Episodio 1

Modelo von Neumann

Se divide la computadora en 5 unidades principales interconectadas entre sí:

- Memoria
- ALU
- Entrada
- Salida
- Control

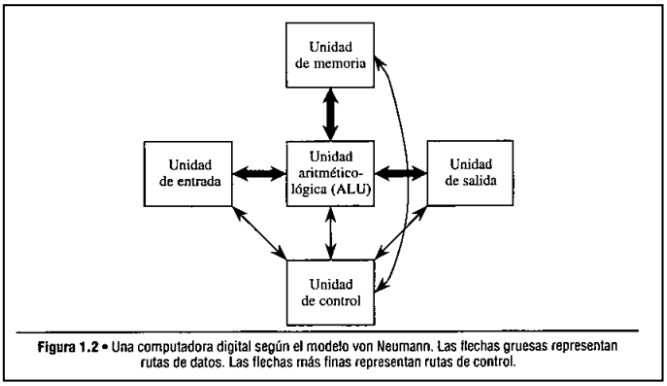


Figura 1.2 • Una computadora digital según el modelo von Neumann. Las flechas gruesas representan rutas de datos. Las flechas más finas representan rutas de control.

Principal innovación: programas guardados en memoria propia de la computadora en vez de estar almacenados externamente. Da origen a compiladores y demás, permitiendo mayor flexibilidad.

Modelo de Bus

Una "refinación" del modelo anterior, consisten en crear varios buses para la comunicación entre las distintas unidades. Además se agrupa la ALU y la unidad de control en un solo bloque funcional, la CPU. Se agrupa la E/S también. Los buses a saber son:

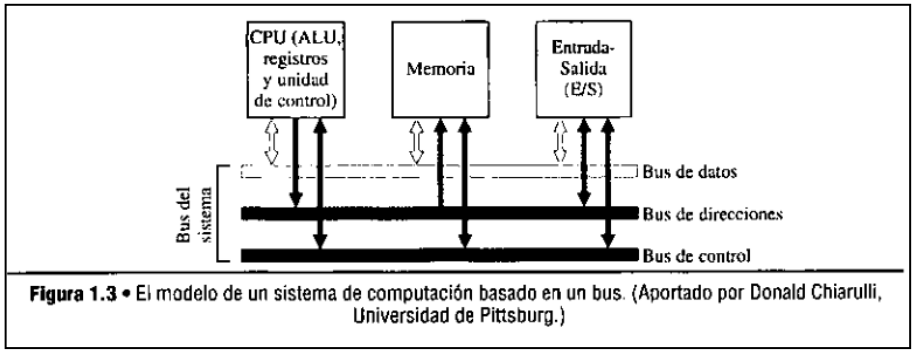


Figura 1.3 • El modelo de un sistema de computación basado en un bus. (Aportado por Donald Chiarulli, Universidad de Pittsburg.)

- *Bus de datos*: transporta información per sé
- *Bus de direcciones*: transporta direcciones para R/W en memoria
- *Bus de control*: contiene la información sobre cómo se debe manipular la información en los demás buses
- *Bus de alimentación*: distribución de energía eléctrica. Se sobreentiende en los diagramas funcionales

Los buses, físicamente, no son mas que un conjunto de cables agrupados donde confluyen o divergen en varias partes hacia sus destinos/orígenes.

Niveles de la computadora

Para analizar la computadora se la divide en 7 niveles, comenzando desde el nivel físico mas concreto hasta el mas "abstracto" que es el que el usuario veo. La escala es como sigue:

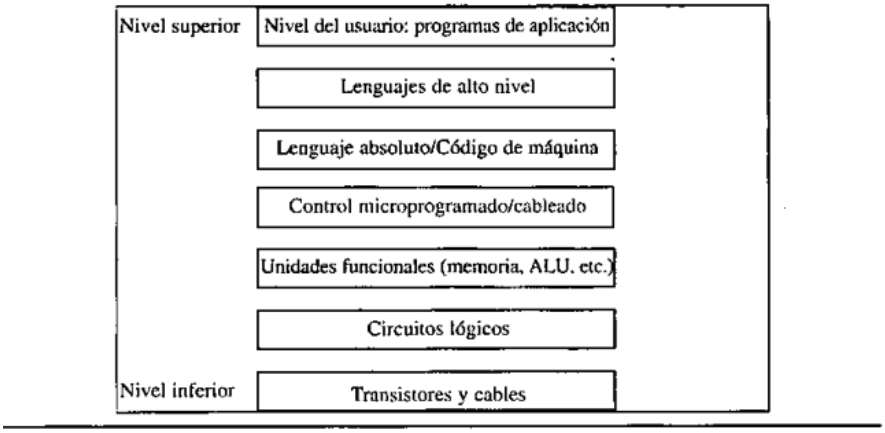


Figura 1.4 • Niveles de máquina en la jerarquía de un sistema.

- *Nivel de usuario*: programas que el usuario utiliza, no importan los detalles internos de funcionamiento sino la usabilidad y se maneja la E/S usando periféricos.

- *Lenguajes de alto nivel*: Se usan para programar el nivel de usuario. Son lenguajes multi-arquitectura (cross-platform) donde se ocultan los detalles de funcionamiento interno de la maquina y se programa utilizando abstracciones comunes. El compilador o intérprete traduce las instrucciones de alto nivel a las instrucciones reales correspondientes a la arquitectura donde se ejecuta el programa. Esto se conoce como compatibilidad de código fuente.
- *Lenguaje absoluto/código de máquina*: es el lenguaje mas básico que ofrece la arquitectura para programar. Se debe conocer los detalles de la arquitectura, como por ejemplo la cantidad y tamaño de los registros, las operaciones disponibles y sus limitaciones. Esto se conoce como ISA (Instruction Set Architecture), juego de instrucciones o "código ASM". Se utiliza un assembler para traducir dicho código a la data binaria que es la que la CPU ejecuta finalmente (y la que se guarda en memoria para su ejecición).
- *Nivel de control*: es el nivel encargado de llevar a cabo las operaciones del ISA. Generalmente viene programado en un microprograma en el firmware de la máquina (ejecutado por un microcontrolador), pero también puede estar cableado directamente.
- *Unidades funcionales*: implementan las operaciones de la unidad de control, como transferencia de datos entre registros, aritmética, lectura/escritura, guardar datos, etc,etc. En otras palabras, son registros de la CPU, la ALU y la memoria.
- *Circuitos lógicos*, *transistores y cables*: la base de la computadora; las operaciones lógicas básicas y el transporte de señales eléctricas de un lado a otro.

Los niveles existen para darle distintos enfoques al análisis. Por ejemplos los programadores van a estar preocupados por los 3 primeros niveles, mientras que los diseñadores de computadoras se preocupan con los 4 restantes. Estos últimos son generalmente los que están más sujetos a restricciones de costo/rendimiento.

