BI-11736
DON
MFN-4059
tesis

CINVESTAV-IPN
Biblioteca de Ingeniería Eléctrica

FB0000009692

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

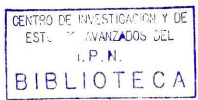
CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL

INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION DE COMPUTACION

ESTUDIO COMPARATIVO DE ARQUITECTURAS DE PROCESADORES CON BASE EN LA GENERACION DE CODIGO



Tesis que presenta el Ing. Octavio Héctor Juarez Espinosa para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA con opcion en COMPUTACION.

Trabajo dirigido por el Dr. Jan Janecek Hyan.

México D.F.
junio de 1990

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I.P.N.
BIBLIOTECA
INGENIERIA ELECTRICA

Becario de CONACYT

XM
CLASIF.: 90.8
ADQUIS.: 61-11736
FECHA: 24-VIII-90
PROCED.: DON
$

A LA MEMORIA DE MI PADRE

A MI MADRE Y HERMANOS

A EVA

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

AGRADECIMIENTOS

Doy las gracias:

Al doctor Jan Janecek por sus enseñanzas, observaciones y críticas para la realización de este trabajo.

Al doctor Guillermo Morales y al ing. Rodolfo Rosado por el equipo facilitado para trabajar.

Al maestro Andrés Vega por la lectura y corrección de este texto.

A los maestros, compañeros y personal de la SECCION DE COMPUTACION por sus enseñanzas, ayuda y solidaridad.

Al CONACYT y al CINVESTAV por la ayuda brindada para la obtención del grado.

INDICE

PAGINA

0.- INTRODUCCION 1

1.- DESCRIPCION DE ARQUITECTURAS 4

1.1.- ARQUITECTURAS CON UN NUMERO LIMITADO DE REGISTROS 5

1.2.- ARQUITECTURAS CON UN NUMERO ABUNDANTE DE REGISTROS 9

1.3.- MAQUINAS DE PILA 18

1.4.- COMENTARIOS FINALES 25

2.- EL TRANSPUTER Y OCCAM 26

2.1.- MODOS DE DIRECCIONAMIENTO 27

2.2.- DISEÑO DEL CONJUNTO DE INSTRUCCIONES 28

2.3.- EL LENGUAJE OCCAM 31

3.- SMALL C 38

3.1.- SOBRE SMALL C 38

3.2.- SINTAXIS DEL LENGUAJE 38

3.3.- DIFERENCIAS CON C 41

3.4.- LA IMPLEMENTACION DE SMALL C 44

4.- GENERACION DE CODIGO PARA EL TRANSPUTER 52

4.1.- EL TRANSPUTER Y EL LENGUAJE OCCAM 52

4.2.- EL FORMATO DE SALIDA 53

4.3.- CODIGOS QUE SE PUEDEN UTILIZAR DENTRO DE LA CONSTRUCCION GUY 55

4.4.- ALGUNOS CODIGOS GENERADOS 57

4.5.- LA COMPILACION DE EXPRESIONES 61

4.6.- LOS PSEUDOCODIGOS 68

4.7.- PROBLEMAS SURGIDOS AL IMPLEMENTAR LOS CAMBIOS 69

4.8.- ESTRUCTURAS DE DATOS UTILIZADAS 76

4.9.- EL ESPACIO DE TRABAJO DE UN PROCEDIMIENTO 78
4.10.- LAS CADENAS DE OPTIMIZACION 79
5.- RESULTADOS 83
6.- CONCLUSIONES 88
REFERENCIAS 90
APENDICE A (SINTAXIS DE OCCAM) 93
APENDICE B (CODIGO GENERADO) 101

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

INTRODUCCION

El presente trabajo es un estudio comparativo de arquitecturas basado en el código generado para un compilador conocido.

En el diseño de arquitecturas se puede apreciar una modificación de conceptos. La evolución de la tecnología, permite tener procesadores cada vez más poderosos.

El progreso de arquitecturas se debe al deseo de obtener mayor densidad de código, que repercuta en una mayor velocidad de ejecución, lo que aparentemente se traduce a un alto precio en el mercado y a procesadores complejos.

Uno de los principales criterios de diseño de arquitecturas es sin duda la escritura de compiladores.

Algunos diseñadores de arquitecturas prefieren incluir en el conjunto de instrucciones, algunas que soportan construcciones de lenguajes de alto nivel, mientras que otros prefieren un conjunto muy reducido de instrucciones con su meta principal fijada en la velocidad de ejecución.

A mayor sintonía existente entre una arquitectura de una máquina y un compilador, el trabajo que se realiza es mucho más efectivo.

La sintonía entre arquitecturas y compiladores influye en la escritura de los algoritmos de compilación y de optimización de un lenguaje en particular.

Algunos de los problemas, relativos a la arquitectura, que surgen al escribir un compilador son:

--Las máquinas de muchos registros requieren que el compilador asigne los registros, lo que se complica al tratar de hacer óptima la asignación.

--En las máquinas que utilizan ventanas de registros se debe controlar cuando el número de variables locales rebasa el número de registros asignados a ellas, así como el salvado de ventanas de llamada cuando el nivel de anidamiento en una llamada es mayor al número de ventanas de llamada disponibles.

--Cuando se tiene una arquitectura con paralelismo en las fases de ciclos de instrucción, se tienen que introducir retardos cuando hay dependencia entre los datos de dos instrucciones.

-- Si el número de modos de direccionamiento y el número de instrucciones es muy abundante, el problema al escribir el compilador es el proceso de seleccionar la instrucción y el modo de direccionamiento adecuado.

En este contexto una de las principales metas de este trabajo es visualizar de una manera práctica las posibilidades que tiene la arquitectura del TRANSPUTER de INMOS en el contexto de una arquitectura muy popular como es el 8086 de INTEL y de algunas arquitecturas actuales con diferente filosofía de diseño.

Aunque el TRANSPUTER es un chip diseñado para ambientes de programación paralelo, su comparación en este trabajo es en el nivel secuencial.

El compilador utilizado para hacer la comparación es SMALL C y las arquitecturas en las que se hace la comparación práctica son: 8086 de INTEL y TRANSPUTER de INMOS.

El interés por trabajar con el TRANSPUTER se debe a que es una arquitectura fuera del diseño convencional de un procesador.

Este trabajo está compuesto de seis capítulos. En el primero de ellos se describen las arquitecturas objeto de esta comparación y las arquitecturas que constituyen el contexto.

En el segundo capítulo se detalla el conjunto de instrucciones del TRANSPUTER y se describe el lenguaje OCCAM.

En el tercer capítulo se describe la implementación del compilador de SMALL C .

En el capítulo cuatro se describen los cambios realizados al compilador de SMALL C para generar código para el TRANPUTER.

Los resultados de la comparación de códigos generados y de velocidades de ejecución se presentan en el capítulo cinco, para culminar con el capítulo seis de conclusiones.

CAPITULO 1

DESCRIPCION DE ARQUITECTURAS

Se entiende por arquitectura en este trabajo, lo que un programador en lenguaje de máquina ve: registros, tipos de datos, formatos y conjunto de instrucciones.

En este capítulo se categorizan las arquitecturas con base en sus componentes y se hace una descripción de cinco de ellas.

La categorización es la siguiente:

* **MAQUINAS CON NUMERO LIMITADO DE REGISTROS.**

  Dentro de esta categoría se incluyen las máquinas que trabajan con acumuladores y es aquí donde se incluye el 8086 de INTEL.

* **MAQUINAS CON ABUNDANTE NUMERO DE REGISTROS**

  Arquitecturas tales como RISC y el Motorola 68000 se incluyen en esta clase.

* **MAQUINAS DE PILA**

  En esta categoría se hacen dos sub-categorías:

  MAQUINAS DE PILA LIMITADA

  En esta categoría se ubica el TRANSPUTER de INMOS.

  MAQUINAS DE PILA NO LIMITADA

  La máquina representativa de esta clase es la HP3000.

La categorización se hizo con base en la característica distintiva de cada arquitectura.

En el diseño de cada arquitectura se puede encontrar una filosofía y así se pueden definir dos grupos:

a).- El grupo de arquitecturas con un conjunto de **instrucciones de alto nivel.** El término instrucciones de alto nivel significa: una arquitectura con un conjunto de

instrucciones más poderoso y capaz de soportar construcciones de lenguajes de alto nivel.

b).- El grupo de las arquitecturas con un conjunto reducido de instrucciones, que buscan aumentar la velocidad de ejecución.

A continuación se hace una descripción de cada una de estas arquitecturas.

1.1.- ARQUITECTURAS CON UN CONJUNTO LIMITADO DE REGISTROS

La computadora diseñada por Von Neumann cuenta solamente con un registro interno, acumulador, el que sirve para todas ejecutar todas las operaciones.

El contar con un registro para todas las operaciones, se puede ver como el extremo opuesto a las máquinas con muchos registros.

Actualmente existen máquinas que conservan la estructura de la máquina de Von Neumann, tales arquitecturas cuentan con un acumulador y algunos registros auxiliares. Algunos ejemplos de esas máquinas son: 8080, 8085 y 8086. El 8086 de Intel se describe a continuación.

1.1.1 EL 8086 DE INTEL.

El microprocesador 8086/88 a diferencia de los microprocesadores convencionales, está dividido en dos unidades que pueden operar independientemente, realizando cada una sus funciones en paralelo con la otra, estas unidades son la unidad de ejecución (EU) y la unidad de interfaz con el bus (BIU).

EL BIU está involucrado con la búsqueda de instrucciones (ciclos fetch), contiene una cola de instrucciones de 6 bytes, la cual actúa como memoria temporal con la característica FIFO, la cual se llena continuamente mientras la EU no solicite acceso

al bus, de esta cola la EU va tomando las instrucciones.

Tanto el 8086 como el 8088 tienen un bus de direccionamiento de 20 bits de amplitud, lo que lo provee de la capacidad de direccionar un megabyte de memoria.

Sin embargo, el registro de direccionamiento tiene una amplitud de 16 bits. Esto equivale a 64K bytes. Este procesador usa un método llamado segmentación para direccionar el megabyte de memoria.

1.1.1.1.-JUEGO DE REGISTROS.

El 8086/8088 contiene 14 registros de 16 bits. Algunos pertenecen a la EU (Unidad de ejecución) y otros a la BIU (Unidad de Interfaz con el BUS).

La EU tiene los siguientes registros:

* Cuatro registros generales de 16 bits (AX,BX,CX,DX) que pueden dividirse en 8 registros de 8 bits (AH, AL, BH, BL, CH, CL, DH, DL). En este caso A representa el acumulador; B el registro de base; C un contador y D el registro de datos.
* Cuatro registros apuntador y de índice (SP, BP, SI, DI), los cuales no pueden subdividirse. SP es el apuntador a la pila; BP es el apuntador base; SI y DI que son los registros índice fuente y destino.
* Un registro de banderas de procesador. Estos incluyen: indicador de cero (ZF), indicador de paridad (PF), un indicador de signo (SF), indicador de acarreo (CF), indicador auxiliar (AF), indicador de dirección (DF), indicador de interrupción (IF), indicador de sobreflujo (OF) e indicador de desvío (TF).

La BIU tiene los siguientes registros:

* Cuatro registros de segmento (CS, DS, SS y ES). Sus códigos representan a los registros de segmento de código, datos, pila y extra respectivamente.

2 Un apuntador a instrucción.

1.1.1.2.-MODOS DE DIRECCIONAMIENTO.

Se entiende como modo de direccionamiento, a la forma como un operando es específicado.

El 8086 tiene 7 modos de direccionamiento básicos:

Inmediato.
Directo.
Registro.
Registo Indirecto.
Registro relativo.
Basado Indexado.
Relativo Basado Indexado.

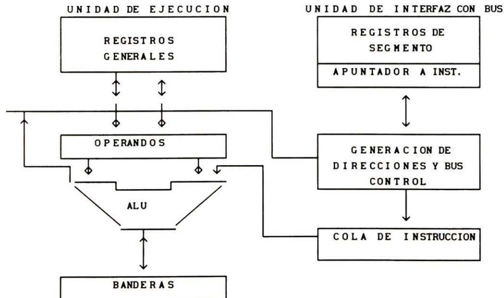


FIGURA 1.1. EL procesador 8086

* **El modo inmediato.** El dato es de 8 ó 16 bits de longitud y es parte de la instrucción.

```
MOV AL,0DH
MOV BX,02FCH
```

* **El modo directo.** La dirección efectiva es parte de la instrucción.

```
MOV varw,AX
```

* **Registro.** El dato está en el registro especificado por la instrucción.

```
MOV AX,DX
ADD AX,DX
```

* **Registro Indirecto.** La dirección efectiva del dato está en el registro base BX o un registro índice que es especificado en la instrucción.

```
MOV AX,[DI]
MOV [BX],AL
```

* **Registro Relativo.** La dirección efectiva es la suma del desplazamiento de 8 ó 16 bits y el contenido de un registro base o un registro índice.

```
MOV AX,[BX +9H]
```

* **Basado Indexado.** La dirección efectiva es la suma de un registro base y un registro índice.

```
ADD [BX +SI],AX
```

* **Relativo Basado Indexado.** La dirección efectiva es la suma de un desplazamiento de 8 ó 16 bits y una dirección basada indexada.

```
MOV AX,[BX + SI + 7]
```

1.1.1.3.-EL CONJUNTO DE INSTRUCCIONES.

En general las instrucciones de 8086 se pueden clasificar en los siguientes grupos:

* Transferencia de datos.
* Aritmética entera binaria.
* Operaciones lógicas.
* Gestión de bits.
* Aritmética codificada en binario.
* Gestión de cadenas.
* Control del programa.
* Control del sistema.

1.2.- ARQUITECTURAS CON UN NUMERO ABUNDANTE DE REGISTROS

Dentro de esta categoría se incluyen dos arquitecturas con un gran número de registros: M68000 y RISCII.

El M68000 con 16 registros y la RISCII (SPARC comercialmente) con 192 registros.

Este tipo de máquinas evitan hacer muchos accesos a memoria ya que las variables temporales se pueden mantener en registros en lugar de estar en memoria, sin embargo al hacer una llamada a un procedimiento hay que salvar un conjunto muy grande de registros.

Existen tres formas de administrar los registros para escribir un compilador :

* Asignarles una tareas específica a cada uno de los registros. (Cuando el número no es grande).

* Asignar los registros en tiempo de compilación de una manera óptima. La complicación se encuentra al implementar los algoritmos de compilación, pues se deben utilizar algoritmos de programación dinámica o de coloreo.

* Hacer ventanas de registros de longitud fija para cada marco de llamada de procedimientos o funciones en lenguajes estructurados. En este tipo de máquinas el problema de asignar registros en tiempo de compilación desaparece y el acceso a parámetros y variables locales es relativamente veloz.

El problema de las ventanas de registros, es que cuando el número de llamadas a procedimientos llena la pila de registros se debe salvar el marco de la ventana del procedimiento con más tiempo en la pila.

Existe una diferencia entre las dos arquitecturas que se describen en esta sección y es en la forma de asignar registros.

