Departamento de Arquitectura Instituto de Computación

Universidad de la República

Montevideo - Uruguay

Notas de Teórico

Organización del CPU

Arquitectura de Computadoras

(Versión 4.3a - 2012)

14 ORGANIZACIÓN DE LA UNIDAD CENTRAL DE PROCESO (CPU)

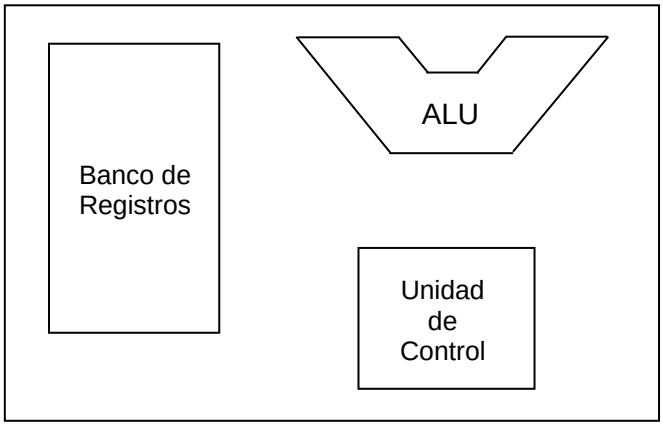
14.1 Introducción

En este capítulo veremos un posible diseño interno de una unidad central de proceso de una arquitectura de von Neumann, incluyendo sus variaciones de "lógica cableada" y "lógica microprogramada".

Presentaremos como ejemplo la organización del procesador de la arquitectura MIC-1, desarrollada por Andrew S. Tanenbaum con fines didácticos en su libro *Structured Computer Organization* (Organización de Computadoras, un Enfoque Estructurado), en particular para su 3ra Edición.

14.2 Componentes Básicos de la CPU

Como ya dijimos antes una CPU (Centrol Processor Unit) está conformada por tres sub-sistemas fundamentales: la ALU (Arithmetic Logic Unit, Unidad Aritmética y Lógica), la CU (Control Unit, Unidad de Control) y el Register Set (también denominado Register Bank) ó sea Conjunto (o Banco) de Registros.



El **Banco de Registros** contiene registros de tres categorías desde el punto de vista de su función en relación con los programas y el funcionamiento interno de la CPU:

- Totalmente visibles: estos son los ya mencionados registros de uso general ó personalizados que contienen operandos o direcciones para su utilización en las instrucciones. El programador de “bajo nivel” los manipula directamente en los programas.
- Parcialmente visibles: son registros que tienen funciones especiales pero participan de algún modo indirecto en las instrucciones. El programador los manipula indirectamente en determinadas instrucciones específicas. Ejemplos de este tipo de registros son el **IP** (**I**nstruction **P**ointer), también denominado **PC** (**P**rogram **C**ounter) que contiene la dirección de la próxima instrucción a ejecutarse (en algunas arquitecturas almacena la dirección de la que se está ejecutando en este momento), el **SP** (**S**tack **P**ointer) que contiene el puntero al primer lugar de la pila en las arquitecturas “de stack” y el **PS** (**P**rocessor **S**tatus) también denominado registro de **FLAGS** (en el caso de Intel) que contiene el estado del procesador incluyendo el valor que tomaron los bits de condición (**N**egative, **Z**ero, **C**arry, **O**verflow) en función del resultado de la última operación realizada por la ALU.
- Internos: son registros que utiliza la Unidad de Control de la CPU para poder

ejecutar las instrucciones. Almacenan constantes, el estado de la CU, la instrucción en ejecución (su código binario), resultados intermedios de cálculos de direcciones, etc. No son visibles de ninguna manera al programador.

La **Unidad de Control** es, en definitiva, una máquina secuencial que realiza el "ciclo de instrucción": conjunto de acciones ordenado y secuencial que interconectan adecuadamente los distintos elementos en el tiempo, para lograr el objetivo de ejecutar la instrucción realizando la operación indicada sobre los operandos correspondientes y almacenando el resultado en el lugar indicado. Esta máquina secuencial funciona sincronizada por un reloj, el cual también es utilizado para sincronizar todas las actividades de los otros elementos del sistema (memoria y entrada/salida). En las primeras computadoras el reloj era el mismo para todos los elementos. Últimamente se utilizan relojes independientes (aunque vinculados) para cada sub-sistema. En muchos diseños se utilizan más de un reloj para la CPU, con la misma frecuencia, pero desfasados (0º, 90º, 180º y 270º, por ejemplo) a los efectos de ser utilizados para sincronizar distintas partes del circuito compensando los diferentes retardos de propagación de las señales en los circuitos internos de la CPU.

La **Unidad Aritmética y Lógica** es un conjunto de circuitos (típicamente combinatorios) que implementan un conjunto de operaciones, que incluyen suma y resta (en aritmética complemento a 2), operaciones lógicas bit a bit (AND, OR, EXOR, NOT) y operaciones de desplazamiento (shift). Las ALUs más avanzadas incluyen operaciones de multiplicación y división (aunque en este caso se implementan como una máquina secuencial que implementa algún algoritmo para estas operaciones).

14.3 Ciclo de Instrucción

Se denomina ciclo de instrucción a la secuencia de acciones que realiza la CPU (más específicamente la Unidad de Control) para lograr ejecutar una instrucción del programa almacenado en memoria.

Un ciclo de instrucción típico tiene 5 pasos característicos:

- **Fetch**: este paso consiste en leer la próxima instrucción a ejecutarse desde la memoria.
- **Decode**: en este paso se analiza el código binario de la instrucción para determinar qué se debe realizar (cuál operación, con qué operandos y donde guardar el resultado)
- **Read**: en este paso se accede a memoria para traer los operandos
- **Execute**: es la ejecución de la operación por parte de la ALU sobre los operandos
- **Write**: en el último paso se escribe el resultado en el destino indicado en la instrucción.