Episodio 4

Introducción

En este capítulo se trata la arquitectura de programación desde el punto de vista del lenguaje de máquina (o assembler). Como se mencionó anteriormente, es el lenguaje de programación de mas bajo nivel y para escribir programas es necesario conocer la arquitectura desde el punto de vista de las unidades funcionales ya que se trata directamente con ellas: registros, memoria, ALU, etc. A pesar de que finalmente los programas se guardan en binario, se escriben usando el lenguaje de mnemónicos que es el assembler: la traducción entre código ASM simbólico y el código binario que finalmente puede leer y ejecutar la máquina es directa. El conjunto de operaciones se conoce como Instruction Set Architecture (ISA).
Aunque las instrucciones y datos están almacenados en memoria, durante la ejecución del programa cada instrucción del programa se carga en el CPU desde la memoria, de a una y secuencialmente al mismo tiempo que se leen los datos en los registros necesarios para su ejecución. En todo el libro se utiliza la arquitectura "ficticia" ARC, derivada de la arquitectura real SPARC, basada en RISC.

Memoria

La memoria consiste de un array de celdas de 8 bits (byte) indexadas secuencialmente por su *dirección de memoria*. Generalmente se utiliza el concepto de "word" (palabra) como unidad básica de memoria, cuyo tamaño depende exclusivamente de la arquitectura. En general, y en el caso de ARC, 1 word equivale a 4 bytes o 32 bits. También se usa el concept de "dword" (double word), que consiste en un word del doble de tamaño, así como half-word (hword) y quad word (4*word).
Cuando se almacenan palabras de mas de 1 byte, se pueden utilizar dos convenciones con respecto al orden en las que se las almacena a nivel byte: big-endian o little-endian. En el primer caso los bytes con los bits más significativos se almacenan en las posiciones más bajas de memoria, en el otro al revés. En ambos casos se refiere al byte por la dirección más baja en memoria. Por ejemplo si se quiere almacenar el número 258 en un hword de 16 bits, en la posición de memoria 2000 tenemos:

- Little endian:
  - *2000*: 00000010 (2)
  - *2001*: 00000001 (256)

- Big endian:
  - *2000*: 00000001 (256)
  - *2001*: 00000010 (2)

Donde el primer número indica la posición de memoria, seguido de los 8 bits en el byte y entre paréntesis su valor decimal con el peso dado por la interpretación de un entero unsigned de 16 bits.

En la arquitectura ARC se utiliza un modelo de memoria con un *espacio de direcciones* (rango numérico al que puede acceder la CPU) de 32 bits, lo que significa que se puede acceder desde la posición 0 hasta la posición $2^{32} - 1$, ya que la primer dirección de memoria es el cero. La memoria está dividida en sectores, formando el *mapa de memoria* como se muestra en la figura:

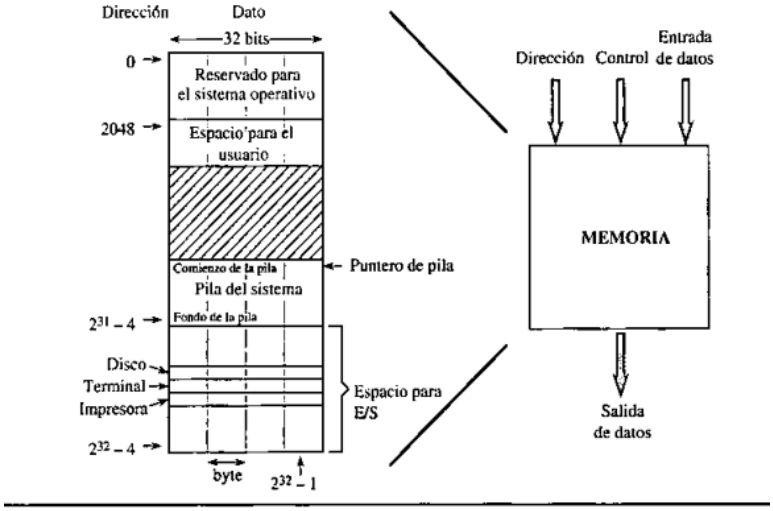


Figura 4.4 • Un mapa de memoria para un ejemplo de arquitectura. (No dibujado en escala.)

Los programas del usuario se cargan a partir de la dirección 2048. Notar que la pila decrece en dirección de memoria cuando aumenta su data asi que es factible que se encuentren ambos el espacio para el usuario y la sección reservada para la pila, resultando en overflow y obviamente malfuncionamiento del programa. También se muestra lo que se conoce como *entrada-salida mappeada en memoria*, donde los dispositivos de entrada/salida tienen asignados por el sistema operativo una sección de memoria que leen/escriben para comunicarse con los programas y viceversa.
Es normal utilizar hexadecimal para describir direcciones de memoria ya que si bien se utiliza direccionamiento al byte, solo se mueven datos del tamaño de la palabra por ser el tamaño fijo de los registros y buses. Por ejemplo, en ARC se utilizan palabras de 32 bits por lo que toda la memoria que se desee acceder debe estar dentro del rango 0x00000000-0xFFFFFFFF

CPU (Central processing unit – unidad central de proceso)

La CPU consiste en dos secciones, el datapath (trayecto de datos) y la unidad de control. El datapath contiene la ALU y los registros, la unidad de control es la encargada de interpretar la actual instrucción en ejecución asi como de realizar transferencias entre registros. La interacción entre estas dos unidades está mediada por dos registros especiales. El PC (*program counter*) y el IR (*instruction register*). El PC contiene la actual dirección de memoria de donde se carga la instrucción de la memoria principal para su ejecución. Esta instrucción se carga justamente en el IR, donde se la interpreta. Este proceso se conoce como *ciclo de fetch* o *búsqueda-ejecución*:

1. Buscar en memoria la próxima instrucción
2. Decodificarla
3. Busqueda en memoria de operandos (opcional)
4. Ejecución de la instrucción y almacenamiento de resultados
5. Repeat

La CPU se trata en detalle en el episodio 6.

Detalles de la arquitectura ARC

ARC es un subconjunto de la arquitectura SPARC, basados en RISC (Reduced Instruction Set Computer). Como tal, presenta una cantidad mínima de instrucciones y es muy simple en su diseño. Los puntos importantes son:

- Todos los registros son de 1 word = 32 bits
- 32 registros de uso general, + el PC y el IR
- Un registro especial , el PSR (Processor Status Register), para indicar flags (códigos de condición) provenientes de la ALU. A saber: z (zero), n (negativo), c (carry – overfow unsigned), v (overflow signed)
- Las instrucciones son de 32 bits
- ARC es una maquina de carga-descarga (load-store), las instrucciones deben cargarse en el IR antes de ser ejecutadas, asi como los operandos necesarios para ejecutarla deben ser cargados en registros antes. En otras palabras, no se opera con la memoria principal sino con lo que está en los registros únicamente. También se escribe a memoria solo con desde los registros.

Formato del lenguaje simbólico en ARC

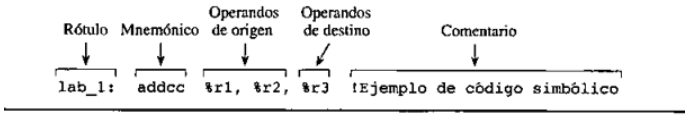


Figura 4.8 • Formato de una sentencia en el lenguaje simbólico SPARC (también ARC).