1.2.1.-MOTOROLA 68000.

El procesador Motorola 68000 permite un mayor espacio direccionable, modos de direccionamiento flexibles, un número razonablemente grande de registros del CPU y un conjunto de instrucciones que facilitan las construcciones de los lenguajes de alto nivel.

Usa direcciones de 24 bits, resultando un espacio direccionable de 16 Megabytes.

Los datos externos son manejados en palabras de 16 bits.

Los registros son de 32 bits de longitud y son 8 para direcciones y 8 para datos.

Los registros de datos son de propósito general y son acumuladores y contadores.

El registro siete de direcciones es el apuntador a la pila.

Tiene un registro de estado y un bit que le permite seleccionar entre modo de usuario y modo supervisor.

El contador de programa tiene 32 bits y usa 24 para direccionar.

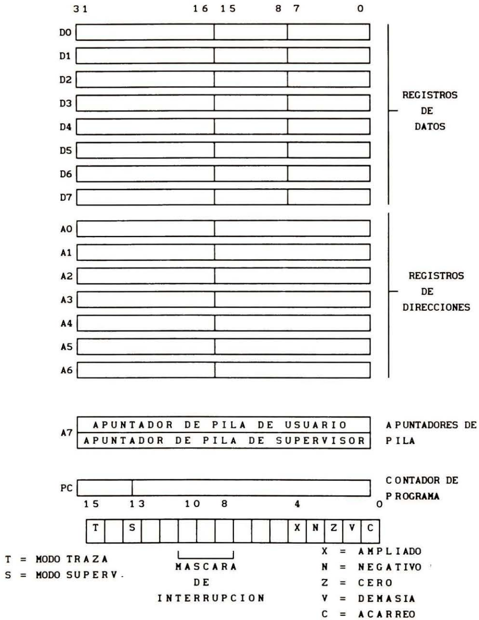


FIGURA 1.2.- REGISTROS DEL 68000.

Los modos de operación se dividen en:

SUPERVISOR. Todas las instrucciones pueden ser usadas.

USUARIO. Ciertas instrucciones pueden ser usadas.

1.2.1.1.- MODOS DE DIRECCIONAMIENTO.

* **Modo Inmediato.** El operando es contenido en la instrucción. Las medidas de los operandos son: byte, word, longword y números muy pequeños.
* **Modo Absoluto.** La dirección absoluta de un operando se da en la instrucción. (24 ó 16 bits)
* **Modo registro.** El operando está en un registro.
* **Modo indirecto de registro.** La dirección efectiva del operando es el contenido de un registro de dirección especificado por la instrucción.
* **Modo de Autoincremento.** La dirección efectiva del operando está en un registro de dirección. Después de accesar el operando, los contenidos del registro se incrementan.
* **Modo de autodecremento.** Se decrementa el registro A_n y la dirección efectiva del operando es el nuevo contenido de A_n.
* **Modo indexado básico.** En la instrucción se especifica un desplazamiento de 16 bits y un registro A_n. La dirección efectiva está dada por: EA = desplazamiento + A_n.
* **Modo complemento indexado.** En la instrucción se da: un desplazamiento de 8 bits,A_n y un registro índice R_k (A_n ó D_n) y la dirección efectiva se obtiene así: EA = desplazamiento + A_n + R_k.
* **Modo Relativo Básico.** Es igual que el modo indexado básico, pero PC = A_n.
* **Modo completo relativo.** Es igual que el modo completo índexado pero con PC = A_n.

1.2.1.2.- CONJUNTO DE INSTRUCCIONES.

El M68000 proporciona un conjunto grande de instrucciones

que se pueden operar con todas las medidas de datos.

Se pueden utilizar todos los modos de direccionamiento con todas las instrucciones.

La mayoría de las instrucciones requiere que un operando sea especificado en un registro y el otro con cualquier modo de direccionamiento.

Pocas instrucciones tienen dos operandos en memoria principal por ejemplo: MOVE, CMPM y ABCD.

Si no se especifica la medida, se usa Word por default.

1.2.2.- COMPUTADORAS RISC.

El término RISC es la abreviatura de "Computadoras con conjunto reducido de instrucciones".

El diseño de arquitecturas RISC apunta en dos direcciones: simplificar el hardware por un lado y aumentar la sinergía entre compiladores y arquitecturas.

Los argumentos de quienes diseñan computadoras con conjuntos de instrucciones numeroso y complicado son:

1).-Conjuntos ricos de instrucciones simplifican compiladores.
Construir compiladores para máquinas con muchos registros, es un trabajo difícil.
Compiladores para arquitecturas con modelos basados en pilas u operaciones en memoria son mucho más simples y confiables.

2).- Conjuntos ricos de instrucciones podrían aliviar la crisis del software.
La idea de crear instrucciones de máquina parecidos a los estatutos en un lenguaje de programación, para cerrar la brecha semántica entre los lenguajes de programación y los lenguajes de máquina.

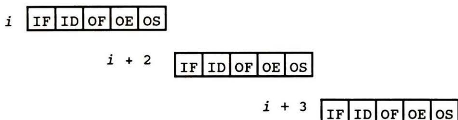
3).- Conjuntos ricos de instrucciones podrían mejorar la calidad de la arquitectura.

1.2.2.1.- ARQUITECTURAS RISC.

Estas máquinas tienen un conjunto reducido de instrucciones y se ejecutan generalmente en un ciclo.

Los principios de diseño de RISC son:

* Las microinstrucciones no son más rápidas que las instrucciones.



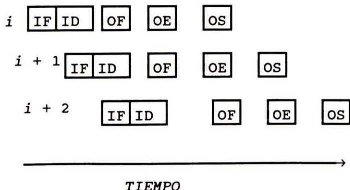


FIGURA 1.3. Ejecución secuencial y ejecución paralelo.

* El microcódigo no es mágico. Mover el software a microcódigo no es mejor, es más duro. Las mismas primitivas de hardware asumidas por las microinstrucciones deben estar disponibles en lengauje ensamblador.
* Decodificación simple y ejecución "pipelined" (paralela), son más importantes que la medida del programa.
  Por "pipelined" se entiende particionar un trabajo en piezas, para que partes de instrucciones diferentes se ejecuten al mismo tiempo.
  El paralelismo de ejecución de instrucciones, hace que la diferencia de velocidades con las máquinas que no tienen estas características sea bastante grande como se puede ver en la figura 1.3.
* La tecnología de compiladores es usada para simplificar instrucciones, antes que para generar instrucciones complejas.

1.2.2.2.-CARACTERISTICAS DE LAS MAQUINAS RISC.

* Las operaciones son de registro a registro, con solo LOAD y STORE para accesar memoria.
* Las operaciones y los modos de direccionamiento son reducidos. Las operaciones entre registros se completan en un ciclo, permitiendo una unidad de control más simple. Instrucciones de ciclos múltiples de reloj, tales como operaciones de punto flotante, son ejecutadas por software o por un co-procesador .
  Solamente se cuenta con dos modos de direccionamiento: indexado y relativo al contador de programa.
* Los formatos de instrucción son simples. Esto permite a RISC aumentar la velocidad de decodificación.
* Las ramificaciones de RISC evitan los castigos del "pipeline".

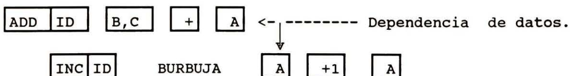
Una instrucción de ramificación en una computadora con paralelismo de ejecución, normalmente retarda el "pipeline" hasta que la instrucción con la dirección de la ramificación es leída.

Surge un problema cuando existen dependencias de datos y/o direcciones entre instrucciones. En la figura 1.4. se observan ejemplos de períodos de latencia por dependencias entre instrucciones.

1.2.2.3.- RISC I Y RISC II.

Los primeros sistemas identificados como RISC, fueron los construidos en BERKELEY. Se desarrollaron dos modelos RISC I y RISC II. Ambos tienen básicamente la misma arquitectura: RISC I tiene 31 instrucciones, mientras RISC II tiene 39. Ambas son máquinas de 32 bits, con un CPU con arreglo de registros: 78 registros en RISC I y 138 registros en RISC II. Esto no quiere decir que los 138 registros están disponibles para ser usadas amigablemente. En realidad, cada procedimiento creado tiene solamente treinta y dos registros a su disposición.

PIPELINED CON DEPENDENCIA DE DATOS



PIPELINED CON DEPENDENCIA DE DIRECCIONES.

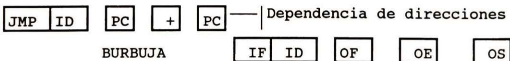


FIGURA 1.4.DEPENDENCIAS DE RAMIFICACIONES Y DATOS ENTRE INSTRUCCIONES FORZA RETARDOS,O BURBUJAS EN PIPELINES.

Los registros disponibles para cada procedimiento en la RISC de Berkeley se dividen en dos partes:

REGISTROS GLOBALES. R0, R1,..., R9: un total de diez. Esos registros pueden ser usados por todos los procedimientos del mismo programa corriendo en la RISC.

VENTANA DE REGISTROS. R10, R11,..., R31: un total de 22. Estos registros son únicos a un procedimiento en particular y ellos podrían corresponder a diferentes conjuntos de registros actuales para un diferente procedimiento. Hay sin embargo una relación entre la ventana de registros de un procedimiento. Al llamar un procedimiento a otro se pasan los parámetros como se indica en la figura 1.5.

La principal ventaja del arreglo de registros es el ahorro de tiempo al pasar los parámetros en una llamada a un procedimiento. En RISC los parámetros quedan en los registros y no hay muchos movimientos de datos de memoria a registros.

Existe otro diseño de RISC en la Universidad de Stanford y la principal diferencia con RISC I y II, es que no contiene un

FIGURA 1.5. REGISTROS VISTOS POR CADA PROCEDIMIENTO.

gran arreglo de registros y no usa el principio de la ventana de registros.

La máquina original tiene 16 registros de 32 bits.

Otra diferencia fundamental entre la RISC y la MIPS es la forma de manejar las dependencias del pipeline.

La RISC de Berkeley maneja este problema desde hardware al retrasar la instrucción *i* hasta que se termina la instrucción *i* - 1.

En MIPS el manejo del problema se hace por software.

1.3.- MAQUINAS DE PILA.

Las máquinas con esta característica realizan todas las operaciones aritméticas, lógicas y relacionales en la pila.

Las dos máquinas que se describen en este estudio difieren en el límite de la pila.

La HP3000 posee una pila de tamaño ilimitado, la pila es de llamada y de evaluación. Durante el algoritmo de compilación no existe el problema de la asignación de registros, ni el de sobreflujo de la pila.

El Transputer difiere de la HP3000, en que su pila es solo para evaluar expresiones y está formada por tres registros.

La pila de llamada en el Transputer se maneja en la memoria apuntada por el registro "Espacio de Trabajo".

A continuación se describen de una forma más completa estas arquitecturas.

1.3.1.-EL TRANSPUTER UNA MAQUINA DE PILA LIMITADA.

Es un chip VLSI con un procesador, memoria para almacenar los programas ejecutados por el procesador y enlaces de comunicación para conexión directa con otros transputadores.

La razón por la que transputer incluye memoria interna, es por que la mayoría de las operaciones que realiza un procesador requieren acceso a memoria y es más rápido accesar una memoria dentro del chip que accesar memoria externa.

El transputer habilita concurrencia en aplicaciones tales como simulación, control de robots, síntesis de imágenes y procesamiento digital de señales. Las aplicaciones intensivamente númericas pueden explotar grandes arreglos de transputadores.

La velocidad de ejecución depende del número de transputadores, la velocidad de intercomunicación y la ejecución de tipo flotante.

Específicamente el chip IMS T800 está integrado por un procesador de 32 bits, una unidad de punto flotante de 64 bits, 4K de memoria RAM, 4 enlaces de comunicación, una interfaz para memoria y una interfaz para periféricos.

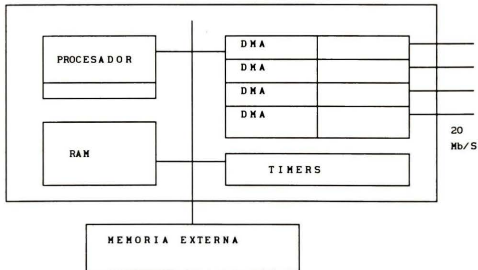


FIGURA 1.6. TRANSPUTER T800

El procesador de 32 bits provee una velocidad de 10 MIPS y la unidad de punto flotante una velocidad de 1.5 Mega Flops.

Específicamente del chip, para este trabajo, nos interesa la arquitectura del procesador.

1.3.1.1.-LOS REGISTROS DEL TRANSPUTADOR.

* Apuntador a instrucción de 32 bits.

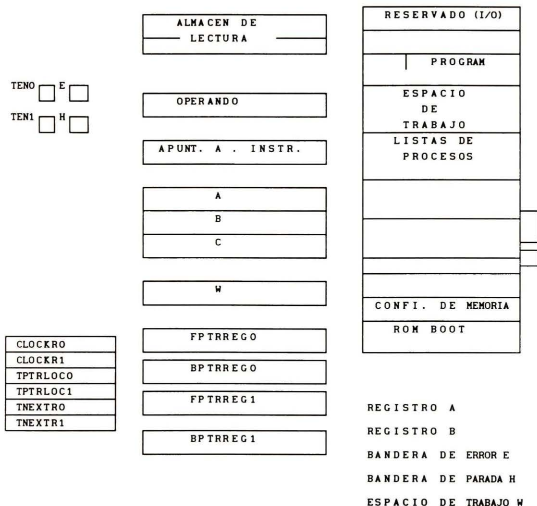


FIGURA 1. 7.PROCESADOR DEL TRANSPUTER

* Registros para el almacen de instrucciones leídas.

* Registro de operando.

* Registro de espacio de trabajo. Este registro de 32 bits puede apuntar a cualquier lugar en memoria y permite el cambio de contexto fácil y rápido.

* Una pila de evaluación de tres registros.

* Dos registros para manejar la cola de procesos de alta prioridad.

* Dos registros para manejar la cola de procesos de baja prioridad.

* Para las funciones de tiempo existen seis registros:

  # Dos relojes, uno para cada nivel de prioridad.

  # Dos registros que apuntan al siguiente elemento en las dos colas de prioridad.

  # Dos registros para indicar el tiempo del primer evento a ocurrir, uno para cada nivel de prioridad.

* Dos bits que indican si las colas de timer no están vacías.

1.3.1.2.- LOS CANALES DE COMUNICACIÓN.

Dos procesos se comunican por medio de canales. Las palabras de memoria sirven para implementar un canal de comunicación entre procesos al interior del transputer.

Para comunicación externa se utilizan cuatro canales, para lo que se reserva memoria en la parte más baja.

La implementación de comunicación externa usa tres registros invisibles para soportar el DMA autónomo, el cual libera al procesador para otro trabajo. Estos registros contienen:

--El contador del número de bytes transferidos.

--Un apuntador a la localidad de memoria (para entrada o salida).

--Un apuntador al espacio de trabajo de procesos.