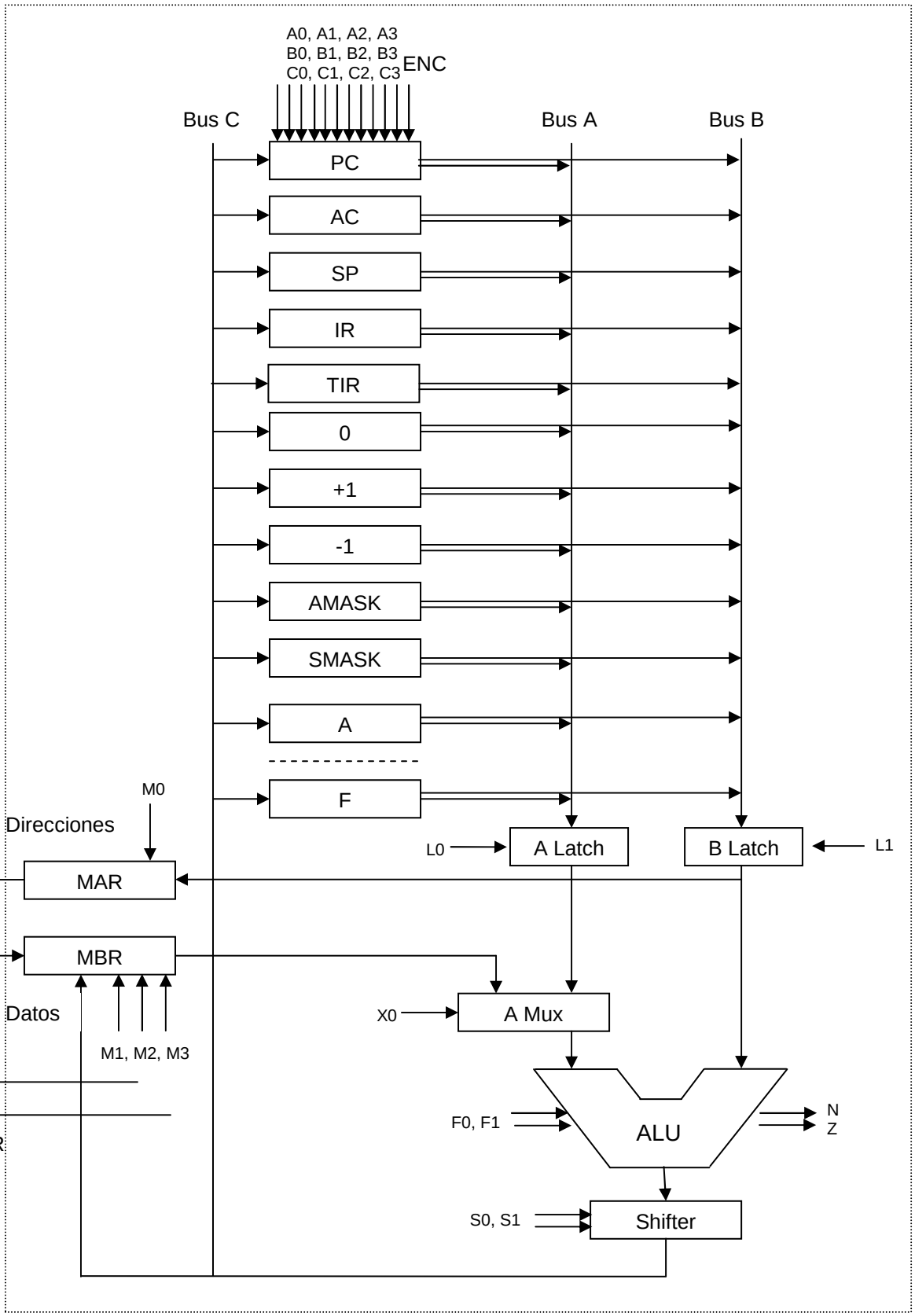
Notemos que no todas las instrucciones requieren de todos los pasos indicados para su ejecución. Por ejemplo las instrucciones que tienen sus operandos en registros, no requieren del paso "read", mientras que las que no guardan un resultado no requieren del paso "write".

Para realizar estos pasos la unidad de control maneja un conjunto de señales de control interno a la CPU y otras de control externo (en la interfaz con la memoria y con el sistema de entrada/salida), las que habilitan las conexiones apropiadas entre los distintos elementos de forma que el respectivo paso del ciclo se cumpla. Por ejemplo: para hacer el "fetch" conecta el bus de direcciones de memoria con el registro PC (que contiene la dirección de la próxima instrucción a ejecutarse), conecta el bus de datos de memoria con

el registro interno IR (Instruction Register) y coloca en el bus de control de la memoria las señales apropiadas para realizar una lectura.

14.4 Estructura de la CPU

En el siguiente diagrama se muestra la estructura de una CPU MIC-1.



En esta propuesta de organización interna se dispone de los siguientes recursos (es una arquitectura de 16 bits):

- Registro PC: contiene el puntero de instrucción (“Program Counter”)
- Registro AC: es el registro ACumulador
- Registro SP: contiene el puntero a la pila (“Stack Pointer”)
- Registro IR: almacena la instrucción leída desde memoria
- Registro TIR: almacena temporalmente copias de la instrucción
- Registro 0: contiene la constante 0
- Regsitro +1: contiene la constante 1 (positiva)
- Registro -1: contiene la constante -1 (negativa en complemento a 2)
- Registros AMASK: contiene el valor binario 0000111111111111 (0x0fff)
- Registros SMASK: 000000001111111 (0x00ff)
- Registros A a F: para uso interno de la Unidad de Control
- Registro MAR (Memory Address Register): almacena la dirección de memoria que se presenta en el bus de direcciones de la memoria durante una operación de lectura o escritura de la misma.
- Registro MBR (Memory Buffer Register): almacena el dato leído de la memoria (en una operación de lectura) o el dato a escribir en la memoria (durante una operación de escritura).
- AMux: multiplexor que elige entre dos entradas posibles.
- ALU: Unidad Aritmética y Lógica, capaz de ejecutar 4 operaciones: suma A+B, A and B identidad A (la salida es igual a su entrada A), not A.
- Shifter: unidad encargada de realizar la operación de desplazamiento: 0 bit (no desplaza nada), 1 bit a la derecha, 1 bit a la izquierda.
- ALatch/BLatch: son registros intermedios utilizados para simplificar los aspectos de sincronismo de las señales, requeridos porque el diseño original de Tanenbaum prevé la construcción del banco de registros con Flip-Flops asíncronos (un diseño basado en FF con reloj por flanco podría evitar el uso de estos registros intermedios).