El lenguaje es case-sensitive. No se pueden usar caracteres especiales excepto guión bajo y punto para los rótulos. Los registros son como sigue:

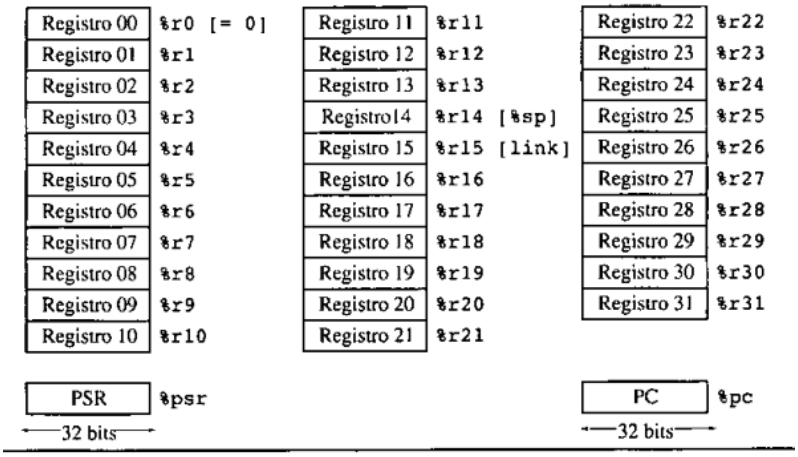


Figura 4.9 • Registros en ARC accesibles al programador.

Donde %r0 es siempre 0, %r14 es el stack-pointer por convención (%sp), %r15 es el registro de enlace como se explica más adelante.

Formato de instrucción en ARC

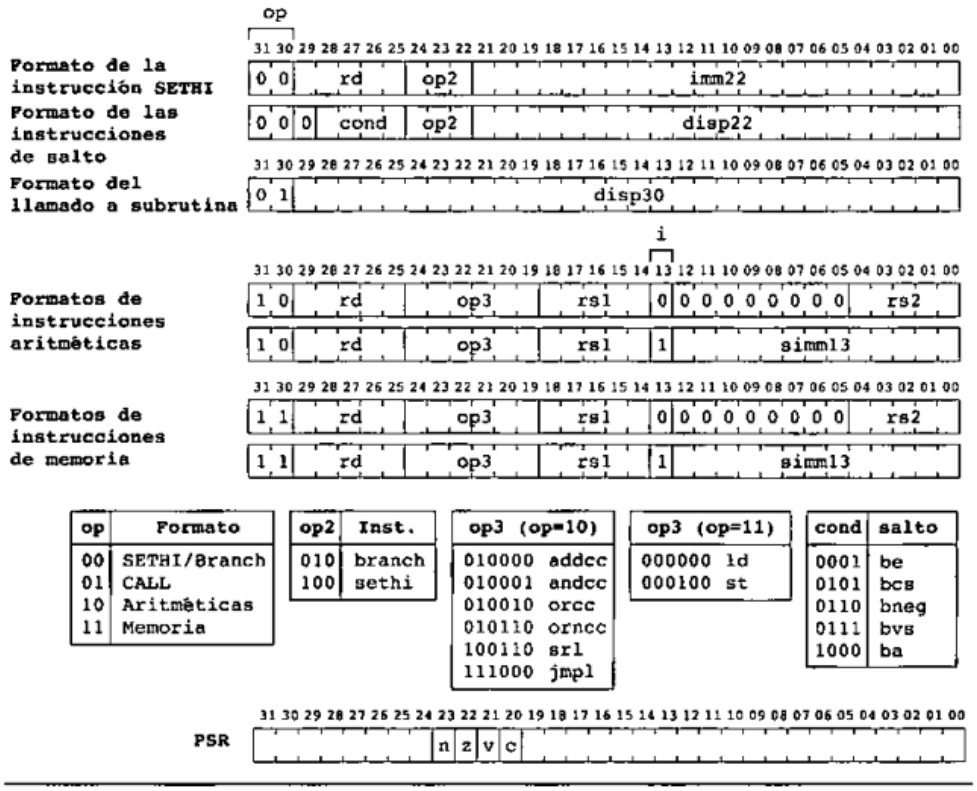


Figura 4.10 • Formatos de instrucción y contenido del PSR en ARC.

Conjunto de instrucciones en ARC, pila y subrutinas

(ver libro para la descripción del set de instrucciones completo)

Truquitos:

- Las operaciones aritméticas tienen dos versiones, con el sufijo 'cc' y sin. Cuando lo llevan, setean los códigos de condición en el PSR si los hubiese. Caso contrario no lo hacen nunca.
- Siempre incluir *.begin* y *.end* al principio y final del programa respectivamente.
- Cuando se utiliza constantes en el código hay restricciones de 13 bits. Por ejemplo, si queremos cargar lo que hay en la posición de memoria 0x4000 no podemos hacer "ld 0x4000, %1" porque no entra en la instrucción de 32 bits que luego se compila. Para hacer algo así tenemos dos opciones: cargar la constante por partes (usando *sethi* podemos setear los 22 bits mas significativos y luego con un *and* o *add* cargar los demás bits. O cargarla en memoria y luego leerla de ahí. Por ejemplo, en la posición de memoria 2052 podemos poner la constante que queremos, luego hacemos *ld 2052, %r1; ld %r1, %r2.*
- Existen dos shifts hacia la derecha, *srl* y *sra.* El primero completa con ceros mientras que el segundo completa con 0 o 1 dependiendo del bit más significativo, extendiendo el signo. Para hacer un shift hacia la izquierda se multiplica por dos, addeando el numero con sí mismo.
- La instrucción *call* carga en el registro %15 la dirección desde donde se saltó. Cuando se termina de ejecutar la subrutina, hay que volver a donde se estaba y seguir con la próxima instrucción. La versión mas simple es poniendo *jmpl %r15+4, %r0.* Pero si se quieren hacer llamadas anidadas se tiene que usar la pila:
- Para utilizar la pila, es conveniente definir las macros *push* y *pop* como sigue:

```
.macro push arg
        add %r14, -4, %r14
        st %r14, arg
.endmacro
.macro pop arg
        ld %r14, arg
        add %r14, 4, %r14
```

.endmacro
Luego, antes de llamar a cualquier subrutina, se debe pushear los argumentos que se van a utilizar. Cuando la subrutina se ejecuta, esta poppea los argumentos de a uno para procesarlos y pushea los resultados en la pila que después se utilizan. En el caso de llamadas anidadas, cada subrutina que invoque a otra subrutina debe pushear el %r15 de manera de que no se sobreescriba y se pierda el punto de enlace.
Opcionalmente, se puede usar *.equ %sp, %r14.*