1.3.2. HP3000 UNA MAQUINA DE PILA NO LIMITADA.

La HP3000 es una minicomputadora de 16 bits que tiene algunas características que es más común encontrar en máquinas grandes.

La memoria principal contiene los datos e instrucciones en dominios separados. No es posible mezclar instrucciones, excepto en el caso de datos inmediatos. Se utilizan registros de hardware como apuntadores al programa y a los segmentos de datos como se ve en la figura 1.8.

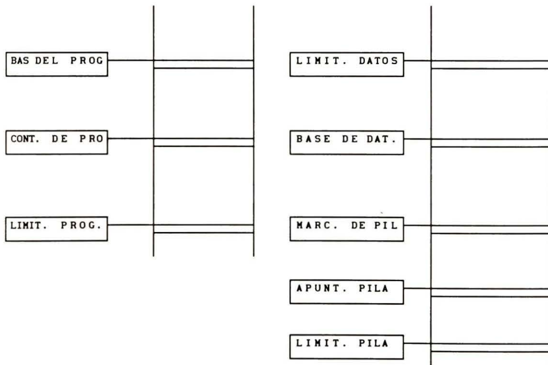


FIGURA 1.8.ORGANIZACION DE PROGRAMA Y DATOS EN HP3000

Tres registros definen el segmento del programa. Los registros de base de programa "PB" y de límite de programa indican el área de memoria que es ocupada por el programa. El contador del programa tiene la función de apuntar a la instrucción en ejecución. Cada registro contiene la dirección de 16 bits.

El segmento de datos se divide en dos partes: la pila y elárea de datos. Se utilizan cinco apuntadores de 16 bits para delinear y efectuar el acceso a estas localidades de memoria.

El contenido del registro de base de datos, "DB" denota la localidad inicial de la pila. La pila crece en el sentido de la dirección superior.

El tope de la pila es apuntado por el apuntador a la pila "SP".

El límite superior de la pila se especifica por el registro de límite de pila "PL". Como se puede ver la pila si tiene un tope, pero en la clasificación se nombra como pila no limitada, para denotar que la pila no tiene un tope de tres como en la arquitectura anterior.

El apuntador "Q" o marcador de pila se emplea para denotar el punto inicial de los datos de una rutina o procedimiento.

En el registro de llamada se guardan cuatro palabras: registro índice, dirección de regreso, registro de condición y distancia al registro de llamada anterior.

Aunque contiene más registros, solo los mencionados resultan visibles al programador.

1.3.2.1.- INSTRUCCIONES DE PILA DE LA HP3000.

La estrategia de las máquinas de pila es efectuar las operaciones con los operandos en el tope de la pila, dejando los resultados en la misma.

Existen 13 clases de instrucciones. En lugar de una descripción completa del conjunto de instrucciones, solo se habla de algunas de ellas.

Su formato es de 16 bits.

Estan primero las de dirección de memoria y su formato es:

15 — 11	10	9	9 — 0
CODIGO OPERACION	X	I	MODOS DE DESPLAZAMIENTO

Algunas instrucciones de esta clase son:

LOAD Para meter una palabra en la pila.

STOR Sacar una palabra de la pila y llevarla a memoria.

ADDM Agregar una palabra específica de la memoria al tope de la pila.

MPYM Multiplicar una palabra de memoria por el tope de la pila.

Otra clase de instrucciones, es la de instrucciones de pila cuyo formato es:

15 — 12	11 — 6	5 — 0
CODIGO OPERACIO	CODIGO PILA A	CODIGO PILA B

Estas instrucciones tienen cuatro ceros en las posiciones de orden superior.Los doce bits restantes se dividen en dos campos de 6 bits cada uno y se puede especificar una operación distinta en cada campo.

Algunas instrucciones de este tipo son:

ADD Suma el contenido de las dos palabras superiores de la pila.

CMP Compara el contenido de las dos palabras superiores de la pila.

1.3.2.2.-Desventajas.

Lograr accesar las localidades de la memoria principal es una de las restricciones de tiempo más críticas en una computadora.

El tiempo necesario para accesar memoria es más largo que el de transferencia de datos entre registros.

El problema en estas máquinas no es en tiempo de compilación, sino en tiempo de ejecución al tener que hacer accesos a memoria.

Implementar toda la pila con registros es muy caro.

Sin embargo, una solución intermedia está en mantener los elementos superiores de la pila en registros y los elementos inferiores en memoria.

1.4 - COMENTARIOS FINALES.

Como se puede ver, realizar una comparación de arquitecturas no es un trabajo sencillo, pues cada una tiene ventajas y desventajas.

Para realizar una comparación completa se requiere tomar en cuenta muchos parámetros.

Sin embargo es conveniente hacer comparaciones que permitan evaluar nuevas tecnologías.

En el siguiente capítulo se describe con mayor detalle la arquitectura del TRANSPUTER, ya que es el centro de este estudio.

CAPITULO 2

EL TRANSPUTER Y OCCAM

El transputer y su lenguaje de programación merecen especial atención, pues es la arquitectura con la que se trabajó en la generación de código.

Este dispositivo ha sido diseñado para soportar procesamiento paralelo.

Los registros usados en programación secuencial se muestran en la figura 2.1.

La pila de evaluación está formada por los registros A, B Y C. Esta pila es utilizada para evaluar expresiones, operandos de instrucción, instrucciones de comunicación y parámetros en llamadas de procedimientos.

El **WS**, es el registro que apunta a donde las variables locales y los apuntadores están almacenados. El apuntador a instrucción, contiene la dirección de la siguiente instrucción. El registro de Operando es usado para formar operandos de instrucciones.

El Transputer T800 cuenta también con una pila de tres registros para evaluar expresiones con operandos de punto flotante. Las direcciones de los operandos de punto flotante se forman en la pila de evaluación.

La unidad aritmético-lógica entera y la de punto flotante pueden operar concurrentemente.

La unidad de punto flotante utiliza solo el 20% del área del chip.

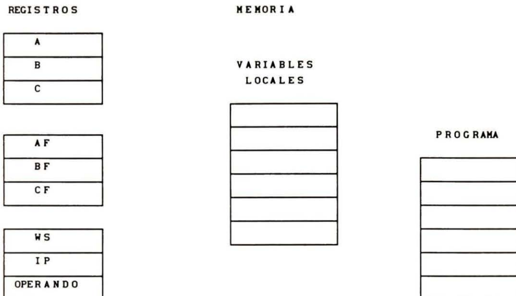


FIGURA 2.1 REGISTROS PARA PROGRAMACION SECUENCIAL

2.1.-LOS MODOS DE DIRECCIONAMIENTO.

Los modos de direccionamiento del Transputador son muy simples.

* **MODO INMEDIATO.**Toma el valor preparado por la instrucción en el registro de operando. Solo es posible transferir 32 bits, pero existen instrucciones cortas para mover 4,8,..,32 bits. El único destino es la pila de evaluación.

  Por ejemplo las instrucción siguiente:
  ldc 3

  En este caso se copia el valor 3 en el registro A, A se copia en B y B se copia en C.
  El término direccionamiento inmediato generalizado

conocido en algunos microprocesadores como "load address" en Inmos se implementa con la instrucción load pointer como sigue:
ldnlp disp (carga un apuntador de memoria externa al espacio de trabajo del proceso).

* **MODO RELATIVO.** El direccionamiento absoluto no existe en Transputer, excepto para instrucciones especiales. El direccionamiento relativo está disponible solo para saltos y garantiza que los módulos sean relocalizables. Los desplazamientos son valores de hasta de 32 bits signados.

* **MODO INDIRECTO DE REGISTRO.** Es llamado local cuando se refiere al registro de espacio de trabajo **W** y no local cuando se refiere al registro **A**.
  Algunos ejemplos son:
  stl disp (almacenamiento local)
  ldnl disp (cargado no local)
  lb (carga byte)
  wsub (palabra por índice)

2.2.- DISEÑO DEL CONJUNTO DE INSTRUCCIONES.

La memoria del transputer está organizada en un espacio lineal de 2^{32} bytes. Los 4 K de memoria interna son más rápidos que la externa. Los dos tipos de memoria difieren solamente en el rango de direcciones.

El conjunto de instrucciones fue diseñado para soportar lenguajes paralelos de alto nivel. Se fomenta el uso de lenguajes de alto nivel, ya que no existe un ensamblador disponible para usarse con el transputer.

Transputer es una máquina de pila, y sobre esto se debe

aclarar que el trabajo con datos escalares es efectivo, pero el trabajo con arreglos complejos es menos eficiente que en las máquinas tipo CISC.

El conjunto de instrucciones del Transputer usa la idea RISC de instrucciones cortas, rápidas y simples; pero las instrucciones tienen diferentes longitudes y tiempos de ejecución.

Se utiliza un almacén de instrucciones leídas para aumentar la velocidad del procesador.

Las 16 instrucciones básicas de 1 byte se presentan en la tabla 2.1.

Mnemónico	Función	Ciclos de reloj
ldl	load local	2
stl	store local	1
ldlp	load local pointer	1
ldnl	load no local	2
stnl	store no local	2
ldnlp	load no local pointer	1
eqc	equals constant	2
ldc	load constant	1
adc	add constant	1
j	jump	3
cj	conditional jump	2 ó 4
call	call	7
ajw	adjust workspace	1
pfix	prefix	1
nfix	negative prefix	1
opr	operate	ver tabla 2

TABLA 2.1. Instrucciones de un byte

El direccionamiento local es con respecto al registro W (Espacio de trabajo) y el direccionamiento no local es referente al registro A de la pila. Una estimación de INMOS es que aproximadamente el 80 % de las instrucciones ejecutadas son de 1 byte de longitud. Una instrucción de 1 byte consiste de 4 bits de código de operación y 4 bits de campo de constante.

El registro de operando es usado para generar constantes más grandes. Cuando se requieren constantes más largas que 4 bits, se hace uso de la instrucción de 1 byte *Pfix* (prefijo).

Cuando se ejecuta una instrucción prefijo los 4 bits del campo constante se cargan en el registro de operando O y son recorridos cuatro lugares a la derecha. Se pueden utilizar varias instrucciones prefijo para generar constantes más largas.

Posteriormente la instrucción que va a utilizar la constante larga, carga sus cuatro bits del campo de constante y es ejecutada con un operando de 32 bits que se encuentra en el registro de operando.

El registro de operando es limpiado depués de cada instrucción, excepto en prefijos. La instrucción de prefijo negativo complementa el registro de operando antes de hacer el corrimiento.

A diferencia de otras arquitecturas, el Transputer tiene dos instrucciones de comparación y solo una instrucción de salto condicional. Así el registro A es usado preferentemente, en lugar de una bandera en un registro de estado.

El salto condicional *cj* es realizado, si el contenido de A es falso.

La istrucción *OPR* es usada para generar las instrucciones restantes. El campo de datos en la instrucción de operación es usado como un valor de código de operación.

Lo anterior produce otras 16 instrucciones de 1 byte que se muestran en la tabla 2.2.

Las instrucciones Operar usan sus operandos en la pila. Por medio de la operación prefijo, instrucciones más largas a un byte pueden ser ejecutadas utilizando el registro de operando para extender el conjunto de instrucciones.

El conjunto básico de instrucciones del Transputer tiene alrededor de 70 instrucciones compuestas de dos bytes

incluyendo la multiplicación y la división.

2.3.-EL LENGUAJE OCCAM.

Occam ha sido descrito como el lenguaje ensamblador para procesadores paralelo.

Para soportar procesamiento paralelo en el Transputer, INMOS desarrollo Occam, un lenguaje de alto nivel con primitivas de procesamiento paralelo, basado en el trabajo extensivo de C.A. Hoare en el desarrollo de lenguajes CSP (communicating sequential process).

Occam es un lenguaje basado en los conceptos de concurrencia y comunicación. Puede ser usado para describir la estructura de un sistema en término de microcomputadoras conectadas .

Mnemónico	Función	Ciclos
rev	reverse	1
gcall	general call	3
bsub	byte subscript	1
wsub	word subscript	2
lb	load byte	5
add	add	1
sub	substract	1
gt	greater than	2
diff	difference	1
startp	start process	12
endp	end process	13
in	input message	2w + 19
out	output message	2w + 19
outword	output word	23
outbyte	output byte	23

TABLA 2.2. Instrucciones Operar de un byte.

Esencialmente el diseño de programas en Occam está basado en la comunicación entre procesos que cooperan y la sincronización de los mismos. De esta manera un programa está constituido de un conjunto de procesos que se intercomunican.

Un proceso ejecuta una secuencia de acciones y termina. Cada acción puede ser una asignación, una entrada o una salida. Una asignación cambia el valor de una variable, una entrada recibe un valor de un canal y una salida envía un valor a un canal.

En cualquier tiempo entre el inicio y terminación, un proceso debe estar listo y esperando para comunicarse en uno o más canales. La comunicación es sincronizada. Cuando un proceso de entrada y uno de salida están listos para comunicarse en el mismo canal, el valor de salida es copiado del proceso de salida al proceso de entrada. Después los dos procesos continuan.

Cada canal proporciona una manera de conexión entre dos procesos concurrentes.

Occam puede ser usado para programar una red de computadoras. Cada computadora con memoria local ejecuta un proceso con variables locales y cada conexión entre dos computadoras se implementa como un canal entre dos procesos.

Occam también puede utilizarse para programar una computadora individual. La computadora comparte tiempo entre procesos concurrentes, y los canales son implementados por valores en memoria.

2.3.1.-PROCESOS PRIMITIVOS Y CONSTRUCTORES.

Los programas de Occam pueden ser construidos partiendo de tres procesos primitivos: **entrada, salida y asignación.**

La asignación

v := e

coloca el valor de la variable v al valor de la expresión **e**.

La salida

c ! e

envía el valor de la expresión **e** por el canal **c**.

La entrada

c ? e

recibe el valor de la expresión **e** por el canal **c**.

Los constructores son usados para combinar procesos y formar procesos mayores. El constructor **SEQuential** causa que sus componentes sean ejecutados uno después de otro, terminando cuando el último componente termina. El constructor **PARallel** causa que sus componentes sean ejecutados concurrentemente, terminando solo después de que todos sus componentes han terminado. El constructor **ALTernative** selecciona un proceso para ejecución, terminando cuando el componente seleccionado termina. También los constructores **IF** y **WHILE** son proporcionados.

2.3.2.-CANALES Y COMUNICACION.

Cada proceso concurrente opera con sus propias variables y no son compartidas con ningún otro proceso. Los procesos concurrentes solo se comunican usando canales.

Un canal permite una comunicación entre procesos concurrentes. Cada canal conecta dos procesos: un proceso siempre envía sus salidas al canal y otro recibe entradas del canal. La comunicación es sincronizada y ocurre cuando un proceso envía y otro recibe.

La comunicación puede ser pensada como una asignación distribuida:

```
PAR
  c ! x

  c ? y             es lo mismo que          y := x
```

2.3.3.-REPETICION.