Notas:

- La interfaz con la memoria y la E/S se realiza a través del bus de datos, el bus de direcciones y el bus de control (formado por las señales RD y WR).
- Los registros, la ALU y la unidad de Shift son de 16 bits de tamaño de palabra.
- La unidad de desplazamiento no siempre se explicita como en el ejemplo ya que se puede considerar que forma parte de la ALU.
- Normalmente las ALUs calculan también las salidas C (Carry) y V (oVerflow).
- Los valores de las constantes de los distintos registros está vinculada con las necesidades de la “macroarquitectura” asociada a la MIC-1 y no constituyen un requisito en otras microarquitecturas.
- Tener presente que este es tan solo un ejemplo de cómo se puede construir internamente una CPU y ni siquiera es uno que pueda considerarse óptimo.

Por razones de simplicidad se ha omitido dibujar la Unidad de Control en el diagrama anterior. Esta es, como dijimos, una máquina secuencial. Las salidas de este circuito secuencial son todas las señales de control indicadas en el diagrama, que actúan sobre los distintos recursos internos y el bus de control externo de manera de completar los distintos pasos del ciclo de instrucción visto.

Las funciones que cumplirían las distintas señales de control son:

- A3, A2, A1, A0: seleccionan un registro y conectan su salida al bus A (ej: 0000/PC, 0001/AC, 0010/SP, 0010/IR, 0100/TIR, 0101/0, 0110/+1, 0111/-1, 1000/AMASK, 1001/SMASK, 1010/A, 1011/B … 1111/F)
- B3, B2, B1, B0: seleccionan un registro y conectan su salida al bus B (ej: idem a la codificación para A)
- C3, C2, C1, C0: seleccionan un registro y conectan su entrada al bus C (ej: idem a la codificación para A)
- ENC: habilita el bus C para que se guarde el valor en el registro seleccionado
- X0: selecciona si conecta a su salida la entrada que viene desde el MBR o desde el bus A (ej: 0 – MBR, 1 – Bus A)
- F1, F0: codifican la operación a realizar por la ALU (ej: 00 – A+B, 01 – A and B, 10 – A, 11 – not A)
- S1, S0: codifican la operación de desplazamiento (ej: 00 – no desplaza, 01 – un bit a la derecha, 10 – un bit a la izquierda)
- L1, L0: controlan la carga de los registros intermedios del bus B y el bus A respectivamente.
- M0: controla la carga en el MAR (ej: 1 – carga)
- M1, M2, M3: controlan la forma que se cargan los datos en el MBR, cuando y desde donde se produce. M1 activa la carga (ej: 1 – carga). M2 es la señal de lectura de memoria RD (ej: 1 – lee la memoria). M3 es la señal de escritura de memoria (eg: 1 – escribe la memorias).

Veamos como manejaría la Unidad de Control estas señales de control para lograr ejecutar el ciclo de instrucción. Consideremos el ejemplo en el que se va a ejecutar una suma entre un operando en memoria y el acumulador, con modo de direccionamiento directo (la dirección está en los últimos 12 bits de la instrucción).

El primer paso es el *fetch*. Para leer la instrucción de memoria se debe cargar el contenido del registro PC en el MAR, realizar una operación de lectura desde memoria y se debe conectar el bus de datos con el registro IR, de forma de cargar en él la instrucción. Para ello hay que:

- conectar la salida del registro PC al bus B (B3 = 0, B2 = 0, B1 = 0, B0 = 0)
- seleccionar cargar el BLatch (L0 = 1)
- seleccionar cargar MAR (M0 = 1)
- seleccionar cargar el MBR (M1 = 1)
- seleccionar lectura de memoria (M2 = 1)
- seleccionar MBR en el AMux (X0 = 0)
- seleccionar la operación identidad-A en la ALU (F1 = 1, F0 = 0)
- seleccionar no desplazamiento en el Shifter (S0 = 0, S1 = 0)
- conectar la entrada del registro IR al bus C (C3 = 0, C2 = 0, C1 = 1, C0 = 0)
- habilitar el bus C (ENC = 1)

Al colocar todas estas señales en los valores indicados se procederá a la lectura de la instrucción contenida en la posición PC de la memoria. Notemos que de acuerdo a los tiempos de acceso de la memoria se pueden llegar a necesitar mantener las señales por más de un ciclo de reloj a los efectos de tener en cuenta las latencias (tiempos de propagación) de la circuitería involucrada.

Para completar el paso de *fetch* es necesario actualizar el valor de PC, para lo que se requiere las siguientes señales (en el siguiente ciclo de reloj):

- conectar la salida del registro PC al bus B (B3 = 0, B2 = 0, B1 = 0, B0 = 0)
- conectar la salida del registro +1 al bus A (A3 = 0, A2 = 1, A1 = 1, A0 = 0)
- seleccionar cargar el ALatch (L1 = 1)

- seleccionar cargar el BLatch (L0 = 1)
- seleccionar Bus A en el A Mux (X0 = 1)
- seleccionar A + B en la ALU (F1 = 0, F0 = 0)
- seleccionar no desplazamiento en el Shifter (S0 = 0, S1 = 0)
- conectar la entrada del registro PC al bus C (C3 = 0, C2 = 0, C1 = 0, C0 = 0)
- habilitar bus C (ENC = 1)

El siguiente paso del ciclo de instrucción es el *decode*. La decodificación se realiza analizando el código binario de la instrucción. Para ello se implementa en la CU una función mediante un circuito combinatorio que toma como entrada el contenido del IR y da como salida los valores apropiados de las señales de control, condicionando también el próximo estado de la máquina secuencial, como en el caso del ejemplo ya que al reconocer el modo de direccionamiento directo debe realizarse el paso de *read*.