- Otra manera de transferir datos entre subrutinas es utilizando una convención fija de registros o una zona de memoria reservada, la "zona de transferencia de datos".
- Si se utilizan rutinas o rótulos definidos en módulos separados se debe utilizar *.global <rotulo>* donde se lo define y *.extern <rotulo>* donde se lo pretende usar
- Siempre poner *.org 2048* antes de la primer instrucción para indicar el espacio reservado para el OS.
- Cada flag de la ALU setteado en el PSR tiene su correspondiente salto condicional*. z=be, c=bcs, v=bvs, n=bneg*. La razón por la que el de *z* es "branch equal" es porque para comparar si dos números son iguales se los resta y se pregunta si el resultado es cero. También existe '*ba'* para saltar siempre ("branch always"). También existe *bne* para cuando *z* –no- está setteado aka "branch not equal"

Otras arquitecturas y variantes

Para reducir u optimizar el tráfico de memoria a veces se busca modificar las instrucciones. Por ejemplo se puede simplificar la instrucción *add A,B,C* (A+B -> C) como, *add A,B* (A+B -> B). Incluso se puede utilizar un único registro si se tiene en cuenta un registro especial de acumulador: *add A* (acc+A -> acc). Esto reduce el tamaño del programa y el tránsito de memoria, bajando costos y aumentando velocidad.
La mayoría de las arquitecturas modernas incluyen otros registros especiales además de los ya vistos:

- Registros source y destination para acceso mas rápido a arreglos en memoria
- Registros optimizados para floating-point
- Registros especiales para mayor resolución, por ejemplo para manejo del tiempo
- Registros especiales para niveles restringidos o privilegiados del procesador, para ser usado solo por la BIOS o el OS por ejemplo.

Modos de direccionamiento

Existen varias formas de acceder a un dato:

- Inmediato: el dato es una constante incorporada en la instrucción a ejecutarse (ej. en simm13 de *add*)
- Directo: el dato está en una posición de memoria con dirección constante embebida en el comando *ld* (ej. *ld 5000, %r1*)
- Indirecto: se utiliza un registro como puntero a la dirección de memoria donde está el dato a cargar (ej, pila de llamadas)
- Variantes de indirecto: indexado, basado en registro y combinaciones. Básicamente, se utiliza 1 o 2 registros sumados (o no) a una constante para acceder a la posición de memoria donde están los datos. (ej, acceder a un array, un struct, etc)

Episodio 5

Introducción

En este capítulo se trata el proceso por el cual los programas finalmente corren en la máquina, desde que se escriben en lenguajes de alto nivel hasta que son cargados en memoria como código binario para su ejecución. Los pasos a detallar en este capitulo son, en orden: compilación, ensamblado (assembly), enlance (linking) y carga.

Compilación

Los pasos que debe hacer el compilador de un lenguaje como C con cada sentencia son:

- Análisis lexicográfico: reconocer los elementos básicos del lenguaje como identificadores, definiciones y delimitadores.

- Análisis sintático: analizar los símbolos del paso anterior para reconocer la estructura del programa. El *parser* reconoce las sentencias y verifica los identificadores, expresiones, constantes y demás.
- Análisis de nombres: asociar los nombres de identificadores con variables y luego con posiciones en memoria donde se almacenarán.
- Análisis de tipo: determinar el tipo de todos los datos requeridos y verificar que sean compatibles. Este paso y el anterior se agrupan como *"análisis semántico"*
- Asignación de acciones y generación de código: a cada sentencia del programa se la traduce en una o más secuencias del lenguaje ensamblador.
- Pasos adicionales finales: posibles niveles de optimización, control de registros, agregar símbolos de debugging, etc.

Cuando se compila un programa, este se traduce finalmente a lenguaje de ensamblador que es específico de la arquitectura donde se corre el programa. Además, se tiene que tener en cuenta las restricciones de dicha arquitectura, por ejemplo la cantidad de registros o de memoria disponible. Es posible que el compilador soporte compilar un programa para una arquitectura diferente donde corre el mismo compilador; este proceso se llama *cross-compiling*: la "especificación de asignación" le indica al compilador para qué arquitectura se desea compilar el programa.

Variables

Existen dos tipos fundamentales de variables: globales (estáticas) y locales (automáticas). Las primeras son accesibles durante el curso de todo el programa y están fijas en memoria en una ubicación conocida al momento de compilación. Las variables locales solo son ubicadas en memoria cuando están en "scope" (dentro de la subrutina que las utiliza, etc) y luego se pierden. Como tales, solo se puede saber donde se ubican al momento de ejecución.
Una forma natural de implementar las variables automaticas es usando una pila LIFO como ya se vio. Además, se suele utilizar *direccionamiento base,* donde se accede a las variables usando una copia del %sp, llamada %fp, más un offset. Por ejemplo, "*ld %fp, -12, %r1*". Esto permite la carga de variables de la pila a los registros en una sola instrucción y también resalta el hecho de que para las variables automática solo se conoce su posición en memoria como un desplazamiento respecto del %fp.

Además de las variables de tipos simples los lenguajes ofrecen datos compuestos como los *structs* y los *arrays*. En el caso de los structs se define y declara una instancia 'p':

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

Luego se accede como *p.x* o *p.y*. La dirección en memoria se calcula como un offset con respecto a la primera. Entonces en este caso, *p.x* tendría la dirección de *p*, mientras que *p.y* tendriá la dirección de *p+4* y *p.z* la dirección *p+8*.

En el caso de los arrays (o vectores, arreglos) , se los define con un tipo homogéneo y un tamaño "n" conocido al momento de compilación. Opcionalmente en lenguajes como Pascal se puede determinar el índice donde comienza el arreglo. En C y muchos otros lenguajes, se utiliza la convención de *[0;(n-1)]*. Nuevamente, la dirección el elemento con índice 'i' se calcula como un offset con respecto al primero elemento:

*Posición del elemento i-ésimo: base + (i – start)*size*

Donde *start* sería 0 siempre para los lenguajes como C y *size* es el tamaño del tipo declarado para el array (por ej, 4 bytes si es un arreglo de 32 bit ints). Entonces, para acceder al elemento i-esimo se debe realizar una suma, un producto y otra suma. Suponiendo un array de ints llamado A, *base* en %r2, *i* en %r3, *start* en %r4 y *size*=4, el código ASM sería:

```
sub %r3, %r4, %r6
sll %r6, 2, %r6
ld A, %r6, %r1
```

En lenguajes como C se ahorra la primer instrucción. Ambas operaciones *sub* y *sll* no son instrucciones pero se pueden definir como macros. La primera sería una adición con el complemento a 2, la segunda es la adición del argumento con si mismo 'n' veces (x+x = 2x). Recordar que un shift de "n" dígitos a izq. equivale a multiplicar por 2^n, mientras que un shift a derecha equivale a dividir por 2 y truncar.

Secuencias de control

Para implementar las bifurcaciones condicionales, loops, etc, se utilizan los flags del PSR y generalmente el salto incondicional en conjunto con el "be" que chequea el flag z (ver libro para ejemplos detallados). Si se quiere chequear si dos variables son iguales se las resta, descarta el resultado y se salta condicionalmente con 'be', donde el salto se realiza si las dos variables son iguales y se sigue con la línea siguiente en el caso contrario.