La repetición es proporcionada por un constructor convencional **WHILE**. Un proceso es ejecutado repetidamente hasta que el resultado de la evaluación de una expresión es falso.

En Occam no existen **goto, etiquetas o exit** .

2.3.4.-CONSTRUCTOR ALTERNATIVO.

El constructor alternativo selecciona uno de los componentes para ejecución. Cada componente tiene una **guardia,** la cual es una entrada con una condición opcional. El proceso que más pronto está listo para ser ejecutado es seleccionado. Si varias guardias están listas, se selecciona una arbitrariamente.

Un ejemplo de esta construcción es:

```
WHILE going
  ALT
    buffer.in ? ch
      buffer.out ! ch
    stop ? any
      going := FALSE
```

2.3.5.-REPLICADORES.

Los replicadores son utilizados para describir colecciones de procesos similares. El efecto del replicador, es el mismo que una lista de procesos, en la cual cada elemento es una copia del proceso en el replicador.

Por ejemplo:

```
PAR  i=0 [0 FOR n]               PAR
  c[i] ! i               =         c[0] ! 0
                                   c[1] ! 1
                                   c[2] ! 2
                                   ....
                                   c[n-1] ! n -1
```

El uso principal del replicador es la construcción de arreglos de procesos concurrentes.

El replicador puede ser usado con ALT, SEQ y PAR.

Cuando se usa con SEQ es similar al FOR convencional de otros lenguajes.

2.3.6.-ABSTRACCION.

Hay un mecanismo de abstracción que permite que un proceso reciba un nombre. Un proceso con nombre puede tener parámetros y en particular los parámetros pueden ser canales.

2.3.7.-VALORES.

El lenguaje descrito es independiente de los tipos de datos proporcionados. Claramente, los procesos primitivos y los constructores pueden ser usados con cualquier tipo de datos.

2.3.8.- TIPOS DE DATOS.

Los datos pueden ser de tipo arreglo o primitivo.

La categoría de datos primitivos está constituida por:

INT	(Enteros)
BYTE	
BOOL	(Booleanos)
TIMER	
CHAN OF protocol	(Canales de comunicación)

La categoría de los datos de tipo arreglo:

En el arreglo [e]T el valor de "e" define el número de componentes en un arreglo del tipo T del arreglo.

Los datos de tipo entero pueden ser de 16, 32 ó 64 bits.
Los datos de tipo real pueden ser de 32 o de 64 bits.

2.3.9.- UN PROCEDIMIENTO EN OCCAM.

El procedimiento "números" se presenta a continuación para resaltar algunos detalles del lenguaje.

Al observar el programa se puede ver que no hay una palabra reservada para indicar donde inicia y donde termina un bloque, ya que esto se expresa por la sangría.

Otro punto importante que se debe mencionar es que en OCCAM si hay anidamiento de procedimientos.

El símbolo ":" expresa el final de una definición.

```
PROC Numeros(CHAN OF INT in,out)
  INT i:
  SEQ
    i:= 2
    WHILE i <> EndToken
      PRI ALT
        in ? i
          SKIP
        TRUE & SKIP
          SEQ
            out ! i
            i := i + 1
:
```

CAPITULO 3

SMALL C

3.1.-SOBRE SMALL C.

SMALL C aparece en 1980 en la revista "Dr. Dobb's Journal" en un artículo titulado *UN COMPILADOR DE SMALL C PARA EL 8080*, en el cual Ron Cain presenta un pequeño compilador para un subconjunto del lenguaje C. Con un algoritmo de un paso el compilador generaba código en lenguaje ensamblador para el procesador 8080.

A través del tiempo ha evolucionado el compilador, pues nuevas versiones del compilador han sido escritas y la que se utiliza en este trabajo es la versión 2.2. y el autor es J.E. Hendrix.

3.2.-SINTAXIS DEL LENGUAJE.

ARGUMENT DECLARATION:

Objet Declaration.

ARGUMENT LIST:

NameList.

DIRECTIVE:

```
#INCLUDE "FILENAME"
#INCLUDE <FILENAME>
#INCLUDE FILENAME
#define  Name  String de caracteres?
#if def Name
#else
#endif
```

```
     #asm
     #endasm
CONSTANT
     Integer
     'Character'    (Secuencias de escape permitidas)
     'CharacterCharacter' (Secuencias de escape permitidas)
EXPRESION CONSTANTE
     Constant
     Operator Constant Expression
     ConstantExpression Operator ConstantExpression
     (Constant Expression)
DECLARATOR:
     Objet Initializer?
Secuencias de Escape:
     \n     (new line)
     \t     (tab)
     \b     (backspace)
     \f     (formfeed)
     \OctalInteger
     \OtherCharacter
EXPRESSION:
     Primary
     Operator Expression
     Expression Operator
     Expression Operator Expression
FUNCTIONDECLARATION:
     void? Name(ArgumentList?)
               ArgumentDeclaration?..
               CompoundStatement
GLOBALDECLARATION:
     ObjetDeclaration
     FunctionDeclaration
```

INITIALIZER:

```
=ConstantExpression
={ConstantExpressionList}
=StringConstant
```

OBJET:

```
Name
*Name
Name[ConstantExpression?]
Name()
(* Name)()
```

OBJETDECLARATION:

```
Type DeclaratorList;
extern Type? DeclaratorList;    (global only)
```

PRIMARY:

```
Name
Constant
StringConstant
Name [Expression]
Primary (ExpressionList?)
(Expression)
```

PROGRAM:

```
Directive?... GlobalDecalaration...
```

STATEMENT:

```
;
ExpressionList;
return ExpressionList?;
Name:
goto Name:
if (ExpressionList) Statement
if (ExpressionList) Statement else Statement
switch (ExpressionList) CompoundStatement
case ConstantExpression
default:
```

```
     break;
     while (ExpressionList) Statement
     for (ExpressionList?;
          ExpressionList?;
          ExpressionList?) Statement
     do Statement while (ExpressionList);
     continue;
     {ObjetDeclaration? ... Statement?...}
```

STRINGCONSTANT:

```
     "CharacterString"
```

TYPE:

```
     char
     int
     unsigned
     unsigned char
     unsigned int
```

3.3.-DIFERENCIAS CON C.

3.3.1.-TIPOS DE DATOS.

Las limitaciones más significativas de SMALL C son los tipos de datos que soporta: enteros, caracteres, apuntadores, y arreglos de enteros y caracteres de una dimensión.

La limitación anterior hace que no todos los programas de C se puedan traducir a SMALL C.

3.3.2.-IDENTIFICADORES NO DECLARADOS.

SMALL C cuando ve un nombre no declarado asume que es una función y automáticamente la declara como tal. Si la referencia es seguida por paréntesis, una llamada es generada, de otra

manera, la dirección de la función es generada por referencia a la etiqueta con el nombre. Si la función es definida más tarde, la etiqueta para la función es generada. Si por otro lado, no es definida, entonces SMALL C automáticamente declara el nombre como una referencia externa, para ser resuelta en tiempo de ejecución.

Otra limitación significativa, es la carencia de soporte para estructuras y uniones. Esta diferencia es menos significativa que la anterior.

3.3.3.-NOMBRES DE FUNCIONES COMO ARGUMENTOS.

SMALL C acepta como argumentos

```
int arg
```

para declarar un argumento formal que apunta a una función, y

```
arg(...)
```

para llamar a la función. Esto es porque SMALL C evalúa la expresión y usa el resultado como offset en el segmento de código para la función deseada.

Una mejor sintaxis que aumenta la compatibilidad con C es:

```
int (* arg)()
```

y

```
arg(..)
```

respectivamente.

3.3.4.-LLAMADOS INDIRECTOS A FUNCIONES.

SMALL C ve a cualquier expresión seguida por paréntesis como una llamada a función, mientras que C acepta solamente expresiones aritméticas basadas en nombres de funciones como (* func) o (*fa[x]).

Pero SMALL C acepta expresiones como:

ia[x](...) que se puede expresar también así:
(* ia[x])()

3.3.5.-PASO DE PARAMETROS.

A la inversa de los compiladores de SMALL C pasa parámetros de izquierda a derecha.

Los compiladores de C lo hacen en orden inverso para trabajar funciones con número variable de argumentos, tales como **scanf y printf.** SMALL C no modifica su forma de pasar parámetros, solo agrega un contador del número de argumentos de una función que se pasa en el registro **cl.**

3.3.6.-VALORES RETORNADOS.

SMALL C retorna solamente datos de tipo entero. Los compiladores de C devuelven valores de cualquier tipo.

3.3.7.-EVALUACION DE OPERADORES DE ASIGNACION.

SMALL C evalua primero el lado izquierdo de los operadores de asignación, antes de evaluar el lado derecho. Esto implica que variables usadas para determinar el destino de valores asignados son no afectados por la expresión del lado derecho.

La mayoría de los compiladores de C, evaluan el lado derecho primero, permitiendo la influencia del destino.

3.3.8.-CONVERSION DE CARACTERES A ENTEROS.

SMALL C al igual que otros compiladores de C promueven a enteros los caracteres, considerándolos signados.

3.3.9.-SINTAXIS DE LA DIRECTIVA #include.

La directiva #include no requiere comillas ni paréntesis angulares como en las otras versiones de C.

3.3.10.-VIEJO ESTILO DE OPERADORES DE ASIGNACION.

SMALL C no reconoce el estilo original de asignación en el cual el signo fue escrito como prefijo antes que como sufijo.
Por lo tanto secuencias como =+ o =* son tomadas como dos operadores en lugar de uno.

3.4.LA IMPLEMENTACION DE SMALL C.

El compilador de Small C escrito por Hendrix está organizado de la siguiente manera:

FIGURA 3.1 ORGANIZACION DEL COMPILADOR DE SMALL C

3.4.1.-Las funciones de la parte posterior.

Las funciones de este módulo se dividen en tres grupos:
a).- Las que generan código.
b).- Las funciones de optimización.
c).- Las funciones de salida.

Lo más importante para mencionar en este módulo es el código intermedio que se genera al compilar un programa.
Como es conocido, al compilar un programa de un lenguaje

"x", lo primero que se realiza es un análisis léxico, el cual tiene el trabajo de identificar los símbolos válidos en un lenguaje.

El compilador también posee un analizador sintáctico y éste recibe los símbolos, "tokens", del analizador léxico para verificar que la construcción que es analizada está bien formada de acuerdo a la gramática del lenguaje.

Finalmente, y una vez que se ha reconocido una expresión bien formada se genera el código correspondiente a la construcción analizada.

Antes de generar el código, es necesario optimizar, es decir, producir un código que ocupe menos memoria y se ejecute más rápidamente. Con este fin en el compilador de Small C se genera un código intermedio, llamado pseudocódigo.

Esos pseudocódigos son pequeños valores enteros, cada uno de los cuales corresponde a una instrucción, una secuencia de instrucciones o una instrucción parcial.

Los pseudocódigos tienen nombres semánticos, para reconocerlos, aunque no se esté muy familiarizado con los p-codes.

La función "outcode" se encarga de traducir los p-codes a lenguaje ensamblador.

En estos pseudocódigos se utiliza la siguiente simbología:

SIMBOLO	SIGNIFICADO
0	El valor cero.
1	El registro primario, (AX).
2	El registro secundario, (BX).
b	Byte.
f	Salto por condición falsa.
l	Actual etiqueta de la pila de literales.

m — Referencia a memoria por etiqueta.

n — Constante numérica.

p — Referencia indirecta a memoria, por medio de un apuntador en el secundario.

r — Repetición "r" veces.

s — Referencia a la pila.

w — Word.

_ — Instrucción incompleta.

Algunos ejemplos de los pseudocódigos son:

ADD12 — Suma contenidos del registro secundario al primario.

DBL2 — Dobla los contenidos del registro secundario.

Small C tiene un total de 107 pseudocódigos para la generación de código.

Algunas secuencias de traducción son:

```
P-CODE            TRANSLACION
ADD12        \211 ADD AX,BX\n
CALLm        \020 CALL <m>\n
DBL1         \010 SHL AX,1\n
DIV12        \011CWD\nIDIV BX\n
```

Las cadenas y los pseudocódigos están relacionados por medio de el arreglo code[].

Las cadenas de translación inician con un byte que es codificado en una secuencia octal y contiene información para el optimizador.

3.4.1.1.-EL OPTIMIZADOR.

En SMALL C se realiza una optimización local y solo se hace en la evaluación de expresiones.

Para optimizar el código se utiliza un buffer, al cual se genera el código intermedio antes de optimizarlo.

El buffer tiene la siguiente forma:

	P-CODE	VALOR
stage →	25	1
	12	0
snext →		

FIGURA 3.2 BUFFER DE OPTIMIZACION.

Se utilizan 47 secuencias para optimizar el código. Estas secuencias contienen un antecedente y un consecuente.

La función Peep() busca empatar los códigos en el buffer con alguna de las secuencias y si tiene éxito se substituye el código en el buffer por el lado derecho de la regla.

Para el empatamiento y la substitución se hace uso de un lenguaje diseñado por el autor.

Un ejemplo de estas secuencias es:

```
ADD1n,0         ifl|m2,0,ifl|0,rDEC1,neg,0,ifl|p3,rINC1,0,0
```

Si se utiliza esta regla se busca empatar el pseudocódigo ADD1n y si se tiene éxito se ejecuta el lado derecho de la siguiente manera:

Si el valor asociado al P-CODE es menor que -2 no se hace nada y termina, pero si es menor que cero genera rDEC1 y neg.

Si no cumplió con los dos casos anteriores entonces el valor es positivo y si este valor es menor que tres entonces se genera rINC1 y termina.

Los metacódigos de optimización son:

go Le dice al optimizador que ajuste el apuntador "n" localidades hacia adelante o hacia atrás.

Su aparición es así: go | [b], donde [b] es un valor que puede ser m1 (-1), m2 (-2), m3 (-3), p1 (1), p2 (2) o p3 (3).

gc Copia el código que está a "n" localidades en cualquier dirección a la localidad apuntada por **snext.**

gv Copia el valor de "n" localidades hacia arriba o hacia abajo de la entrada actual (snext) a la localidad **snext + 1.**

sum Este código reemplaza el valor apuntado por **snext** por la suma de este valor y el valor que se encuentra a "n" localidades de la entrada actual.

neg Niega el valor de la localidad apuntada por **snext + 1.**

ife Este código permite tomar una decisión de acuerdo al valor de la entrada actual. En caso de que el valor sea igual al de la entrada actual, se genera el código que le sigue, de lo contrario lo salta.

ifl Este código funciona igual al anterior, pero para la condición "menor que".

swv Intercambia los valores de la actual entrada y el valor que se encuentra a "n" entradas de la entrada actual apuntada por **snext.**

topop El propósito de está instrucción es convertir un POP2 en otro pseudocódigo.

3.4.2.-PARTE FRONTAL.

Se le llama parte frontal porque las funciones de este módulo establecen la entrada al "parser" o analizador

sintáctico.

En este módulo se tienen las funciones que obtienen, preprocesan y hacen el análisis léxico del código fuente, antes de enviarlo al analizador sintáctico.