Para realizar el paso de *read* se debe cargar la dirección del operando (contenida en los últimos 12 bits de la instrucción) en el MAR, para luego realizar una operación de lectura desde memoria, cargando el MBR desde el bus de datos, de forma de almacenar en él el operando. Para ello hay que:

- conectar la salida del registro IR al bus B (B3 = 0, B2 = 0, B1 = 1, B0 = 0)
- conectar la salida del registro AMASK al bus A (A3 = 1, A2 = 0, A1 = 0, A0 = 0)
- seleccionar cargar el ALatch (L1 = 1)
- seleccionar cargar el BLatch (L0 = 1)
- seleccionar Bus A en el A Mux (X0 = 1)
- seleccionar A and B en la ALU (F1 = 0, F0 = 1)
- seleccionar no desplazamiento en el Shifter (S0 = 0, S1 = 0)
- conectar la entrada del registro A al bus C (C3 = 1, C2 = 0, C1 = 1, C0 = 0)
- habilitar bus C (ENC = 1)

con estas señales y sincronizado por el reloj se guarda la dirección contenida en la instrucción en el registro A. A continuación (próximo ciclo de reloj) se colocan las señales:

- conectar la salida del registro A al bus B (B3 = 1, B2 = 0, B1 = 1, B0 = 0)
- seleccionar cargar el BLatch (L0 = 1)
- seleccionar cargar MAR (M0 = 1)
- seleccionar cargar el MBR (M1 = 1)
- seleccionar lectura de memoria (M2 = 1)

En el paso siguiente se realiza el *execute*. Para ello se deben activar las siguientes señales:

- conectar la salida del registro AC al bus B (B3 = 0, B2 = 0, B1 = 0, B0 =1)
- seleccionar cargar el BLatch (L0 = 1)
- seleccionar MBR en el A Mux (X0 = 0)
- seleccionar la operación A + B en la ALU (F0 = 0, F1 = 0)
- seleccionar no desplazamiento en el Shifter (S0 = 0, S1 = 0)
- conectar la entrada del registro AC al bus C (C0 = 1, C1 = 1, C2 = 0, C3 =1)
- habilitar bus C (ENC = 1)

Sincronizado por el reloj se realiza la operación en la ALU y se guarda el resultado en el registro AC.

Cabe destacar que al ser una instrucción cuyo resultado se almacena en un registro, no corresponde que exista un paso de *write*.

14.5 Lógica Cableada vs Lógica Microprogramada

Hay dos filosofías de diseño claramente diferenciadas para la Unidad de Control: la “lógica cableada” y la “lógica microprogramada”.

El diseño de la CU en base a la filosofía de “lógica cableada” se hace como cualquier circuito secuencial: en base a un diagrama de estados (que contemple todas las situaciones posibles y por tanto todos los estados posibles en función de las distintas instrucciones y sus variantes) traduciéndola a una máquina de Mealy (por ej.). Para el ejemplo anterior deberíamos tomar como salidas todas las señales de control necesarias y como entradas los bits del código binario de las instrucciones.

En este caso el resultado es óptimo desde el punto de vista de la velocidad de los circuitos, pero tiene poca flexibilidad ya que cualquier cambio en el diseño del set de instrucciones de la CPU genera la necesidad de re-escribir el diagrama de estados y, por ende, cambiar todo el circuito lógico que lo implementa.

En cambio la filosofía de “lógica microprogramada” propone que la CU se construya en base a un autómata más sencillo que “ejecute” el ciclo de instrucción de cada instrucción de la CPU siguiendo en secuencia un conjunto de “microinstrucciones” que contienen los valores de las señales apropiados para esa instrucción particular. Es decir que para cada instrucción a nivel de la CPU existirá un “microprograma”, consistente en una secuencia lógica de microinstrucciones que establecerán el orden de los eventos necesarios para lograr la ejecución de la instrucción en la CPU, incluyendo todas las “bifurcaciones” de la secuencia que se requieran en base a los distintos tipos de direccionamiento y cualquier otra variante que admita la instrucción específica. En este caso hay una penalización en la velocidad de proceso porque el microprograma se almacena en una ROM interna de la CPU y cada microinstrucción debe ser leída de ella, pero simplifica enormemente la tarea de modificar el set de instrucciones, ya que simplemente alcanza con cambiar el contenido de la ROM del microprograma. También las máquinas microprogramadas permiten fácilmente implementar instrucciones complejas, tales como la búsqueda de un elemento en un array, ó la copia o la comparación de arrays, etc.

La tendencia en materia de diseño de CPUs era, hasta la década de 1980, utilizar lógica microprogramada. La aparición de las arquitecturas RISCs puso en cuestión esa tendencia, ya que dichas propuestas promovían el uso de lógica cableada, en particular asociado al concepto de optimizar la velocidad de proceso. De todos modos es probable que la ventaja que esos diseños obtuvieron en materia de rendimiento se deba principalmente al uso óptimo de técnicas tales como el “pipelining” (que se verá más adelante) permitido por las características de su set de instrucciones. El punto es que si se usa un set “reducido” con fines de optimización, también resulta más sencillo implementar la CPU en “lógica cableada”.

14.6 Microprogramación Horizontal y Vertical

Si quisiéramos implementar en microcódigo (microprogramación) la Unidad de Control del ejemplo tendríamos que cada microinstrucción debería contener los bits codificados de las señales de control de lo distintos caminos de las señales de datos, como ocurre con el acceso al banco de registros. Este es uno de los enfoques posibles y de hecho tiene la ventaja de reducir la cantidad de bits necesarios para codificar cada microinstrucción. Recibe el nombre de **microprogramación vertical**.

Por el contrario la **microprogramación horizontal** utiliza microinstrucciones con los bits de control sin codificar. Para el caso del ejemplo se requerirían 16 bits de control para

cada bus (1 bit para cada uno de los registros) en vez de los 4 utilizados en forma codificada.

La microprogramación vertical supone que determinados conjuntos de recursos solo pueden ser utilizados de a uno por vez en cada función y de allí que pueden ser codificados. Esto permite ahorrar bits (y por tanto espacio en la ROM de microprograma), pero queda firme esa restricción que puede ser perjudicial a la hora de generar altos niveles de paralelismo, en especial en arquitecturas más complejas que la del ejemplo.

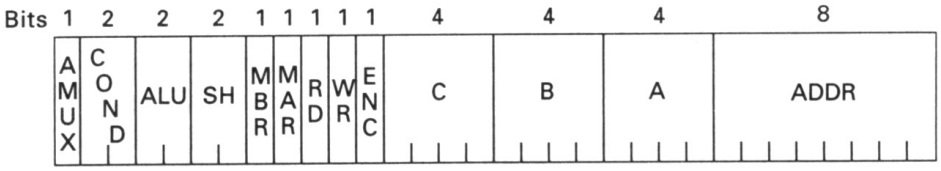
14.7 Microinstrucciones

La microarquitectura MIC-1 que hemos tomado como ejemplo propone una implementación mediante microprogramación. Para ello cuenta con una ROM para microprogramas (denominada “Control Store”), interna a la CPU, que permite almacenar un total de 256 microinstrucciones. Cada microinstrucción es codificada con un código binario de 32 bits.