Ensamblado

El ensamblado es la traducción del programa en lenguaje simbólico a código binario. Es un proceso directo ya que las instrucciones en ASM se corresponden directamente con instrucciones del procesador. Los siguientes ítems deben ser parte de cualquier ensamblador:

- Permitir al programador especificar ubicación en memoria (aunque sea relativa al comienzo del programa) de variables
- Inicialización automática de datos al comienzo del programa
- Proveer expresiones mnemotécnicas para cada instrucción del procesador y modos de direccionamiento
- Permitir el uso de rótulos para direcciones de memoria constantes
- Proveer construcciones como macros y otras opciones como la dirección de comienzo del programa, la posibilidad de hacer aritmética básica y lógica pre-ensamblado.

La mayoría de los ensambladores usan una técnica conocida como "ensambladores de dos pasadas". En la primer pasada se determina las direcciones de todos los datos e instrucciones, mediante un contador llamado *location counter* (contador de posición). Si bien comienza en cero y aumenta secuencialmente, se puede modificar su valor directamente con la directiva *.org*. Durante esta pasada el ensamblador también realiza cualquier operación de cálculo necesaria, expande las macros e inserta cualquier definición de identificadores (rotulos y constantes) en la **tabla de símbolos**, junto con su ubicación en memoria (según el location counter).
La razón para la segunda pasada es el hecho de que algunos identificadores se utilizan en un programa antes de ser declarados ("*referencia previa"*), y es justamente una de las funciones que cumple la tabla de símbolos. Si en la segunda pasada se encuentra un identificador que no está presente en la tabla de símbolos, es un error y el ensamblado se interrumpe. Caso contrario, se lo reemplaza por su posición en memoria cuando se traduce la instrucción a código binario.

Cuando se completa el proceso de traducción, se incluye información adicional en el módulo ensamblado:

- Nombre y tamaño del módulo
- Dirección de comienzo, si existe (sería la posición del *main()* en C por ejemplo)
- Información sobre los símbolos globales y externos, para la correcta interacción entre los módulos
- Información sobre las rutinas de biblioteca que el módulo utiliza
- Los valores de constantes a cargar en memoria, ya que algunos programas esperan que se cargue por separado
- Información de reubicación: cuando se combinan modulos, la mayoría deben re-ubicarse en memoria. Algunas direcciones se marcan como reubicables y otras no.

Desde el punto de vista del programador, en la mayoría de los casos no importa o finalmente no se sabe la ubicación exacta de los programas en la memoria dado que los programas están compuestos de varios módulos. Por eso, la mayoría de las direcciones se especifican como *reubicables*. Una excepción conocida seria la entrada/salida mappeada en memoria. Esta información se coloca en un *diccionario de reubicación* en el módulo ensamblado final, para el linker/loader.

Enlace y carga

El linker se encarga de combinar los distintos módulos y convertir el *código objeto* en un *módulo de carga* (aka, un ejecutable). Resuelve todas las referencias globales y externas y reubica las direcciones de los diferentes módulos. El módulo de carga se carga en memoria por un *cargador (loader)*, duh, que también puede cambiar las direcciones si el programa se carga en una dirección distinta a la especificada por el linker. En algunos casos se utiliza linkeo "dinámico", postergando la resolución de referencias a rutinas en bibliotecas compartidas (DLLs) hasta la ejecución del programa.
Toda la información requerida para reubicar un módulo se almacena en la tabla de símbolos contenida en el módulo ensamblado, por lo que está disponible tanto para el linker como el cargador. Esto es necesario para saber cuáles símbolos son reubicables, especialmente en el caso de un *cargador-reubicador*, donde se tienen más de un programa corriendo en memoria como en cualquier OS moderno: en este caso, una vez que el linker resolvió todas las referencias y las posiciones de memoria son todas validas y conocidas, el cargador se encarga de agregarle un offset a todas las direcciones de memoria reubicables, tal que se puedan cargar más de un programa en memoria y funcionen correctamente.
El archivo ejecutable contiene la información sobre su posición de carga, dirección de comienzo y otra posible información como puntos de entrada para rutinas que otros programas pudiesen acceder (si fuese una biblioteca compartida).
Otro approach es la utilización de una MMU (*Memory Management Unit)*, que realiza la reubicación durante la carga utilizando un registro base con respecto al cual todas las demás direcciones se offsetean/reubican (en el programa se hace todo desde la dirección cero y es transparente al ensamblador).

Episodio 6

Introducción

En los caps anteriores se estudiaron los niveles mas altos, desde el usuario hasta el nivel del lefnguaje de máquina accesible al programador/compilador/ensamblador. Ahora es el turno del nivel que sigue en la escala, encargado de implementar el nivel superior, el de la arquitectura de programación: la microarquitectura. Está compuesta por una sección de control y el datapath (trayecto de datos), que interactúan con la memoria donde reside el programa, el input y el output. Un detalle es que la microarquitectura es totalmente transparente a la arquitectura de programación: se puede implementar de manera completamente distinta y sin requerir ningún cambio en los niveles superiores. La elección de implementación generalmente se debe a un balanceo de costos, velocidad, hacer upgrades, etc. En este capitulo se ve el caso especifico de la arquitectura ARC.

Fundamentos de la microarquitectura

La microarquitectura es la encargada de que se complete correctamente el ciclo de fetch descripto en el capítulo 4. Es la encargada de: buscar la instrucción a ser ejecutada, determinar qué instrucción se trata, localizar los operandos, ejecutar la instrucción, almacenar los resultados y repetir el proceso. Se la analiza como la figura:

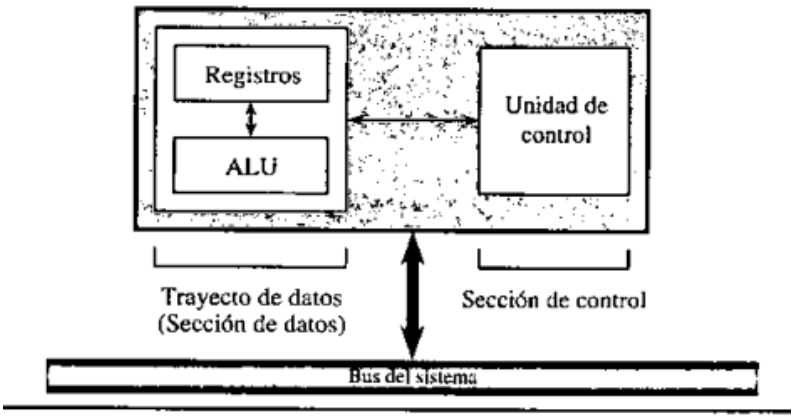


Figura 6.1 • Microarquitectura vista desde el alto nivel.

El datapath consiste en los registros y la ALU, mientras que la sección de control contiene el cableado y memoria necesarios para controlar y ejecutar correctamente el ciclo de búsqueda y cualquier instrucción válida que se cargue en el *%ir*. Las dos maneras de implementar la sección de control que se estudian son: la microprogramada y la cableada.