En Small C el preprocesador trabaja entre las funciones de lectura y el analizador léxico y no separadamente como en otros compiladores.

La entrada se recibe en un arreglo de caracteres "mline" que es preprocesado a "pline".

3.4.3.-EL ANALIZADOR SINTACTICO.

Small C utiliza el método de análisis sintáctico llamado de descenso recursivo.

Al hacer un análisis sintáctico, el programa es dividido en partes cada vez más pequeñas, hasta formar un árbol basado en la gramática del lenguaje.

En la raíz se encuentra el programa completo, mientras que en los nodos intermedios se encuentran construcciones tales como estatutos o expresiones.

El método de descenso recursivo inicia en la raíz de tal árbol y mira para una secuencia de construcciones globales. Cuando reconoce el inicio de una de ellas, trata de reconcer las partes que lo integran y esto se repite hasta llegar a las unidades léxicas más pequeñas, llamadas "tokens".

3.4.4.- EL ANALIZADOR DE EXPRESIONES.

El analizador de expresiones es la parte más difícil del compilador y está implementado con 14 funciones de nivel, una para cada nivel de precedencia y la función **primary** que reconoce los operandos sobre los cuales los operadores

trabajan.

Los operandos reconocidos por **primary** aparecen como identificadores (variables, apuntadores y funciones), constantes o subexpresiones en paréntesis.

Si es un nombre seguido por un paréntesis es detectado en el nivel 14.

Para comunicar los niveles se utilizan 4 funciones: **down(), down1(), down2()** y **skim()**.

En estas cuatro funciones está la lógica de las funciones de nivel. Esta es una forma elegante en la que el autor logró escribir este analizador sin un exceso de código.

Las funciones líderes del manejo de expresiones son:

Doexpr() es llamada cuando la función **statement()** no reconoció una palabra clave.

Constexpr() Es llamada cuando la sintaxis demanda una expresión constante.

Expresión(). Es una de las dos funciones que inician el análisis de expresiones, llamando a nivel_uno.

Test(). Su próposito es generar código para evaluar y probar una expresión para verdadero o falso.

Los operadores reconocidos en cada nivel son:

Nivel 1 {!=,^=,&=,+=,-=,*=,/=,%=,>>=,<<=,=}

Nivel 2 {?:}

Nivel 3 {||}

Nivel 4 {&&}

Nivel 5 {|}

Nivel 6 {^}

Nivel 7 {&}

Nivel 8 {==,!=}

Nivel 9 {<=,>=,<,>}

Nivel 10 {<< , >>}

Nivel 11 {+ -}

Nivel 12	{*,/,%}
Nivel 13	{++,--,~, ! , - , * ,& , sizeof}
Nivel 14	{[,()}
Primary	{(, nombre,constante}

Una descripción, un tanto burda del funcionamiento del analizador de expresiones es la siguiente:

Al llamar a nivel_uno, se va descendiendo de nivel mientras no se reconoce un operador, el descenso se hace mediante las funciones de tubería (down, down1, down2 y skim) ya mencionadas.

Al reconocer un operador, se inicia el regreso o se continua el descenso para leer un operando.

El trabajo del analizador de expresiones se continua hasta que la expresión ha sido analizada completamente.

CAPITULO 4

GENERACION DE CODIGO PARA EL TRANSPUTER

El problema de la generación de código se puede ver como una función de el manejo de memoria, asignación de registros, selección de instrucciones y selección del orden de evaluación.

Un compilador está compuesto de un analizador léxico, un analizador sintáctico y un generador de código .

Para escribir un compilador de un lenguaje "x" para diversas máquinas, el analizador léxico y sintáctico permanecen sin cambios, mientras que el generador de código si es cambiado, debido a que depende de la arquitectura.

Se seleccionó para la experiencia de este trabajo el compilador de SMALL C, porque se dispone de su programa fuente, y debido a que no es necesario modificar los dos primeros componentes, se modificó solo la parte correspondiente a la generación de código.

Debido a que el objetivo de este trabajo no es construir un compilador, sino estudiar el código generado, solo se trabajo sobre el lenguaje, es decir, sin entradas ni salidas.

4.1.- EL TRANSPUTER Y EL LENGUAJE OCCAM.

El transputer con el que se trabajó, se encuentra instalado en una máquina PC AT .

El compilador con el que se trabajó, es el del lenguaje OCCAM elaborado por INMOS.

Este compilador de Occam tiene algunas carencias todavía, ya que aún no se le ha implementado el manejo de funciones y el "case", que si están definidos en la gramática. Para compilar

programas es OCCAM se hace uso de una interfaz que comunica a la PC con la tarjeta del transputer.

Occam es un lenguaje que da muchas facilidades para la comunicación entre procesos, pero para la comunicación con el usuario, se tiene la necesidad de usar un gran número de librerías.

4.2.-EL FORMATO DE SALIDA.

Debido a que no se dispone de un ensamblador se generó código a OCCAM
El código se generó en linea, dentro del formato de un programa en OCCAM.

El formato de salida de un programa, es entonces:

```
PROC main.program(CHAN OF ANY from.filer,to.filer)
    {Declaración de variables}
     PROC uno({parámetros formales})
       SEQ
          {Declaración  de variables}
          {Declaración de procedimientos}
          GUY
             {Códigos  de  máquina}
     :
      SEQ
        GUY
          {Códigos de máquina}
:
```

Así, si tenemos el programa de entrada:

```
main(){
  int  x,y,z;
  x = 2;
  y = x;
  z = x * y;
}
```

Su salida es la que se muestra en la figura 4.1.

```
PROC main.program(CHAN OF ANY from.filer,to.filer)
     INT x,y,z:
      [1]INT tempo:
      SEQ
        GUY
          LDLP 3
          LDC  2
          REV
          STNL 0
          LDLP 2
          LDLP 3
          LDNL 0
          REV
          STNL 0
          LDLP 1
          LDLP 3
          LDNL 0
          STL 0
          LDLP 2
          LDNL 0
          LDL 0
          MUL
          REV
          STNL 0
:
```

FIGURA 4.1. PROGRAMA GENERADO EN OCCAM.

El código generado es colocado dentro de la construcción GUY.

Se debe cuidar la sangría de cada línea.

En el código generado se puede ver una variable que en el programa fuente no aparece. Esta variable es una variable auxiliar que permite salvar los contenidos de la pila de evaluación.

4.3.- CODIGOS QUE SE PUEDEN UTILIZAR DENTRO DE LA CONSTRUCCION "GUY".

No todo el conjunto de instrucciones se puede colocar dentro de la construcción GUY.

Solo se pueden colocar dentro de GUY las siguientes instrucciones:

OPERACIONES DIRECTAS.

ADC	(Suma constante)
CJ	(Salto condicional)
J	(Salto incondicional)
LDC	(Carga una constante en la pila de evaluación)
LDL	(Carga local)
LDLP	(Carga apuntador local)
LDNL	(Carga no-local)
LDNLP	(Carga apuntador no-local)
STL	(Almacenamiento local)
STNL	(Almacenamiento no-local)

OPERACIONES CORTAS INDIRECTAS.

ADD	(Suma)
BSUB	(Subíndice byte)
DIFF	(Diferencia)

GT	(Mayor que)
LB	(Cargar byte)
PROD	(Producto)
REV	(Reverse)
SUB	(Substracción)
WSUB	..(Subíndice Word)

OPERACIONES INDIRECTAS LARGAS.

AND	(Y lógico)
BCNT	(Contador de bytes)
CCNT1	(Checa contador de 1)
CFLERR	(Checa simple longitud infinita o NaN)
CSNGL	(Checa simple)
CSUB0	(Checa subíndice de 0)
CWORD	(Checa palabra)
DIV	(Divide)
FMUL	(Multiplicación fraccionaria)
LADD	(Suma larga)
LDIFF	(Diferencia larga)
LDINF	(Carga simple longitud infinita)
LDIV	(División larga)
LDPI	(Carga apuntador a instrucción)
LDPRI	(Carga prioridad actual)
LDTIMER	(Carga Timer)
LMUL	(Multiplicación larga)
LSHL	(Corrimiento a la izquierda largo)
LSHR	(Corrimento a la derecha largo)
LSUB	(Substracción larga)
LSUM	(Suma larga)
MINT	(Entero mínimo)
MOVE	(Mueve mensaje)
MUL	(Multiplica)
NORM	(Normaliza punto flotante)

NOT	(No)
OR	(O)
REM	(Residuo)
ROUNDSN	(Redondea un número de simple longitud de punto flotante)
SB	(Almacena Byte)
SETERR	(Coloca Error)
SHL	(Corrimiento a la izquierda)
SHR	(Corrimiento a la derecha)
STTIMER	(Almacena Timer)
SUM	(Suma)
TESTERR	(Prueba error falso y limpia)
TESTHALTERR	(Prueba y para en error)
TESTPRANAL	(Prueba el procesador analizando)
UNPACKSN	(Desempaca números de punto flotante de longitud simple)
WCNT	(Contador de palabras)
XDBLE	(Extiende a doble)
XOR	(Or exclusivo)
XWORD	(Extiende a Word)

El utilizar solo un subconjunto de las instrucciones genera una serie de restricciones tales como:

* No se puede hacer una llamada a un procedimiento.
* No se puede hacer el ajuste de apuntador del espacio de trabajo.

Como resultado, se incluyen partes del código en Occam.

4.4.- ALGUNOS CODIGOS GENERADOS.

En seguida se presentan una serie de códigos que generan algunas construcciones de "C".

4.4.1.- ASIGNACION.

Para la expresión x = 8 el código producido al compilar la expresión es:

```
LDLP  x   (Se carga en la pila la dirección de x).
LDC   8   (Se carga en la pila la constante 8)
REV           (Intercambia los contenidos de los
            registros A y B).
STNL  0   (Almacena los contenidos del registro B en
          la dirección del registro A +
          Desplazamiento).
```

El código óptimo para esa expresión es:

```
LDC 8
STL x
```

4.4.2.-ARREGLOS.

La compilación de la expresión :

x[i] := y [4]

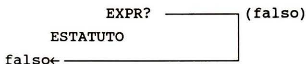
es la siguiente:

```
LDLP x
LDL i
REV
WSUB
LDLP y
LDC 4
REV
WSUB
LDNL 0
REV
STNL 0
```

La instrucción WSUB permite calcular la dirección del elemento a accesar o almacenar.

4.4.3.- LA CONSTRUCCION IF.

Para la expresión **if (expr) statement** se genera el código :



Para la construcción:

```
if ( x < 30)
     x = 8;
    LDLP 0   ─┐
    LDNL 0    │
    LDC 30    ├─ EVALUACION DE EXPRESIONES.
    REV       │
    GT       ─┘
    CJ .E2   ──┐ (FALSE)
    LDLP 0     │
    LDC 35     │
    REV        │
    STNL 0     │
    :E2    ←───┘
```

La expresion:

if(expr) estatuto1
else estatuto2

Se genera lo siguiente:

4.4.4.- LA CONSTRUCCION WHILE.

La expresión:

while (expr) estatuto

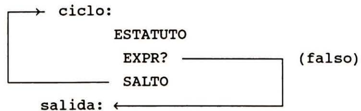
Se compila a:

4.4.5.- LA CONSTRUCCION DO-WHILE.

Una expresión de la forma:

do estatuto while (expr)

Se traduce a un esquema de la forma:

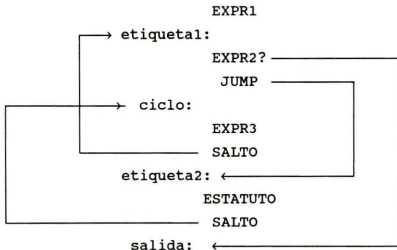


4.4.5. - LA CONSTRUCCION FOR.

La expresión:

for(expr1 ; expr2 ;expr3) estatuto

La forma del código ensamblado es:



4.5.- LA COMPILACION DE EXPRESIONES.

La compilación de expresiones es el punto más significativo de la generación de código, ya que se debe vigilar que la pila de evaluación no se llene.

Una expresión puede representarse por medio de un árbol. En dicho árbol los nodos intermedios son operadores y los nodos hoja son operandos.

Por ejemplo la expresión:

Y * 5

tiene la representación:

Para evaluar esta expresión se requieren dos registros solamente, y por lo tanto no rebasa la pila.

El problema se presenta cuando el árbol tiene más de tres niveles de profundidad, ya que entonces se requieren más de tres registros y se hacen necesarios almacenamientos de resultados intermedios.

La forma en que se cuida el sobreflujo de la pila, es mediante una variable global que se incrementa o decrementa según se carga o almacena un elemento de la pila.

El salvado de elementos de la pila se hace exactamente al evaluar una rama del árbol y si el número de registros disponibles es inferior a 2.

Para el problema de generación de código óptimo se requiere de tres elementos:

a).-Elegir el orden de evaluación de las instrucciones del código intermedio.

b).-Seleccionar instrucciones de máquina a ser empleadas.

c).-Hacer una asignación de registros óptima.

Para realizar las tres tareas anteriores se utilizan métodos de programación dinámica o de coloreo.

En el presente trabajo no se enfatizó sobre la optimización del código.

El caso en el que se trabajó más especificamente, es en el de la selección de instrucciones, tomando el método utilizado por Hendrix y que es un método basado en empatamiento de patrones como se mencionó en el capítulo anterior.

Sin embargo, ya que el conjunto de instrucciones y el de modos de direccionamiento es muy reducido, el problema de la selección de instrucciones se simplifica.

4.5.1.- EL COMPILADOR CUIDANDO EL SOBREFLUJO DE LA PILA.

En seguida se hace una descripción de la implementación de SMALL C y la forma en que se trabajó con la pila de tres registros.

Se ha mencionado anteriormente, que el análisis de una expresión, se realiza mediante el uso de 14 funciones de nivel. La comunicación entre los niveles más profundos y los niveles superiores se efectúa mediante un arreglo que tiene la información del operando leído.

El arreglo is tiene 7 elementos y son:

Elemento	Contiene
is[ST]	Dirección en la tabla de símbolos, sino 0.
is[TI]	Tipo de objeto indirectamente referenciado, cero de otra manera.
is[TA]	Tipo de dato de la dirección, sino 0.
is[TC]	Tipo de Constante, sino 0.
is[CV]	Valor de la constante.
is[OP]	p-code del más alto operador binario.
is[SA]	Dirección en el buffer de "oper 0", sino 0.

Cuando se lee el inicio de una expresión, se inicializa al valor 3 la variable **disponibles**.

Se describirá a mayor detalle el funcionamiento de las funciones **down()**, **down1()** y **down2()**, que son fundamentales en el análisis de una expresión.

4.5.1.1.- DOWN().

Esta función es llamada por cada nivel de la jerarquía que mira para operadores binarios. Primero pasa el control a la rama izquierda, mediante la función DOWN1() para analizar el

operando izquierdo. Al recibir el control otra vez, si ve uno de los operadores anticipados, pasa el control a la rama derecha por medio de DOWN2() y después DOWN1().

4.5.1.2.- DOWN1().