Cada microinstrucción deberá tener bits disponibles para indicar los valores (0/1) de las señales que controlan los distintos elementos, a saber:

4 bits para control de bus A
4 bits para control de bus B
4 bits para control de bus C
1 bit para habilitar el bus C
2 bits para control de latches de bus
2 bits para control de la ALU
2 bits para control del Shifter
1 bit para control de AMux
1 bit para control del MAR
3 bits para control del MBR

El control de los latches en realidad es necesario en prácticamente todos los casos, por lo que se lo realizará directamente mediante el reloj, con lo que se requieren 22 señales. Dado que se utilizan 32 bits para codificar las microinstrucciones, los restantes 10 bits se utilizan para manejar la secuenciación del microprograma, como se explicará a continuación. Antes veamos el formato de las microinstrucciones:



AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch	0 = No jump	0 = A + B	0 = No shift	0 = No
1 = MBR	1 = Jump if N = 1	1 = A AND B	1 = Shift right 1 bit	1 = Yes
	2 = Jump if Z = 1	2 = $\underline{A}$	2 = Shift left 1 bit	
	3 = Jump always	3 = $\overline{A}$	3 = (not used)	

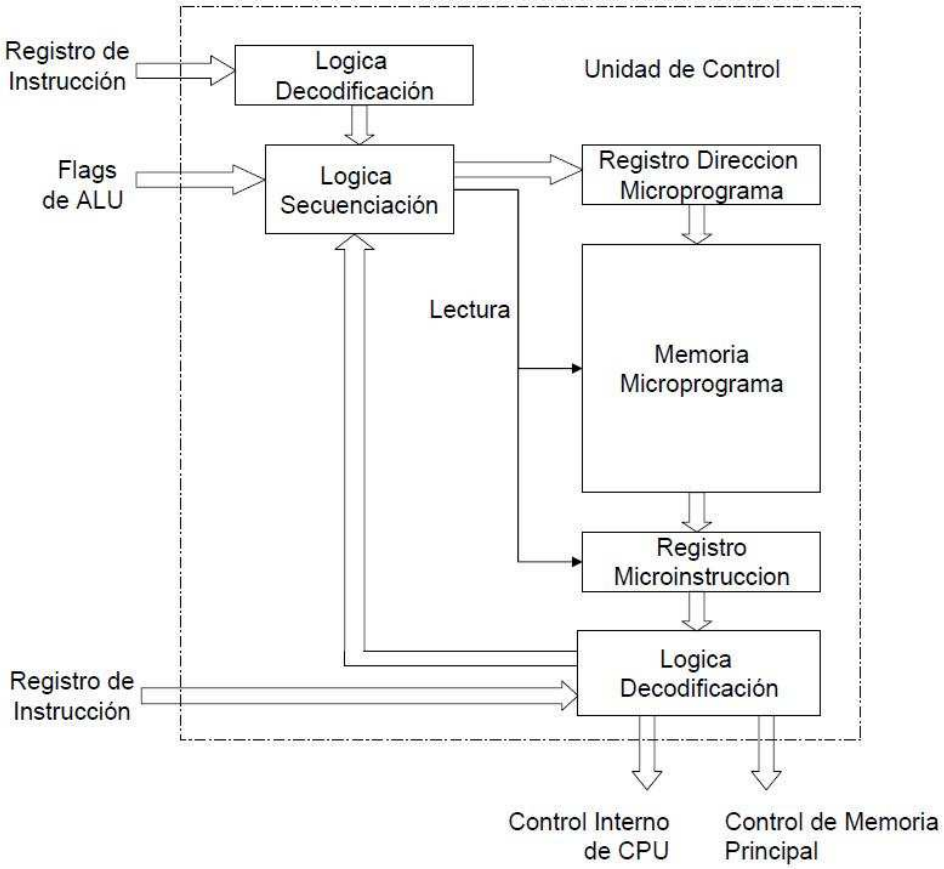
Para secuenciar las microinstrucciones del microprograma se procede de la siguiente manera. En función de lo que esté codificado en los bits COND la siguiente microinstrucción será:

00: la ubicada a continuación en la memoria ROM de microprograma
01: si la salida N de la ALU es 1 la ubicada en la dirección de la ROM ADDR
10: si la salida Z de la ALU es 1 la ubicada en la dirección de la ROM ADDR
11: siempre la ubicada en la dirección de la ROM ADDR

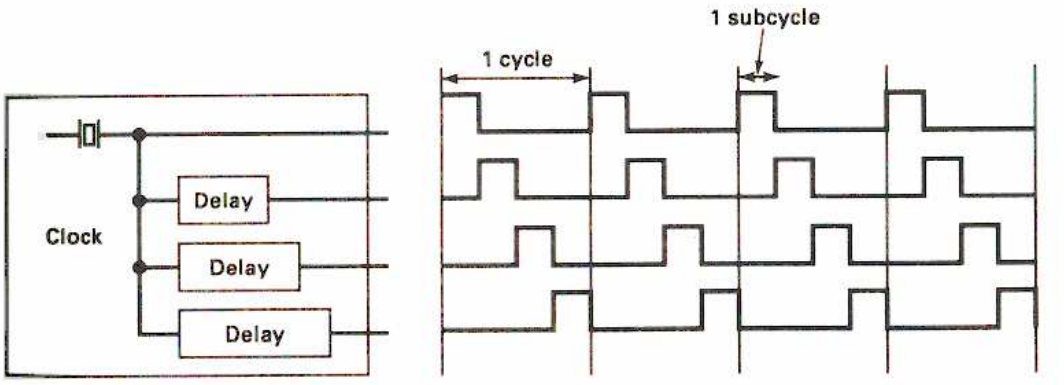
14.8 Unidad de Control Microprogramada

En general una implementación de una unidad de control microprogramada dispone de una ROM para almacenar las microinstrucciones de los microprogramas que implementan el ciclo de instrucción de las instrucciones de la CPU, y de una máquina secuencial que ejecuta la secuenciación de las microinstrucciones de acuerdo a la lógica que se haya codificado.

