.

No tiene sentido definir como global a un .equ dado que .equ ya se resuelve en tiempo de ensamblado.

Los rótulos locales no se utilizan luego del tiempo de ensamblado por lo cual dos subrutinas distintas pueden tener el mismo nombre en dos módulos distintos. (por ejemplo un rótulo o etiqueta llamada *loop*).

Enlazador: ejemplo

El ensamblador incluye en cada módulo un encabezado con información acerca de los símbolos que son globales o externos para que puedan ser resueltos por el **enlazador.**

```
! Main program
      .begin
      .org  2048
      .extern sub
main: ld    [x], %r2
      ld    [y], %r3
      call  sub
      jmpl  %r15 + 4, %r0
   x: 105
   y:  92
      .end
```

```
! Subroutine library
      .begin
ONE   .equ     1
      .org   2048
      .global sub
sub:  orncc  %r3, %r0, %r3
      addcc  %r3, ONE, %r3
      jmpl   %r15 + 4, %r0
      .end
```

Enlazador: tabla de símbolos

Symbol	Value	Global/ External	Reloc- atable
sub	–	External	–
main	2048	No	Yes
x	2064	No	Yes
y	2068	No	Yes

Main Program

Symbol	Value	Global/ External	Reloc- atable
ONE	1	No	No
sub	2048	Global	Yes

Subroutine Library

Enlazador: reubicación de módulos

El ensamblador tiene la responsabilidad de determinar cuales rótulos son reubicables cuando se construye la tabla de símbolos.

Cada módulo objeto tiene su tabla de símbolos, no tiene sentido determinar cómo reubicable a un rótulo externo (caso de sub en el módulo main).

Las constantes definidas con .equ ya fueron utilizadas durante el proceso de ensamblado.

Las posiciones de memoria determinadas por una posición relativa a un .org tales como x e y suelen ser reubicables.

Toda la información requerida para la reubicación de un módulo se encuentra en el diccionario de reubicación contenido en el archivo ensamblado disponible para el linker

Carga (*loader*): no repite el trabajo del linker

Es un programa que ubica al módulo de carga en la memoria principal. Al haber pasado ya por el linker tenemos un único módulo a ubicar en memoria. Además el loader debe inicializar ciertos valores tales como el *stack pointer* y el *program counter*.

El ensamblador o en linker no va a poder reconocer en qué dirección de memoria va a residir nuestro programa dado que siquiera puede saber cuántos programas van a convivir en memoria en el momento de carga. Su tarea es ubicar al programa de forma que varios puedan coexistir en memoria de forma simultánea.

¿Cuantos bytes creen que ocupa un booleano en lenguaje C? ¿Por qué?
¿Cuántos bits son suficientes para representar este tipo de datos?