La función **down1()** recibe como parámetro la dirección de la función de nivel objetivo, salva la dirección actual del buffer de optimización y si al regreso de la función de nivel llamada, el valor de is[TC] indica que resultó un valor constante, elimina del buffer de optimización el código generado.

4.5.1.3.- DOWN2 ().

Es la función más difícil del analizador.

Se debe recordar que está función es llamada por DOWN(), cuando se ha reconocido un operador binario.

Recibe como parámetros los p-codes de los operadores (oper y oper2), el nivel objetivo (level 1) y los arreglos is[] e is2[]. Is[] contiene las propiedades del operando izquierdo .

Is2[] recibirá las propiedades del operando del lado derecho.

PSEUDOCODIGO.

INICIO

1.- Salva la dirección de la entrada actual al buffer de optimización.

2.- **Si el operando izquierdo es una constante**

3.- Se llama el siguiente nivel, mediante down1 y si se requiere, se carga un elemento en la pila y **se** *decrementa el número de disponibles.*

4.- **Si operando izquierdo es cero.**

Coloca la información de la dirección actual en el buffer de optimización en la localidad is[SA].

FinSi

5.-Sino es una constante el operando izquierdo.

6.- Si disponibles es menor o igual a uno

7.- Se obtiene una variable temporal

8.- Se prende una bandera.

9.- Se salva el tope de la pila en la temporal.

10.- *Se incrementa disponibles.*

11.- Carga el operando derecho en la pila mediante DOWN1()

Decrementa disponibles.

12.- Si el operando derecho es una constante.

13.- Si derecho es cero

Se coloca is[SA] a la dirección original.

14.- Elimina la instrucción PUSH y el código generado.

15.- Si op == ADD

16.- Cargar la constante en la pila

Decrememementar *disponibles.*

17.- Sino

18.- Carga la constante en la pila.

Intercambia los dos primeros elementos de la pila.

Decrementa disponibles.

Finsi

19.- Sino (Derecho no constante)

21.- Si se prendió la bandera

20.- Realiza un POP (carga a la pila de evaluación, el contenido de la variable temporal en memoria).

Libera la variable.

```
          Decrementa disponibles.
        Finsi (Derecho constante).
      Finsi (Izquierdo constante).
21.-Si OP es binario
22.-    Si el izquierdo o el derecho son de tipo unsigned
          Se seleccionan operaciones sin signo.
23.-    Si ambos son constantes
24.-      Se indica que el resultado es constante is[TC].
25.-      Se pasa el valor de la constante is[CV].
26.-      Elimina el código derecho.
27.-      Si es constante sin signo
            Pasa is[TC] = UINT.
28.-    Sino (Ambos constantes)
29.-      Realiza operación
          Incrementa disponibles.
30.-      Si op = Substracción y ambos son direcciones
31.-        Se divide el resultado entre dos.
          Finsi.
32.-      Pasa el p-code del operador is[OP] = oper.
        Finsi (Ambos Constantes)
33.-    Si op = ADD o SUB
34.-      Si ambos son direcciones
35.-          Pasa resultado no es una dirección.
36.-      Sino (Ambos direcciones)
37.-        Si el derecho es dirección.
38.-              Pasa la dirección de la tabla derecha.
39.-              Pasa el contenido de is2[TI]
40.-              Pasa el contenido de is2[TA]
41.-          Sino
                Pasa el izquierdo por default.
              Finsi  (Derecho dirección).
        Finsi (Si op = ADD o SUB).
42.-    Si izquierdo no está en  la  tabla  de  símbolos  o
```

derecho está en la tabla de símbolos y es unsigned.

43.- Pasa el valor de is2[ST].

44.- **Sino**

45.- Pasa la dirección en la tabla del operando izquierdo o cero.

Finsi.

Finsi (Operador Binario).

Cuando se hace un PUSH en este algoritmo, se está salvando el subárbol izquierdo ya evaluado.

La variable **disponibles** se ha utilizado para controlar el sobreflujo de la pila.

4.5.1.- CODIGO GENERADO EN LA COMPILACIÓN DE UNA EXPRESION.

Para la expresión :

```
f = (f * (f + ( f - (f * 5))))
```

Se genera el código :

```
LDLP f
LDLP f
LDNL 0
STL  0
LDLP f
LDNL 0
STL  1
LDLP f
LDNL 0
STL  2
LDLP f
```

```
LDNL 0
STL 3
LDC 5
LDL 3
MUL
LDL 2
REV
SUB
LDL 1
ADD
LDL 0
MUL
STNL 0
```

4.5.2.- CODIGO GENERADO CON OCHO REGLAS DE OPTIMIZACION.

El código generado utilizando solamente 8 reglas de optimización es de 19 lineas, 5 menos que el código anterior.

El porcentaje promedio de lineas reducidas por medio de la optimización con 8 reglas oscila entre el 10% y el 15 %.

4.6.- LOS PSEUDOCODIGOS.

Siguiendo la metodología implementada por Hendrix de utilizar un pseudocódigo intermedio antes de la generación de código, se cuenta con un total de 64 códigos del compilador para Transputer contra 107 del compilador de 8086.

Los mnemónicos son muy parecidos a los utilizados en 8086, pues indican operaciones y no instrucciones.

Algunos ejemplos de estos pseudocódigos son:

PSEUDOCODIGO	CADENA ASOCIADA
ADD	ADD\n

ADDn ADC <n>\n

POINTs LDLP <n>\n

La reducción del número de pseudocódigos se debe a:

a).- Que se opera en la pila.

b).- No existen operaciones en memoria

c).- Los movimientos entre registros se utilizan poco, solo cuando el orden de los operandos lo requiere en una operación particular.

4.7.- PROBLEMAS SURGIDOS AL IMPLEMENTAR LOS CAMBIOS .

En mi opinión, el compilador de SMALL C está escrito de una manera muy elegante y modular para poder maniobrar sobre él y hacer los cambios que uno desee.

Las restricciones en este caso, surgen de que no se cuenta hasta el momento con un ensamblador para el transputer.

Generar código a Occam y ajustarse a el hecho de que no todas las instrucciones se pueden utilizar como código en linea, constituyen el obstáculo de mayor importancia.

Entre las instrucciones que no se pueden utilizar, están las de llamada a un procedimiento y las de ajuste del apuntador al espacio de trabajo.

4.7.1.-LLAMADAS A FUNCIONES.

Las llamadas a una función, se tienen que realizar en Occam, por lo que en este punto se va a especificar como es que se compila.

Para llamar a una función se pasan los parámetros en la pila de evaluación y si hay más de tres se utlizan las primeras localidades de memoria adjuntas al WS (espacio de trabajo).

De tal modo que la compilación de un llamado en ensambldor

sería de la siguiente manera:

Para la función nombre(x,y,z,w)

```
LDLP W
STL 0  (El cuarto parámetro se salva en  0)
LDLP X  ┐
LDLP Y  ├ Parámetros en la pila.
LDLP Z  ┘
CALL (DESPLAZAMIENTO DE NOMBRE)
```

La compilación de una función a OCCAM se implementó así:

Dada la entrada

```
main(){
     int x;
     x = 5;
     aux(x);
}
aux(c) int c;{
}
```

En la salida la compilación de la función main() da el siguiente código:

```
PROC main.program(CHAN OF ANY from.filer,to.filer)
  INT x:
  PROC aux(INT c)
    SEQ
  :
  SEQ
    GUY
      LDC 5
      STL 0
    aux(x)
:
```

Como se puede ver, el llamado se tiene que hacer desde Occam.

Al ejecutarse la instrucción CALL el sistema automáticamente crea un subespacio de trabajo. El sistema reserva cuatro palabras, una para la dirección de regreso y tres para salvar la pila .

La instrucción RET devuelve este espacio.

4.7.2.- RECURSION.

La recursión directa no está permitida en Occam, por este motivo, no se permiten funciones de este tipo en el compilador de Small C, que genera código para el Transputer.

Otro punto que está ligado a la recursión, es el llamado de funciones por medio de un apuntador, que tampoco se implementó debido a que no se permite manipular desde GUY las

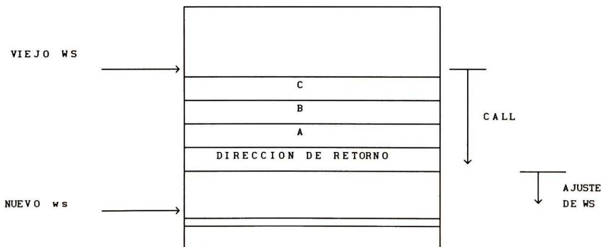


FIGURA 4.2. ESPACIO Y SUBESPACIO EN UN LLAMADO DE UN PROCEDIMIENTO

instrucciones de ajuste general del WS (Workspace) y el CALL generalizado.

4.7.3.-UNA FUNCION COMO FACTOR EN UNA EXPRESION.

Occam no tiene implementadas funciones, por lo tanto al ser llamada una función no salva la pila y esto es necesario si se está evaluando una expresión.

Este problema se superó declarando en cada función un arreglo de dos variables enteras llamado pila.

Se implementa el protocolo de salvar la pila antes del llamado y la recuperación de los valores al regreso.

Para ilustrar lo anterior suponga el siguiente programa de entrada:

```
main(){
  int x,y;

  x = 4;
  y = uno(x) * x;
}
uno(a) int a;{
return a;
}
```

La salida obtenida es :

```
PROC main.program(CHAN OF ANY from.filer, to.filer)
  INT x,y:
  [1]INT pila:  (Declaración de variables para la pila,
                debido a que no se puede manipular el
                apuntador al espacio de trabajo).
  [1]INT tempo:
  PROC uno(INT a)
    SEQ
      GUY
```

```
        LDL 1
        LDNL 0
  :
  SEQ
    GUY
      LDC 4
      STL 3
      LDLP 2
      STL 1   (Salva la pila con un registro ocupado)
    uno(x)
    GUY
      LDL 1    (Recupera la pila)
      REV
      STL 0 (Salva de la pila en una variable temporal)
      LDLP 3
      LDNL 0
      LDL 0 (Recupera el valor salvado)
      STNL 0 (almacena relativo al tope de la pila)
:
```

En ensamblador este código se genera así:

Para la función "main":

```
      AJW -4
      LDC 4
      STL 3
      LDLP 2
      STL 1   (Salva la pila con un registro ocupado)
      LDLP 3
      CALL {OFFSET DE LA FUNCION}
      LDL 1    (Recupera la pila)
      REV
      STL 0 (Salva de la pila en una variable temporal)
      LDLP 3
```

```
LDNL 0
LDL 0 (Recupera el valor salvado)
STNL 0 (almacena relativo al tope de la pila)
AJW 4
RET
```

4.7.4.-LAS VARIABLES TEMPORALES PARA EVALUAR EXPRESIONES.

Como no se puede ajustar el apuntador al espacio de trabajo, algunas de las tareas que se tuvieron que agregar al programa fueron: la de calcular el número de variables temporales, calcular los desplazamientos de dichas variables y al final hacer su declaración en el programa en un arreglo de enteros llamado **tempo.**

La declaración de las variables es lo que permite manipular el apuntador al espacio de trabajo.

Así, dado el programa:

```
main(){
   int f;
   f = 33;
   f = (f * (f - ( f + (f + 5))))
}
```

Se obtiene la salida :

```
PROC main.program(CHAN OF ANY from.filer,to.filer)
  INT f:
  [4]INT tempo:
  SEQ
    GUY
      LDC 33
```

```
STL 4
LDL 4
LDL 4
STL 0 (Salva el tope en la pila en una temporal)
LDL 4
STL 1 (Salva el tope de la pila en otra
         temporal)
LDL 4
STL 2  (Salva el tope de la pila)
LDL 4
STL 3  (Salva el tope de la pila)
LDC 5
LDL 3 (Recupera el valor salvado en 3)
ADD
LDL 2 (Recupera el valor salvado en 2)
ADD
LDL 1 (Recupera el valor salvado en 1)
REV
SUB
LDL 0 (Recupera el valor salvado en 0)
MUL
STNL 0
:
```

Este código, no es el óptimo, pues se puede generar sin utilizar las variables temporales. Lo anterior se puede realizar reordenando el árbol de la expresión. Sin embargo, el trabajo de modificación del orden de evaluación no se implementó.

El código generado al ensamblador sería el mismo, pero en lugar de hacer la declaración de variables se hace un ajuste del WS en cinco unidades al inicio y al fin del procedimiento.

4.7.5.- LAS VARIABLES DEL SWITCH.

Para compilar un "SWITCH" es necesario contar con variables en las que se almacenan los selectores. También se declara un arreglo con las variables de tipo selector en las funciones que lo requieren.

Dado el esqueleto del programa:

```
main(){
     int ...
     char ...

     switch( x){
     case
      .
      .
     }
}
```

La salida de la compilación tiene el esqueleto:

```
PROC main.program(CHAN OF ANY from.filer,to.filer)
  {variables locales}
  [1]INT selector:
  SEQ
    GUY
      Código en linea
:
```

4.8.- ESTRUCTURAS DE DATOS UTILIZADAS PARA DAR FORMATO A LA SALIDA Y HACER CALCULO DE DESPLAZAMIENTOS.

Las estructuras de datos utlizadas para hacer los cambios al compilador son en su mayor parte arreglos, ya que Small C no acepta estructuras.

4.8.1.- ARREGLO PARA VARIABLES TEMPORALES.

Para asignar las variables temporales en la evaluación de expresiones, se usa este arreglo para marcar las variables ocupadas.

Con su ayuda se contabiliza el número de variables temporales utilizadas antes de declarar las variables temporales en el inicio de una función.

4.8.2.- VARIABLES TEMPORALES.

La tabla VARTAB está implementada por medio de un arreglo con los siguientes campos:

- IDENT — Contiene el identificador del tipo de símbolo de que se trata.
- DESPL — Este campo guarda el desplazamiento del símbolo con respecto al WS.
- OFF — Contiene el desplazamiento con respecto al BP de 8086.
- MED — Este contiene la medida del objeto, del cual se calcula el desplazamiento.
- NOM — El nombre de la variable se encuentra en este campo.

Esta tabla se utiliza para recalcular los desplazamientos de las variables antes de generar código.

4.8.3.- ARREGLO PARA VARIABLES GLOBALES.

Se tienen dos arreglos para las variables globales. Uno para guardar sus direcciones en la tabla de símbolos y otro arreglo con el nuevo deplazamiento relativo al WS.

El objetivo de estas estructuras es por un lado permitir su escritura en la sección de declaraciones y por otro

calcular el desplazamiento para su acceso desde programa.

4.8.4.- ALMACENES DE CODIGO GENERADO.

Se implementaron dos estructuras de datos, parecidas a las del almacén de optimización. Son arreglos de enteros en donde se avanza el apuntador a la siguiente entrada de dos en dos.

Recordemos que la tabla de optimización tiene para cada entrada dos lugares, una para el pseudocódigo y otra para el valor.

El primer almacén se utiliza para optimización y los dos restantes para las siguientes funciones:

-- **ALMACEN1.**

En este arreglo se vacía el código que ha sido optimizado, y es aquí donde se hace el recálculo de los desplazamientos y posteriormente se envía hacia el archivo de salida.