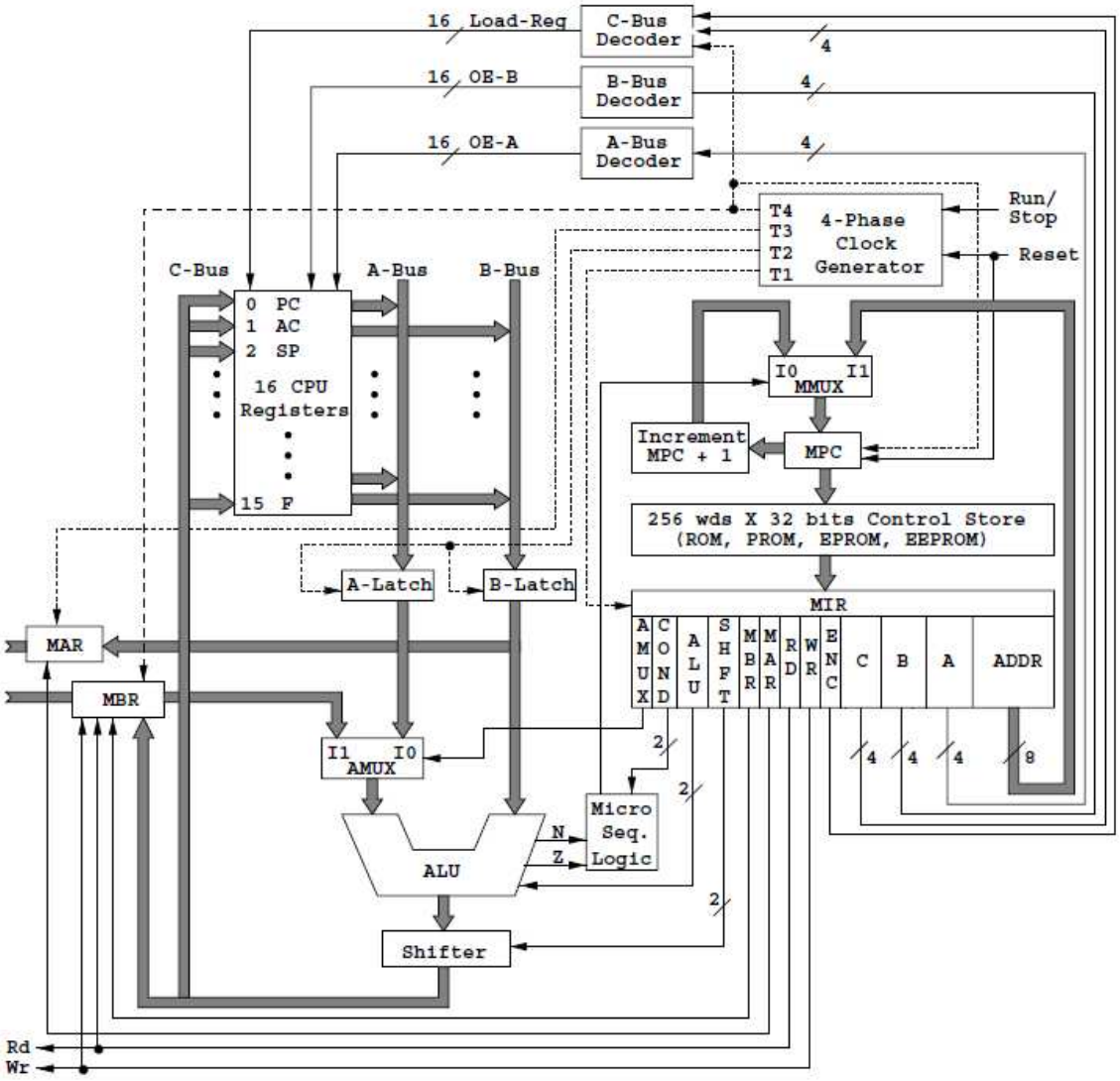
El esquema general es:



La implementación de MIC-1 utiliza para su sincronismo interno la técnica de relojes desfasados, que son señales de reloj obtenidas de un reloj “padre” mediante la utilización de “líneas de retardo” (dispositivos electrónicos que provocan un cierto retraso en la propagación de la señal):



Incorporando la parte de sincronismo y los elementos de la unidad de control microprogramada al esquema de la MIC-1, ésta queda:



Los elementos de la unidad de control microprogramada de la MIC-1 son:

- MPC (**M**icro **P**rogram **C**ounter): es un registro que almacena la dirección (8 bits) de la microinstrucción que se va a ejecutar
- Increment: circuito combinatorio que suma uno
- Mmux: es un multiplexor que selecciona entre la salida del Increment y los últimos 8 bits del MIR para determinar la dirección de la ROM
- Control Store: memoria ROM de 256 palabras de 32 bits donde se almacenan las microinstrucciones
- MIR (**M**icro **I**nstruction **R**egister): registro que almacena la microinstrucción leída desde la Control Store y desde el cuál se conectan las distintas señales de control al resto de los elementos de la microarquitectura

Notar que este diseño no responde estrictamente al esquema general presentado, porque busca la máxima simplicidad en su implementación. Esto lleva a realizar la decodificación en base a operaciones realizadas por la ALU (como se verá más adelante). Esto no es lo habitual en diseños comerciales de unidades de control.

14.9 Micro Ensamblador

Un microprograma será una colección de códigos binarios que representen a las microinstrucciones necesarias para controlar las señales deseadas.

Por ejemplo:

AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR	
0	00	10	00	0	1	1	0	0	0000	0000	0000	00000000	0x10c00000
0	00	10	00	0	0	1	0	0	0000	0000	0000	00000000	0x10400000
1	00	10	00	0	0	0	0	1	0011	0000	0000	00000000	0x90130000
0	00	00	00	0	0	0	0	1	0000	0110	0000	00000000	0x00106000

Al igual que para el caso de las instrucciones, “programar” en códigos binarios es un tanto engorroso. Por eso se desarrollan lenguajes “ensambladores” que permiten utilizar una nomenclatura bastante más amigable para representar las microinstrucciones.

Para el caso veremos el MAL (Micro Assembler Language) propuesto por Tanenbaum para su MIC-1.

Este micro-ensamblador tiene una notación basada en identificar en cada “sentencia” a todas las señales que deben estar en 1 (no aparecen las que van en 0). Para señalar las asignaciones de los buses A, B y C usa una notación similar a la asignación utilizada en los lenguajes de alto nivel como el Pascal.