El datapath

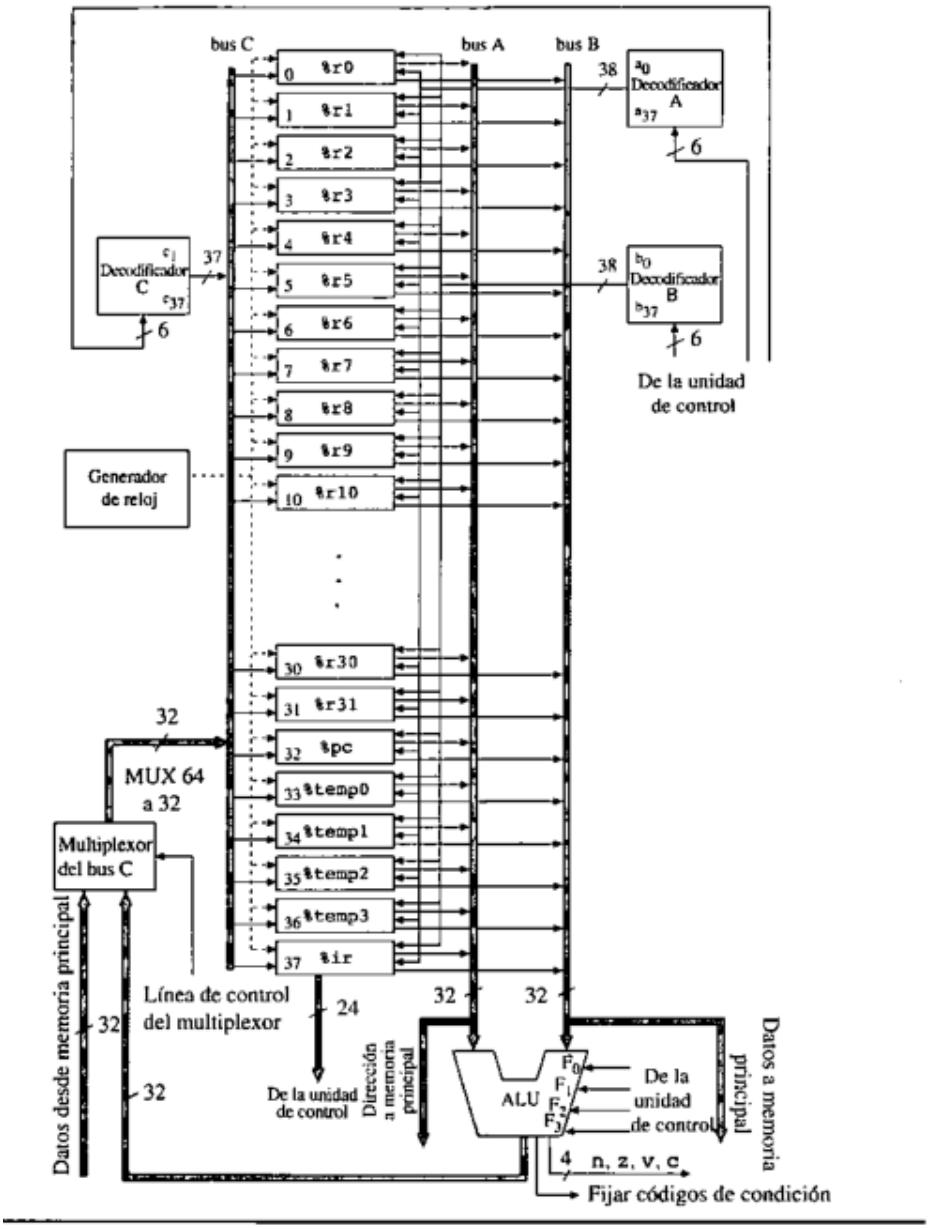


Figura 6.3 • Trayecto de datos en ARC.

Como se ve, el datapath consiste de los registros accesibles al usuario *(%r0-%r32*) y 8 registros adicionales utilizados por la microarquitectura. Recapitulando: el *%pc* es el program counter (dirección de la instrucción a cargar), el *%ir* el instruction register (registro donde se carga la instrucción para decodificarla). Los *%temp* se utilizan (si es necesario) para guardar resultados temporales que necesitan para la ejecución de la instrucción actual en el *%ir*. Brotip sobre mux y decos:

- MUX (multiplexor): toma varias entradas y mediante una “línea de control”, selecciona una única para la salida.
- DEMUX (demultiplexor): inverso del anterior, toma una sola entrada y mediante una línea de control selecciona a qué salida enviar la señal.
- Deco (decodificador): en este caso (hay otras codificaciones) se encargan de convertir una señal de *n* bits en 2^n bits “one-hot”. Esta última configuración significa que en un conjunto de bits, solo uno de ellos puede estar en 1

mientras que el resto está en 0. Entonces por ejemplo, el decodificador C tiene una entrada de 6 lineas o bits. Esto equivale a 2^6 = 64 posibles combinaciones por lo tanto esa sería la cantidad máxima de salidas del decodificador. En este caso se utilizan únicamente 37 de esas salidas por lo que poner un numero mayor a 37 como entrada en el deco C sería un error porque ninguna salida estaría en 1. Para ilustrar otro ejemplo de un decodificador bien simple, supongamos un decodificador con una entrada de 2 bits. La salida serían 4 bits si se usan todas las 4 combinaciones de entrada y un posible mappeo sería: 00 -> 0001; 01 -> 0010; 10 -> 0100; 11 -> 1000.

ALU

Por otro lado, la ALU está incluida en el datapath. Realiza una de 16 operaciones sobre lo que haya en los buses A y B, colocando el resultado en el bus C.

F_3	F_2	F_1	F_0	Operación	Modifica códigos de condición
0	0	0	0	ANDCC (A, B)	sí
0	0	0	1	ORCC (A, B)	sí
0	0	1	0	NORCC (A, B)	sí
0	0	1	1	ADDCC (A, B)	sí
0	1	0	0	SRL (A, B)	no
0	1	0	1	AND (A, B)	no
0	1	1	0	OR (A, B)	no
0	1	1	1	NOR (A, B)	no
1	0	0	0	ADD (A, B)	no
1	0	0	1	LSHIFT2 (A)	no
1	0	1	0	LSHIFT10 (A)	no
1	0	1	1	SIMM13 (A)	no
1	1	0	0	SEXT13 (A)	no
1	1	0	1	INC (A)	no
1	1	1	0	INCPC (A)	no
1	1	1	1	RSHIFT5 (A)	no

Figura 6.4 • Operaciones aritméticas en ARC.

Se implementan las operaciones lógicas y aritméticas básicas, shifteos a izquierda de a 2 y 10 bits (LSHIFT2 y LSHIFT10), aislar los 13 bits menos significativos sin (SIMM13) y con extensión de signo (SEXT13). INC incrementa el valor en 1 del bus A, INCPC incrementa el **valor del bus A (no del PC, pero se ve por que lleva ese nombre.. las instrucciones tienen 4 bytes)** en 4 (un word), y RSHIFT5 hace un shift a derecha de 5 bits, con extensión de signo. Los shifts se utilizan para decodificar las instrucciones, en particular las de salto o para determinar si el bit 13 está encendido o no, aunque también se utilizan para implementar las instrucciones de alto nivel de shift (*srl* y *sra*). Con estas operaciones es posible implementar cualquier otra operación. Por ejemplo, la resta se puede hacer haciendo la suma al complemento: primero se hace el complemento del numero a restar (NOR consigo mismo), luego un INC y finalmente se suman los dos operandos. La ALU se encarga de settear los flags (n-v-c-z) en el *%psr,* mediante la SCC (set condition codes), flag que se activa únicamente con las instrucciones que terminan en CC.