--**ALMACEN2.**

En este almacén se guarda el codigó de la función "main", si existe, para generarlo hasta el final. Lo anterior se debe a que en Occam el programa principal anida las funciones a las que llama.

La mayor parte de las estructuras diseñadas tienen el fin de ajustarse al formato de un archivo Occam.

De igual forma se desarrollaron funciones con las finalidades antes mencionadas, formateo y cálculo de desplazamientos.

4.9.- EL ESPACIO DE TRABAJO DE UN PROCEDIMIENTO PARTICULAR.

El espacio de trabajo de un procedimiento en particular se puede visualizar de la siguiente manera:

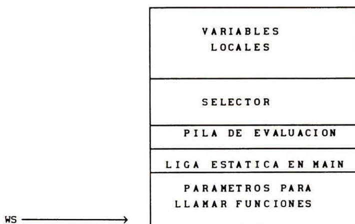


FIGURA 4.3. EL ESPACIO LOCAL DE UN PROCEDIMIENTO.

4.10.- LAS CADENAS DE OPTIMIZACION.

Con referencia a las cadenas de optimización solo se escribieron 8 y con su uso se disminuye el código entre 10% y 15%.

Las reglas son las siguientes:

```
* seq00[] = {0,POINTs,GETwn,SWAP,PUTwp,0,
            go|p2,gc|m1,gv|m1,go|p1,PUTwm,gv|m3,go|m1,0}
```

Para secuencias de la forma:

```
LDLP  n
LDC   k
REV
STNL  0
```

Mediante esta regla se obtiene una expresión más simple:

```
LDC  k
STL  n
```

```
* seq01[] = {0,POINTs,POINTs,GETwp,SWAP,PUTwp,0,
          go|p3,GETws,gv|m2,go|p1,PUTwm,gv|m4,go|m1,0}
    Secuencias de la forma:
               LDLP n
               LDLP m
              LDNL 0
               REV
               STNL 0
    Se simplifican a:
               LDL m
               STL n

* seq02[] = {0,POINTs,GETwp,0,
               go|p1,GETws,gv|m1,0}
    La secuencias del tipo:
               LDLP n
               LDNL 0
    Se simplifican a:
               LDL n

* seq03[] = {0,POINTs,GETws,SWAP,PUTwp,0,
          go|p2,gc|m1,gv|m1,go|p1,PUTwm,gv|m3,go|m1,0}

    Mediante esta regla, las secuencias de la forma:
               LDLP n
               LDL m
               REV
               STNL 0
     Se simplifican a:
               LDL m
               STL n
```

```
*  seq04[] = {0,SWAP,SWAP,0,
              go|p2,0}
```

Con las simplificaciones anteriores, frecuentemente quedan secuencias de la forma:

```
REV
REV
```

Y se eliminan, aplicando esta regla.

```
* seq05[] = {0,POINTs,GETwn,SWAP,DIR_W,0,
             sum|p1,sum|p1,go|p1,gv|m1,go|p2,
             POINTs,gv|m2,0}
```

Secuencias de la forma:

```
LDLP n
LDC  k
REV
WSUB
```

Se simplifican a:

```
LDLP n + k
```

```
* seq06[] = {0,POPm,SWAP,0,
             go|p1,gc|m1,gv|m1,0}
```

Las secuencias de la forma:

```
POP
REV
```

Esta regla se usa como una consecuencia de salvar la dirección donde se almacenará una cantidad, o al calcular una dirección.

Por ejemplo:

```
LDL 0   {pop 0}
REV
```

```
STNL 0
```

Al hacer el POP, lo que se carga es la dirección y si se ejecuta el SWAP la dirección pasa al registro B de la pila y el resultado sería un almacenamiento erróneo.

Se simplifican a:

```
LDL 0
STNL 0
```

```
* seq07[] = {0,PUSH,any,POPm,0,
             neg,sum|p2,ife|0,go|p2,gc|m1,gv|m1,0,
             go|m2,gv|p2,0,0}
```

Para secuencias como:

```
PUSH tempo
Cualquier Código
POP tempo
```

Se obtiene la simplificación:

```
Cualquier Código.
```

Con está regla se optimizan variables temporales, pues cuando se salva una variable y solo hay otra instrucción de cargado que precede al POP, se puede eliminar el uso de está variable temporal.

Por ejemplo:

```
STL 0 {PUSH}
LDC 5
LDL 0 {POP}
```

Se obtiene:

```
LDC 5
```

CAPITULO 5

RESULTADOS

Como se ha mencionado en un principio, comparar arquitecturas es muy difícil, pues se tienen que considerar muchos parámetros.

En esta comparación solo se consideran dos parámetros:

--Número de lineas generadas.

--El número de bytes que ocupa el código.

La máquina PC en que se hizo la corrida de los programas tiene un reloj de 8 MHz.

El transputer trabaja a 20 MHz.

Los programas que se utilizaron en la comparación fueron un total de 5.

* Un programa que hace cálculos con arreglos.
* Un programa para calcular los números primos hasta cierto número.
* Un programa con el SORT de la burbuja.
* Un programa con el método de ordenación de selección.
* El programa de búsqueda binaria.

Los programas tienen muchas iteraciones y cálculos aritméticos, así como el acceso de elementos de un arreglo.

Los programas se compilaron en Small C para PC y en Small C para el Transputer en su forma con optimización y sin optimización.

En lo que resta de este capítulo se presentan las tablas comparativas para estas compilaciones y en el siguiente capítulo se presentan las conclusiones del trabajo.

Se utiliza la abreviatura SO, para indicar sin optimización y TR para transputer.

La primera tabla que se presenta es la que se refiere a elnúmero de lineas generadas en cada una de las compilaciones.

PROGRAMA	PC	PC/SO	TR	TR/SO
ARITMETICA	274	442	332	377
PRIMOS	115	185	156	168
BURBUJA	178	284	246	305
SORT	162	273	227	285
BINARY	133	221	178	224

FIGURA 5.1 TABLA DE COMPARACION DE LINEAS GENERADAS

La segunda tabla comparativa que se presenta es la que se refiere al número de bytes del programa.

PROGRAMA	PC	PC/SO	TR	TR/SO
ARITMETICA	639	913	332	377
PRIMOS	244	366	156	168
BURBUJA	309	383	246	305
SORT	353	545	227	285
BINARY	294	450	178	224

FIGURA 5.2. TABLA COMPARATIVA DEL CODIGO GENERADO EN BYTES

Otro parámetro de comparación es el algoritmo de compilación y se puede ver que debido a que en el transputer hay pocos modos de direccionamiento y menos instrucciones, es más sencillo generar código que para el procesador 8086.

La comparación de velocidades de ejecución parece un poco imprecisa, debido a la diferencia de relojes con que trabajan las máquinas.

Existe una relación de 1 a 2.5 entre los relojes de ambas máquinas.

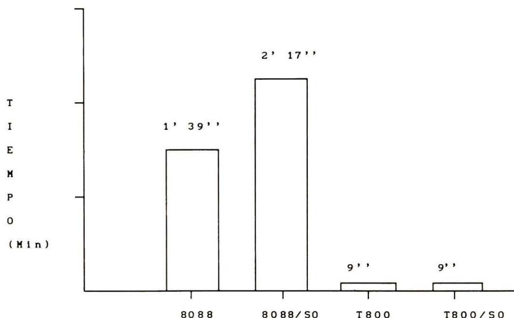


FIGURA 5.3. TIEMPO DE EJECUCION DE PROGRAMA DE CALCULOS ARITMETICOS

Las velocidades de ejecución de dos de los programas corriendo en ambas máquinas se presentan enseguida..

El programa de cálculos aritméticos se ejecutó así:

PC	PC/SO	T	T/SO
1 min. 39 seg.	2 min 17 seg	9 seg 28cent	9 seg 30 cent.

La gráfica comparativa se muestra en la figura 5.3.

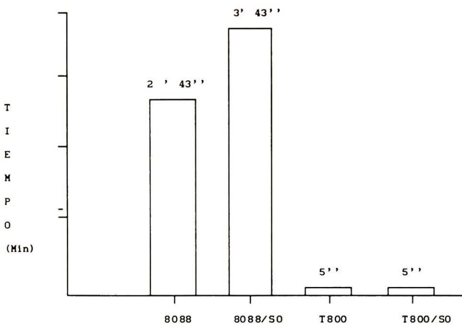


FIGURA 5.4. TIEMPO DE EJECUCION DE EL PROGRAMA DE NUMEROS PRIMOS.

El programa de encontrar los primos hasta el número 10000 se ejecutó así:

PC	PC/SO	T	T/SO
2 min. 43 seg.	3 min 43 seg	5 seg 74 cent	5 seg 74 cent.

La gráfica 5.4 muestra los tiempos de ejecución de este programa.

La rapidez de ejecución del transputer es 10 veces mayor que la de 8086, aunque se debe tomar en cuenta el número de bytes y los relojes de ambas máquinas.

Este resultado, aún tomando en cuenta los parámetros que se han pasado por alto da una idea de la velocidad de proceso de esta máquina de INMOS.

En lo que se refiere a la optimización, tomando el número de lineas se tiene:

En 8086 un promedio de 531 lineas sin optimizar y 367 optimizando, lo que da un 31% de optimización con 47 secuencias de optimización.

En Transputer se obtuvo un promedio de 263 lineas sin optimizar contra 227 optimizando, lo que da un total de 14% de código ahorrado con solo ocho reglas.

CAPITULO 6

CONCLUSIONES

Surge en este momento la pregunta sobre que sería mejor, si una arquitectura con un complejo conjunto de instrucciones y muchos registros o una arquitectura simple con un conjunto reducido de instrucciones y ventanas de registros.

Las arquitecturas tipo RISC requieren de algoritmos complicados de compilación, en cambio para las máquinas de tipo pila se requieren algoritmos más simples.

Wirth menciona que las nuevas características de una máquina debe resolver problemas, no crearlos y opina que una máquina debe tener un conjunto regular y completo de instrucciones en lugar de un conjunto reducido de éstas.

El transputer no es precisamente una máquina RISC, pero tiene algunas características de estas máquinas. Su diseño permite que los algoritmos de compilación escritos para esta máquina sean sencillos.

En las máquinas de tipo CISC (computadoras de complejo conjunto de instrucciones) como el M68000 se ahorran muchos accesos a memoria por el número abundante de registros, pero el cambio de contexto se vuelve lento y la escritura del compilador se complica al tener que asignar los registros.

El 8086 visto como una máquina de número limitado de registros tiene que hacer muchos accesos a memoria y su conjunto de instrucciones no es tan simple .

El propósito de este capítulo no es decir quien es mejor y quien es peor, sino comentar los resultados obtenidos en las pruebas realizadas con los dos compiladors.

La arquitectura de 8086 ha sido comparada con la arquitectura del Transputer, una máquina de pila. La mayor complejidad del conjunto de instrucciones del 8086 y sus modos de direccionamiento, se reflejan en el algoritmo de compilación, ya que se puede observar que las diferencias entre el número de pseudócodigos en uno y otro compilador es de 40 % .

Lo anterior se debe a que Transputer tiene menos instrucciones de las cuales seleccionar, menos modos de direccionamiento y las instrucciones de movimientos entre registros son mínimas.

La densidad de código, medida en bytes, obtuvo mejores resultados en el Transputer, aunque el número de líneas es mayor en los programas compilados a Occam.

Al medir los tiempos de ejecución de los dos programas de prueba, se puede observar una diferencia muy desequilibrada a favor del transputer.

Al concluir este trabajo no creo conveniente afirmar que el transputer es la mejor máquina de las comparadas, más bien es la que obtuvo mejores resultados en las pruebas realizadas.

El transputer posee una arquitectura que permite fácilmente escribir compiladores y que produce un código muy denso que se ejecuta a una alta velocidad.

Los problemas que se encontraron al generar código más bien se deben a las restricciones impuestas por Occam, por lo que está arquitectura sería más aprovechable, desde el punto de vista de escritura de compiladores, si se contara con un ensamblador.

REFERENCIAS

1.-Aho A., Sethi R. and Ullman J.D., Compilers Principles, Techniques and Tools, (1986), Addison-Wesley Publishing Company 1986.

2.-Burns, Alan.Programming In Occam2 (Addison Wesley).

3.-Hamblen, J., O. and Parker, A., Parallel processing on the transputer using OCCAM, Journal of Microcomputer Applications (1989) 12, 1-14.

4.-Hendrix James, A SMALL C COMPILER, M&T Books, 1988.

5.-Hennsey John L., VLSI Processor Architecture, IEEE Transactions On Computers, Vol. c-33,No. 12, (diciembre 1984), 1221-1246.

6.-Homewood Mark and May David, The INMOS T800 Transputer, IEEE Micro (octubre 1987), 10-26.

7.-INMOS, OCCAM2 STANDALONE COMPILER OWNER'S MANUAL .

8.-INMOS, TRANSPUTER INSTRUCTION SET a compiler writer's guide.

9.-Janecek Jan, Register Array Processor, Microprocessing and Microprogramming 21 (1987) 23-30.

10.-Kernighan and Ritchie, El Lenguaje de Programación C, Prentice-Hall Hispanoamericana.(1986).

11.-May David, OCCAM, SIGPLAN Notices, V18-4 (abril 1983), 69 - 79.

12.-Nicoud Jean Daniel, The Transputer T414 Instruction Set, (1989), IEEE MICRO (junio 1989) 60-75.

13.-Patterson D.A., Reduced Instruction Set Computers, Communications of the ACM 28,1 (enero 1985), 8-21.

14.-Piepho Richard, A Comparison of RISC Architectures, IEEE micro (agosto 1989), 51-62.

15.-Sethi, R. and Ullman, J.D., The Generation of Optimal Code for Arithmetic Expressions,Journal of the ACM 17,6 (Octubre 1970), 715-728.

16.-Stein Richard., T800 and Counting, Byte (noviembre 1988), 287-296.

17.-Tabak Daniel, RISC Systems, Microprocessors and Microsystems, Vol. 12-4 (mayo 1988), 179 - 185.

18.-Tommeleim A. and Tiberghien Jacques, A Comparison of Compiler-Generated Code, Microprocessing and Microprogramming 15 (1985), 47-56.

19.-Wayman Russell, OCCAM2: an overview from a software engineering perspective, Microprocessors and Microsystems (octubre 1987) Vol 11, 413-422.

20.-Wirth Niklaus. Microprocessor Architectures: A Comparison Based on Code Generation By Compiler, Communications of the ACM 29,10 (Octubre 1986), 978-990.

APENDICE A

SINTAXIS DE OCCAM 2

La siguiente, es una descripción de la sintaxis de occam 2.

En esta descripción, {proceso}, significa cero o más procesos.

{$_1$, objeto} signific uno o más objetos separados por comas.

{$_0$; objeto} significa cero o más objetos separados por punto y coma.

El símbolo | siginfica "o".

SINTAXIS.