Así por ejemplo:

ac:=d

indica que se desea copiar el contenido del registro D en el AC, para lo cual el bus C se conecta al registro AC, se activa ENC, el registro D se conecta al bus A, la ALU realiza la función identidad y el Shifter no desplaza.

La codificación de esta microinstrucción sería:

AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR	
0	00	10	00	0	0	0	0	1	0000	0000	1101	00000000	0x10100d00

Las cuatro operaciones de la ALU se representan así (los registros usados son a modo de ejemplo):

ac:=ac + d ac:=band(ir, smask) f:=ac c:=inv(a)

los registros destino pueden ser cualquiera (incluyendo el MAR y el MBR), pero el MAR no puede ser operando.

Las dos operaciones efectivas de la unidad de desplazamiento se representan así:

lshift(ac) rshift(ir)

Las operaciones de desplazamiento también aceptan como argumento una operación de ALU:

lshift(tir + tir)

(por ejemplo la microinstrucción anterior hace un shift de dos bits a la izquierda)

La activación de las señales RD y WR se realiza colocando rd ó wr en la expresión.

La secuenciación de las microinstrucciones se realiza mediante la expresión:

if n then goto XXXX if z then goto XXXX

donde XXXX es la dirección (de 8 bits) de la memoria ROM donde se almacena la microinstrucción del microprograma a la que se quiere ir (se puede asumir que es el "numero" de la microinstrucción si se asume que la primera es la de número 0).

Las distintas expresiones pueden ser combinadas en una misma microinstrucción (la misma "sentencia"):

tir:=lshift(tir + tir); if n then goto 33

el efecto de esta microinstrucción es guardar en el registro TIR el resultado de realizar un desplazamiento a la izquierda de 2 bits del propio registro TIR y saltar a la microinstrucción 33 si es que el segundo bit más significativo del valor original del TIR es 1 (notar que la ALU hace solo la suma que representa el desplazamiento a la izquierda de un bit y por tanto allí el bit más significativo del resultado es el segundo bit más significativo del original).

Hay un caso particular de sentencias, donde solo nos interesa determinar un salto en alguna condición de los bits de un registro, pero no queremos asignarlo a ningún otro registro. Para ese caso se dispone del "registro virtual" denominado "alu", que se usa así:

alu:=lshift(tir + tir); if n then goto 33

en este caso el resultado de la operación sobre TIR no afecta a ningún registro, pero la ALU actúa y computa el valor de la salida N según el resultado de la operación.

Por ejemplo: las codificaciones del primer ejemplo mostrado corresponden a las siguientes microinstrucciones:

```
mar:=pc; rd
rd
ir:=mbr
pc:=pc + 1
```

secuencia que analizada implica que el registro PC (Program Counter) se carga en el MAR y se activa la señal RD para realizar una lectura de memoria. Luego en la segunda microinstrucción se mantiene RD activa (esto es porque se está asumiendo que la lectura de la memoria demora dos ciclos de reloj). La siguiente microinstrucción mueve el contenido del MBR (lo que fue leído de memoria) al registro IR y finalmente la última microinstrucción le suma uno al PC. No es difícil identificar esta sucesión de eventos con la etapa de *fetch* del ciclo instrucción.

14.10 Microprograma de la MIC-1

En una máquina de estas características (microprogramada) típicamente el microprograma siempre empieza en la dirección 0, con las microinstrucciones necesarias para hacer el *fetch*. Luego siguen las requeridas para el *decode*. A partir del *decode* se producen saltos hacia las secuencias de microinstrucciones que implementan las etapas que sean requeridas (*read*, execute y write) según el tipo de instrucción y finalmente se vuelve a la dirección 0 para comenzar un nuevo *fetch*.

La etapa de *decode* de este ejemplo de microarquitectura es bastante trabajosa, por la escasez de recursos que su diseñador incluyó. Los recursos disponibles son los estrictamente necesarios para implementar las instrucciones de su macroarquitectura (el set de instrucciones), la cual recibe el nombre de MAC-1. Este conjunto de instrucciones tiene el formato de instrucción que está representado en la siguiente tabla:

OpCode Binary	OpCode Hex	Assembly Mnemonic	Instruction	Meaning or Action
0000xxxxxxxxxxx	0xxx	lodd	Load direct	ac:=m[x]
0001xxxxxxxxxxx	1xxx	stod	Store direct	m[x]:=ac
0010xxxxxxxxxxx	2xxx	addd	Add direct	ac:=ac+m[x]
0011xxxxxxxxxxx	3xxx	subd	Subtract direct	ac:=ac−m[x]
0100xxxxxxxxxxx	4xxx	jpos	Jump if positive	**if** ac≥0 **then** pc:=x
0101xxxxxxxxxxx	5xxx	jzer	Jump if zero	**if** ac=0 **then** pc:=x
0110xxxxxxxxxxx	6xxx	jump	Jump	pc:=x
0111xxxxxxxxxxx	7xxx	loco	Load constant	ac:=x (0≤x≤4095)
1000xxxxxxxxxxx	8xxx	lodl	Load local	ac:=m[x+sp]
1001xxxxxxxxxxx	9xxx	stol	Store local	m[x+sp]:=ac
1010xxxxxxxxxxx	axxx	addl	Add local	ac:=ac+m[x+sp]
1011xxxxxxxxxxx	bxxx	subl	Subtract local	ac:=ac−m[x+sp]
1100xxxxxxxxxxx	cxxx	jneg	Jump if negative	**if** ac<0 **then** pc:=x
1101xxxxxxxxxxx	dxxx	jnze	Jump if nonzero	**if** ac≠0 **then** pc:=x
1110xxxxxxxxxxx	exxx	call	Call procedure	sp:=sp−1;m[sp]:=pc;pc:=x
1111000000000000	f000	pshi	Push indirect	sp:=sp−1;m[sp]:=m[ac]
1111001000000000	f200	popi	Pop indirect	m[ac]:=m[sp];sp:=sp+1
1111010000000000	f400	push	Push onto stack	sp:=sp−1;m[sp]:=ac
1111011000000000	f600	pop	Pop from stack	ac:=m[sp];sp:=sp+1
1111100000000000	f800	retn	Return	pc:=m[sp];sp:=sp+1
1111101000000000	fa00	swap	Swap ac, sp	tmp:=ac;ac:=sp;sp:=tmp
11111100yyyyyyyy	fcyy	insp	Increment sp	sp:=sp+y (0≤y≤255)
11111110yyyyyyyy	feyy	desp	Decrement sp	sp:=sp−y (0≤y≤255)
1111111111111111	ffff	halt	Halt machine	stops fetching instructions

Tomado de “The Microarchitecture/Microprogramming Level” de la Universidad de Maryland

Notar que las instrucciones que tienen operandos directos a memoria contienen una dirección de 12 bits. Por su lado las operaciones “insp” y “desp” usan un operando inmediato de 8 bits. Justamente para poder manipular estos valores es que existen los registros AMASK (máscara de 12 bits) y SMASK (máscara de 8 bits).