Una posible implementación de la ALU es utilizando una LUT (*Look-up table, tabla de búsqueda*). Se “hardcodean” todas las posibles combinaciones de operaciones y entrada para la salida, como el desarrollo gigante una única función lógica de 4 (operación) + 32 (entrada bus A) + 32 (entrada bus B) bits de entrada y 32 (salida bus C) + 1 (SCC) + 4 flags (n,v,z,c). Para simplificarla se la puede descomponer en una cascada de 32 LUTs que implementan cada una de las operaciones, seguida por un circuito *controlador de desplazamientos (barrel shifter)* que implementa los desplazamientos.

Registros

Cada registro no es mas que una colección de 32 FF (uno por cada bit) sincrónicos tipo D activados por flanco descendiente. La única diferencia es que se utiliza la salida del decodificador correspondiente (A,B o C) para determinar si se está leyendo (caso A,B) o escribiendo (caso C) del registro en cuestión o no. Para esto se utiliza una compuerta llamada “tri-state” buffer (tres estados), que funciona como un diodo con un tercer estado: 0, 1 (dependiendo de la entrada) o ‘z’ que significa ‘desconectado eléctricamente’ aka sin señal. Este mecanismo se utiliza para poder seleccionar un único registro como fuente/destino de los 37 disponibles. Algunos registros son especiales, como el *%r0* que es siempre 0 y por lo tanto no require FF’s. El registro *%ir* tiene salidas adicionales para ayudar a decodificar la instrucción allí cargada. El *%pc* solo puede contener un múltiplo de 4 por lo que los 2 bits menos significativos pueden ser desconectados.

La sección de control

El esquema de la CPU incluyendo el datapath, la sección de control y su interacción con la memoria principal:

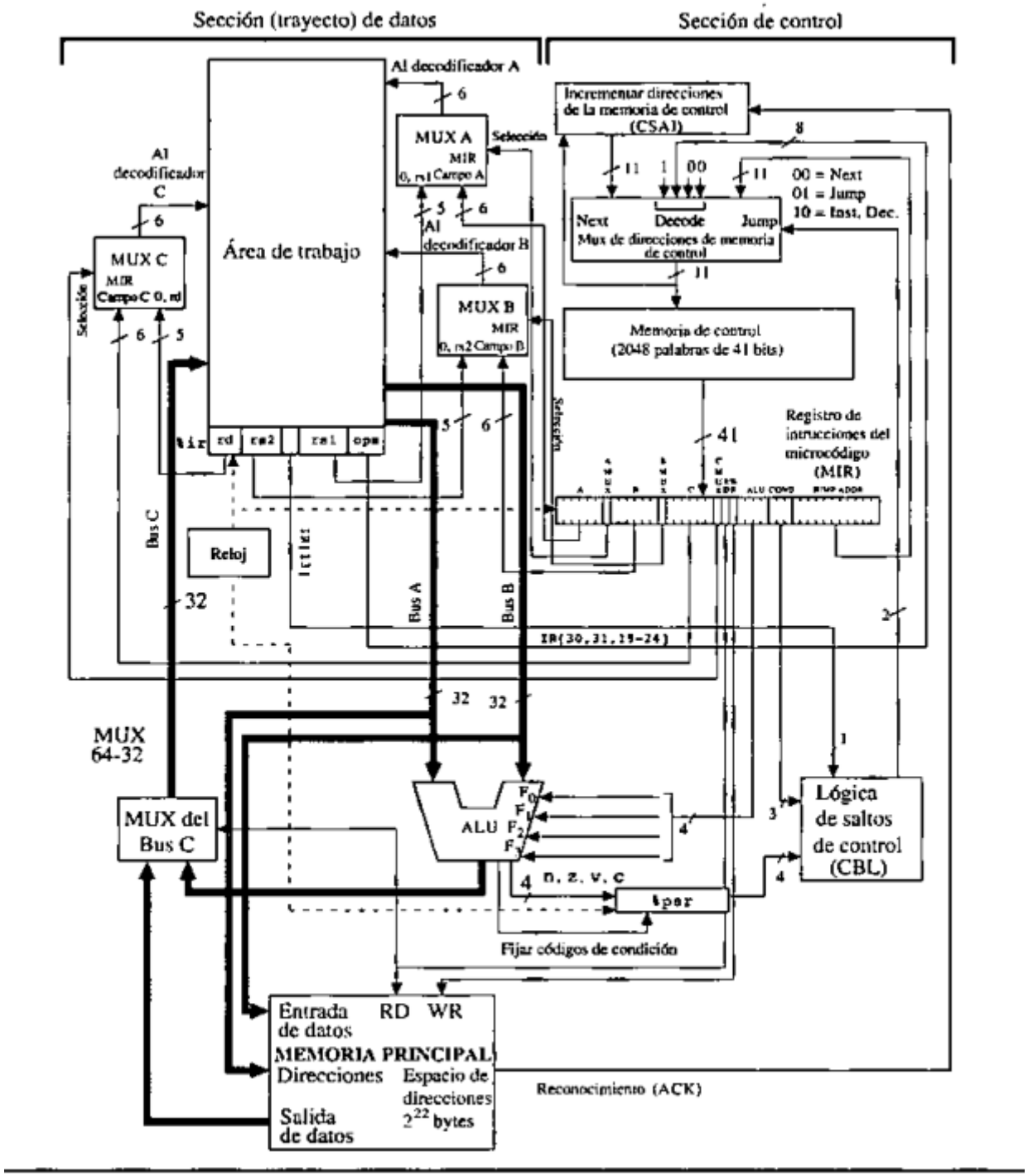


Figura 6.10 • La microarquitectura de ARC.

Además del cableado necesario, la sección de control cuenta con varias secciones importantes:

- El *%psr* que ya sabemos lo que es: un registro de 32 bits donde solo se utilizan 4 para settear los flags n/z/c/v de salida de la ALU
- Una memoria ROM de control (*control store*), de 2048 palabras de **41 bits**. Cada word es una microinstrucción.
- Un registro MIR de 41 bits donde se carga la microinstrucción en ejecución (análogo al IR para la arquitectura de programación).
- Una sección encargada de controlar la lógica de saltos (CBL – Control branch logic). Esta se encarga de manejar los saltos dentro de la memoria de la sección de control

- Un multiplexor de direcciones para la memoria de control
- Un circuito de inicialización (no se muestra) que se encarga de poner la primer palabra de la control store en el MIR.
- Un circuito (CSAI) para hacer la decodificación de la dirección de memoria a leer en la ROM.