PROCESS = SKIP | STOP | ACTION | CONSTRUCTION | BLOCK | CASE SELECTOR { SELECTION}

ACTION = ASSIGNEMENT | INPUT | CASE INPUT | OUTPUT

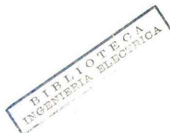
ASSIGNMENT = VARIABLE := EXPRESSION | VARIABLE LIST := EXPRESSION LIST.

VARIABLE LIST = {$_1$, VARIABLE}

EXPRESSION LIST = {$_1$, EXPRESSION } | (VALOF) | NAME($_0$, EXPRESSION))

INPUT = CHANNEL ? INPUT ITEM |
CHANNEL ? {$_1$, INPUT ITEM} |
CHANEL ? CASE tagged LIST|
TIMER ? VARIABLE |

```
                          TIMER    ? AFTER EXPRESSION |
                          PORT      ? VARIABLE.
INPUT ITEM          = VARIABLE | VARIABLE :: VARIABLE


CASE INPUT          = CHANNEL ? CASE
                          (VARIANTE)


GUARDED CASE INPUT = BOOLEAN & CHANNEL ? CASE
                                    (VARIANT)


VARIANT                        = TAGGED  LIST  PROCESS  |  SPECIFICATION
                          VARIANT


TAGGED LIST         = TAG | TAG ; {1 ; INPUT ITEM}


TAG                   = NAME


OUTPUT               = CHANNEL ! OUTPUT ITEM |
                          CHANNEL ! {1 ,OUTPUT ITEM } |
                          CHANNEL ! TAG |
                          CHANNEL ! TAG ;{1 ; OUTPUT ITEM} |
                          PORT ! EXPRESSION


OUTPUT ITEM         = EXPRESSION | EXPRESSION :: EXPRESSION


VARIABLE             = ELEMENT


CHANNEL              = ELEMENT


TIMER                 = ELEMENT


PORT                  = ELEMENT
```



```
CONSTRUCTION          = LOOP | CONDITIONAL | SEQUENCE |
                              PARALLEL | ALTERNATION

LOOP                    = WHILE BOOLEAN
                                PROCESS

BOOLEAN                = EXPRESSION

CONDITIONAL           = IF
                                { CHOICE }
                            | IF replicator
                                CHOICE
CHOICE                  = GUARDED CHOICE | CONDITIONAL |
                              SPECIFICATION | CHOICE

GUARDED CHOICE       = BOOLEAN
                              PROCESS

REPLICATOR             = NAME = BASE FOR COUNT

BASE                    = EXPRESSION

COUNT                   = EXPRESSION

SEQUENCE               = SEQ
                               {PROCESS}
                            | SEQ REPLICATOR
                                PROCESS

PARALLEL               = PAR
                               {PROCESS}
                           | PAR REPLICATOR
                                PROCESS
```

```
                        | PLACED PAR
                             {PLACEMENT}
                        | PRI PAR
                             {PROCESS}
                        | PRI PAR REPLICATOR
                             PROCESS

PLACEMENT             = PROCESSOR EXPRESSION
                            PROCESS

ALTERNATION           = ALT
                            {ALTERNATIVE }
                         | ALT REPLICATOR
                              ALTERNATIVE
                         | PRI ALT
                              {ALTERNATIVE}
                         | PRI ALT REPLICATOR
                              ALTERNATIVE

ALTERNATIVE           = GUARDED ALTERNATIVE | ALTERNATION |
                            CASE INPUT | GUARDED CASE INPUT
                            SPECIFICATION | ALTERNATIVE

GUARDED ALTERNATIVE= GUARD
                             PROCESS

GUARD                   = INPUT
                         | BOOLEAN & INPUT | BOOLEAN & SKIP

BLOCK                   = SPECIFICATION |
                           SCOPE
                         | ALLOCATION
                            SCOPE
```

ESPECIFICATION = DECLARATION | ABBREVIATION |DEFINITION

SCOPE = PROCESS

DECLARATION = TYPE NAME:

TYPE = PRIMITIVE TYPE | PORT OF TYPE | ARRAY TYPE | RECORD TYPE

PRIMITIVE TYPE = CHAN OF PROTOCOL
| TIMER |BOOL | BYTE | INT | REAL32 |
REAL64 | REAL | INT16 | INT32 |INT64

PROTOCOL = NAME
| SIMPLE PROTOCOL | ANY

ARRAY TYPE = [EXPRESSION] TYPE

RECORD TYPE = (({ $_1$, TYPE})

ABBREVIATION = SPECIFIER NAME IS ELEMENT: |
VAL SPECIFIER NAME IS EXPRESSION:

SPECIFIER = PRIMITIVE TYPE |
[EXPRESSION]SPECIFIER
[]SPECIFIER

DEFINITION = SPECIFIER NAME RETYPES ELEMENT:|
VAL SPECIFIER NAME RETYPES EXPRESSION:
| TYPE NAME IS TYPE
| RECORD NAME IS RECORD TYPE
| PROC NAME (({ $_0$,FORMAL})
BODY
:

```
                    |{1,TYPE} FUNCTION NAME ({0,FORMAL})
                      IS EXPRESSION LIST: |
                    {1,TYPE } FUCTION NAME ({ 0 , FORMAL})
                              FUNCTION BODY
                    :
                    | PROTOCOL NAME IS SIMPLE PROTOCOL:
                    |       PROTOCOL      NAME      IS      SEQUENTIAL
                      PROTOCOL:
                    | PROTOCOL NAME
                        CASE
                          {TAGGED PROTOCOL}
                        :
FORMAL             = SPECIFIER NAME |VAL SPECIFIER NAME

BODY               = PROCESS

FUCTION BODY       = VALOF

VALOF              = VALOF
                         PROCESS
                         RESULT EXPRESSION LIST
                     | SPECIFICATION
                       VALOF

SIMPLE PROTOCOL    = TYPE | TYPE ::[]TYPE

SEQUENTIAL
      PROTOCOL     = { 1; SIMPLE PROTOCOL}

TAGGED PROTOCOL    = TAG | TAG ; PROTOCOL

ALLOCATION         = PLACE NAME AT EXPRESSION:

INSTANCE           = NAME({0, ACTUAL})
```

ACTUAL = ELEMENT | EXPRESSION

SELECTOR = EXPRESSION

SELECTION = EXPRESSION

PROCESS

| ELSE

PROCESS

ELEMENT = ELEMENT[SUBSCRIPT] |

[ELEMENT FROM SUBSCRIPT FOR SUBSCRIPT]

| [{, ELEMENT}] | ({, ELEMENT})

| NAME

SUBSCRIPT = EXPRESSION

EXPRESSION = MONADIC.OPERATOR OPERAND

| OPERAND DYADIC.OPERATOR OPERAND

| CONVERSION | MOSTPOS TYPE |

| MOSTNEG TYPE

OPERAND = ELEMENT | LITERAL |

[$\{_1$,EXPRESSION}] |

($\{_1$,EXPRESSION})| (EXPRESSION) |

NAME (($_0$, EXPRESSION)) |(VALOF)

LITERAL = INTEGER | BYTE | INTEGER (TYPE) |

BYTE (TYPE) | REAL (TYPE) | STRING

TRUE | FALSE

INTEGER = DIGITS | #DIGITS

BYTE = 'CHARACTER'

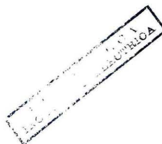
```
REAL                    = DIGITS.DIGITS |
                              DIGITS.DIGITSEexponent
exponent                = +DIGITS | -DIGITS

CONVERSION              = TYPE OPERAND | TYPE ROUND OPERAND
                              TYPE TRUNC OPERAND

MONADIC.OPERATOR        = - | NOT | SIZE

DYADIC.OPERATOR         = + | - | * | / | REM | PLUS | MINUS|
                            TIMES | /\ | \/| >< | >> | << | ~ |
                             AND| OR | = | <> | < | > | <= | >=
```

APENDICE B

LISTADO DEL CODIGO GENERADO POR UN PROGRAMA PARA ARITMETICA DE ENTEROS.

```
#define MAX 1000
#define TAM 200
main(){
   int x[TAM],y[TAM],z[TAM],m[TAM];
   int a,s,d;
   int i,j,k,l;
   for(i = 0 ; i < 100 ; i++){
      x[i] = i + 12;
      y[i] = i * 32;
      z[i] = i - 66;
      m[i] = i * 4;
   }
   for(i = 100 ; i < TAM ; i++){
      x[i] = i * 11;
      y[i] = i - 32;
      z[i] = i + 66;
      m[i] = i / 4;
   }
   for (i = 0 ; i < MAX ; i++)
      for(j = 0; j < TAM ; j++){
         x[j] = y[j] + z[j] - m[j];
         y[j] = x[j] * z[j] * m[j];
         z[j] = x[j] * m[j]   y[j];
         m[j] = x[j]   z[j] + y[j];
      }
}

program user.program(CHAN OF ANY from.filer,to.filer)
   [200]INT x:
   [200]INT y:
   [200]INT z:
   [200]INT m:
   INT a:
   INT s:
   INT d:
   INT i:
   INT j:
   INT k:
   INT l:
   [3]INT tempo:
   SEQ
      GUY
         LDLP 6
         LDC 0
         REV
```

```
STNL 0
:E4
LDLP 6
LDNL 0
LDC 100
REV
GT
CJ .E3
J .E5
:E2
LDLP 6
DUP
LDNL 0
ADC 1
STNL 0
LDLP 6
LDNL 0
ADC -1
J .E4
:E5
LDLP 610
STL 0
LDLP 6
LDNL 0
LDL 0
WSUB
STL 0
LDLP 6
LDNL 0
LDC 12
ADD
LDL 0
STNL 0
LDLP 410
STL 0
LDLP 6
LDNL 0
LDL 0
WSUB
STL 0
LDLP 6
LDNL 0
LDC 32
MUL
LDL 0
STNL 0
LDLP 210
STL 0
LDLP 6
LDNL 0
LDL 0
WSUB
STL 0
LDLP 6
```



```
LDNL  0
LDC  66
REV
SUB
LDL  0
STNL  0
LDLP  10
STL  0
LDLP  6
LDNL  0
LDL  0
WSUB
STL  0
LDLP  6
LDNL  0
LDC  4
MUL
LDL  0
STNL  0
J  .E2
:E3
LDLP  6
LDC  100
REV
STNL  0
:E8
LDLP  6
LDNL  0
LDC  200
REV
GT
CJ  .E7
J  .E9
:E6
LDLP  6
DUP
LDNL  0
ADC  1
STNL  0
LDLP  6
LDNL  0
ADC  -1
J  .E8
:E9
LDLP  610
STL  0
LDLP  6
LDNL  0
LDL  0
WSUB
STL  0
LDLP  6
LDNL  0
LDC  11
```

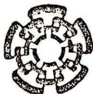
```
MUL
LDL  0
STNL  0
LDLP  410
STL  0
LDLP  6
LDNL  0
LDL  0
WSUB
STL  0
LDLP  6
LDNL  0
LDC  32
REV
SUB
LDL  0
STNL  0
LDLP  210
STL  0
LDLP  6
LDNL  0
LDL  0
WSUB
STL  0
LDLP  6
LDNL  0
LDC  66
ADD
LDL  0
STNL  0
LDLP  10
STL  0
LDLP  6
LDNL  0
LDL  0
WSUB
STL  0
LDLP  6
LDNL  0
LDC  4
REV
DIV
LDL  0
STNL  0
J  .E6
:E7
LDLP  6
LDC  0
REV
STNL  0
:E12
LDLP  6
LDNL  0
LDC  1000
```

```
REV
GT
CJ .E11
J .E13
:E10
LDLP 6
DUP
LDNL 0
ADC 1
STNL 0
LDLP 6
LDNL 0
ADC -1
J .E12
:E13
LDLP 5
LDC 0
REV
STNL 0
:E16
LDLP 5
LDNL 0
LDC 200
REV
GT
CJ .E15
J .E17
:E14
LDLP 5
DUP
LDNL 0
ADC 1
STNL 0
LDLP 5
LDNL 0
ADC -1
J .E16
:E17
LDLP 610
STL 0
LDLP 5
LDNL 0
LDL 0
WSUB
STL 0
LDLP 410
STL 1
LDLP 5
LDNL 0
LDL 1
WSUB
LDNL 0
STL 1
LDLP 210
```

```
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
ADD
STL 1
LDLP 10
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
REV
SUB
LDL 0
STNL 0
LDLP 410
STL 0
LDLP 5
LDNL 0
LDL 0
WSUB
STL 0
LDLP 610
STL 1
LDLP 5
LDNL 0
LDL 1
WSUB
LDNL 0
STL 1
LDLP 210
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
MUL
STL 1
LDLP 10
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
```

```
MUL
LDL 0
STNL 0
LDLP 210
STL 0
LDLP 5
LDNL 0
LDL 0
WSUB
STL 0
LDLP 610
STL 1
LDLP 5
LDNL 0
LDL 1
WSUB
LDNL 0
STL 1
LDLP 10
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
MUL
STL 1
LDLP 410
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
REV
SUB
LDL 0
STNL 0
LDLP 10
STL 0
LDLP 5
LDNL 0
LDL 0
WSUB
STL 0
LDLP 610
STL 1
LDLP 5
LDNL 0
LDL 1
WSUB
LDNL 0
```

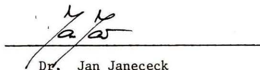
```
STL 1
LDLP 210
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
REV
SUB
STL 1
LDLP 410
STL 2
LDLP 5
LDNL 0
LDL 2
WSUB
LDNL 0
LDL 1
ADD
LDL 0
STNL 0
J .E14
:E15
J .E10
:E11
```



CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL I.P.N.

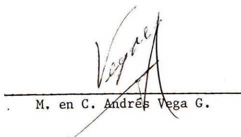
Ave. Instituto Politécnico Nacional 2508 Col. San Pedro Zacatenco
México, D.F., C.P. 07300

EL JURADO DESIGNADO POR LA SECCION DE COMPUTACION DEL DEPARTAMENTO DE INGENIERIA ELECTRICA, APROBO EL DIA 26 DEL MES DE JUNIO DEL AÑO DE 1990 EL TRABAJO DE TESIS ESTUDIO COMPARATIVO DE ARQUITECTURAS DE PROCESADORES CON BASE EN LA GENERACION DE CODIGO DESARROLLADO POR EL ALUMNO: ING. OCTAVIO HECTOR JUAREZ ESPINOZA



Dr. Jan Janececk

Dr. Adriano de Luca P.



M. en C. Andrés Vega G.

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro antes del vencimiento de préstamo señalado por el último sello.

DEVOLUCION

10110-128/90 L

AUTOR JUAREZ ESPINOSA, O.H.

TITULO ESTUDIO COMPARATIVO DE AR-
QUITECTURAS DE PROCESADORES

CLASIF. XM
90.8 **RGTRO.** BI-11736

NOMBRE DEL LECTOR	FECHA PREST.	FECHA DEVOL.
Dr. Guillermo Morales	5-12-95	5/XII/95 4.