Tener en cuenta que esta opción de diseño del formato de instrucción condiciona la capacidad de direccionamiento de esta CPU (un máximo de 4096 palabras de 16 bits). También condiciona los modos de direccionamiento de los operandos (solo directo para las operaciones aritméticas), la disponibilidad de registros (solo se dispone del AC como registro para operaciones) y los tipos de salto (son todos absolutos).

Notar también que los códigos de operación que definen de qué instrucción se trata están armados como un “árbol” (a medida que se avanza hacia la derecha aparecen más opciones según los valores de los bits anteriores). Este diseño determina la forma en que se puede hacer la decodificación (básicamente a fuerza de desplazamientos sucesivos y verificación del valor del bit más significativo resultante), aunque la misma es realmente poco eficiente realizada de esta forma.

En la tabla de la próxima página se puede observar un posible microprograma completo para la implementación de la macroarquitectura MAC-1 en la microarquitectura MIC-1.

Adr: Microinstruction	Comment
0: mar:=pc; rd;	**fetch instr**
1: pc:=pc + 1; rd;	increment pc
2: ir:=mbr; if n then goto 28;	decode ir_{15}
3: tir:=lshift(ir + ir); if n then goto 19;	decode ir_{14}
4: tir:=lshift(tir); if n then goto 11;	decode ir_{13}
5: alu:=tir; if n then goto 9;	decode ir_{12}
6: mar:=ir; rd;	0000 =**LODD**
7: rd;	
8: ac:=mbr; goto 0;	
9: mar:=ir; mbr:=ac; wr;	0001 = **STOD**
10: wr; goto 0;	
11: alu:=tir; if n then goto 15;	decode ir_{12}
12: mar:=ir; rd;	0010 = **ADDD**
13: rd;	
14: ac:=mbr + ac; goto 0;	
15: mar:=ir; rd;	0011 = **SUBD**
16: ac:=ac + 1; rd;	
17: a:=inv(mbr);	
18: ac:=ac + a; goto 0;	
19: tir:=lshift(tir); if n then goto 25;	decode ir_{13}
20: alu:=tir; if n then goto 23;	decode ir_{12}
21: alu:=ac; if n then goto 0;	0100 = **JPOS**
22: pc:=band(ir,amask); goto 0;	perform jump
23: alu:=ac; if z then goto 22;	0101 = **JZER**
24: goto 0;	else don't jump
25: alu:=tir; if n then goto 27;	decode ir_{12}
26: pc:=band(ir,amask); goto 0;	0110 = **JUMP**
27: ac:=band(ir,amask); goto 0;	0111 = **LOCO**
28: tir:=lshift(ir + ir); if n then goto 40;	decode ir_{14}
29: tir:=lshift(tir); if n then goto 35;	decode ir_{13}
30: alu:=tir; if n then goto 33;	decode ir_{12}
31: a:=ir + sp;	1000 = **LODL**
32: mar:=a; rd; goto 7;	
33: a:=ir + sp;	1001 = **STOL**
34: mar:=a; mbr:=ac; wr; goto 10;	
35: alu:=tir; if n then goto 38;	decode ir_{12}
36: a:=ir + sp;	1010 = **ADDL**
37: mar:=a; rd; goto 13;	
38: a:=ir + sp;	1011 = **SUBL**
39: mar:=a; rd; goto 16 ;	
40: tir:=lshift(tir); if n then goto 46;	decode ir_{13}
41: alu:=tir; if n then goto 44;	decode ir_{12}
42: alu:=ac; if n then goto 22;	1100 = **JNEG**
43: goto 0;	
44: alu:=ac; if z then goto 0;	1101 = **JNZE**
45: pc:=band(ir,amask); goto 0;	
46: tir:=lshift(tir); if n then goto 50;	decode ir_{12}
47: sp:=sp + (-1);	1110 = **CALL**
48: mar:=sp; mbr:=pc; wr;	
49: pc:=band(ir,amask); wr; goto 0;	
50: tir:=lshift(tir); if n then goto 65;	decode ir_{11}
51: tir:=lshift(tir); if n then goto 59;	decode ir_{10}
52: alu:=tir; if n then goto 56;	decode ir_9
53: mar:=ac; rd;	1111-0000 = **PSHI**
54: sp:=sp + (-1); rd;	
55: mar:=sp; wr; goto 10;	
56: mar:=sp; sp:=sp + 1; rd;	1111-0010 = **POPI**
57: rd;	
58: mar:=ac; wr; goto 10;	
59: alu:=tir; if n then goto 62;	decode ir_9
60: sp:=sp + (-1);	1111-0100 = **PUSH**
61: mar:=sp; mbr:=ac; wr; goto 10;	
62: mar:=sp; sp:=sp + 1; rd;	1111-0110 = **POP**
63: rd;	
64: ac:=mbr; goto 0;	
65: tir:=lshift(tir); if n then goto 73;	decode ir_{10}
66: alu:=tir; if n then goto 70;	decode ir_9
67: mar:=sp; sp:=sp + 1; rd;	1111-1000 = **RETN**
68: rd;	
69: pc:=mbr; goto 0;	
70: a:=ac;	1111-1010 = **SWAP**
71: ac:=sp;	
72: sp:=a; goto 0;	
73: tir:=lshift(tir); if n then goto 76;	decode ir_9
74: a:=band(ir,smask);	1111-1100 = **INSP**
75: sp:=sp + a; goto 0;	
76: alu:=tir; if n then goto 80;	decode ir_8
77: a:=band(ir, smask);	1111-1110 = **DESP**
78: a:=inv(a);	
79: a:=a + 1; goto 75;	
80: halt; goto 80;	1111-1111 = **HALT**

Tomado de “The Microarchitecture/Microprogramming Level” de la Universidad de Maryland