A partir de la primer palabra (que contiene el primer paso del ciclo de fetch: buscar la instrucción indicada por el *%pc* y cargarla en el *%ir*), se determina a qué parte de la memoria de control saltar, mediante el MUX de direcciones del control store, basado en los valores del campo COND del MIR y de la salida del CBL. La codificación de los 41 bits de cada palabra en la control store memory es el siguiente:

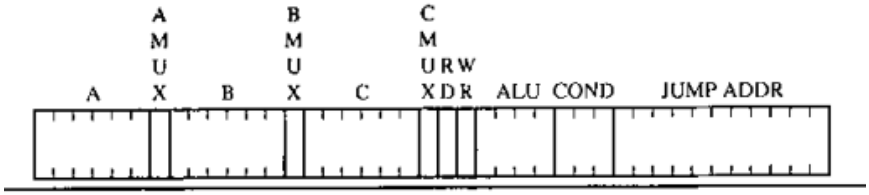


Figura 6.11 • El formato de la palabra de microcódigo.

Los campos A, B y C determinan el registro a activar para cada bus correspondiente. Dos cosas hay que notar: una es que tiene 6 bits, ya que debe poder acceder a los registros especiales pasando el *%r31*. La otra es que este campo puede ser ignorado en operaciones especiales como por ejemplo lectura o escritura a memoria principal (cuando RD o WR están en 1, respectivamente), o si se usa algún registro indicado en el *%ir* (i.e: *ld 1024, %r1.* En este caso el registro 1 a poner en el bus se lee del *%ir*, no del MIR). Los selectores AMUX, BMUX y CMUX cumplen justamente esa función: cuando están activados, el valor del campo respectivo del MIR es irrelevante ya que el decodificador del bus correspondiente va a seleccionar el registro indicado basado en la palabra que hay en el *%ir* y no en el MIR. Para ejemplificar, si cargamos un 000001 en A y ponemos AMUX=0 se va a colocar el registro *%r1* en el bus A. Sin embargo, independientemente de lo que carguemos en A, si ponemos AMUX=1, entonces el registro que se coloque en el bus A va a ser el indicado por el *rs1* del *%ir*.
En las operaciones de lectura y escritura a memoria principal, se toma la dirección del bus A. Para la lectura, el dato a leer se coloca en el bus C mientras que durante una escritura se toma la palabra a escribir del bus B. Para la lectura, el RD controla directamente el MUX que decide si lo que se carga en el bus C viene de memoria o de la ALU. Notar que la ALU no se puede "apagar", sino que simplemente se descarta el resultado en el MUX del bus C cuando se hace una operación de escritura/lectura a memoria. El campo ALU indica cual de las 16 operaciones de la ALU a realizar.
El campo COND indica de donde obtener la próxima dirección de memoria de control: hay 3 posibilidades básicas. Usar la próxima dirección, usar la dirección indicada en JUMP ADDR o DECODE que significa decodificar la instrucción actual en el MIR para determinar la dirección a donde saltar. Esta última se utiliza únicamente en la primer posición de memoria de la microarquitectura en el 2do paso del ciclo de fetch. Por otro lado, los saltos de JUMP ADDR pueden estar sujetos a los códigos de condición del *%psr*, para implementar los saltos condicionales a nivel arquitectura de programación: *be, bneg, bcs, bvs*. La codificación del campo COND se da a continuación:

C_2	C_1	C_0	Operación
0	0	0	Usar NEXT ADDR
0	0	1	Usar JUMP ADDR if n = 1
0	1	0	Usar JUMP ADDR if z = 1
0	1	1	Usar JUMP ADDR if v = 1
1	0	0	Usar JUMP ADDR if c = 1
1	0	1	Usar JUMP ADDR if IR[13] = 1
1	1	0	Usar JUMP ADDR
1	1	1	DECODE

Figura 6.12 • Valores que adopta el campo COND de la palabra de microcódigo.

Para determinar la próxima dirección de memoria en un DECODE se usa el siguiente algoritmo. Se crea una combinación de 11 bits (recordar que la memoria de control tiene 2^{11} = 2048 palabras) para determinar que posición de memoria a leer. Se coloca un 1 en el MSB (most significant bit) y 0 en los 2 LSB (least significant bit). Luego se completan los bits que quedan con los bits 31-30-24-23-22-21-20-19 del *%ir*. En la figura se muestra como esto ayuda a decodificar basado en los campos op, op2 y op3 del IR:

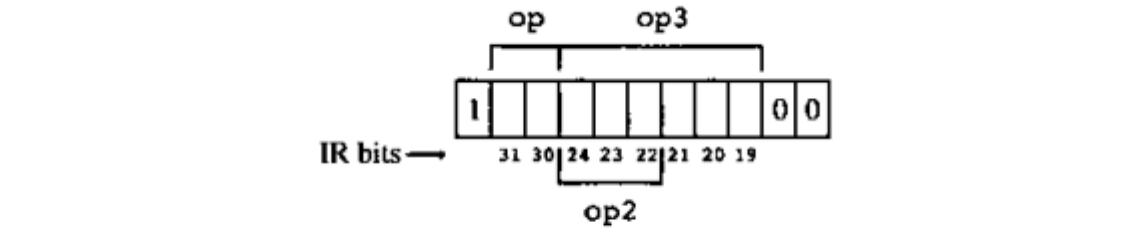


Figura 6.13 • El formato de DECODE para generar la dirección en una microinstrucción.

Temporización

La microarquitectura opera con un CLK de 2 fases, los FF D de los registros usan una configuración master-slave. La figura explica lo que hay que saber:

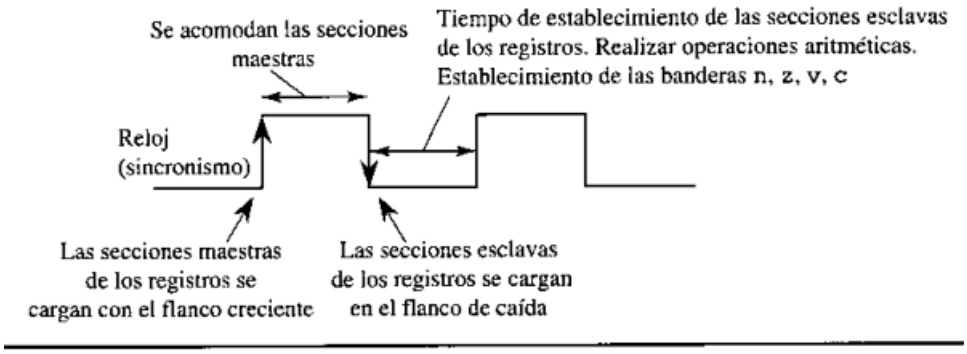


Figura 6.14 • Relaciones de tiempo para el funcionamiento de los registros.

En la primer fase se cargan los registros y en la segunda se realizan las operaciones de la ALU.

El firmware

El programa de la microarquitectura que se guarda en la ROM se denomina *firmware*. Se lo programa en un lenguaje simbólico (lenguaj de microensamblador) o directamente de manera binaria. Como ya se vio, cada palabra debe tener 41 bits y hay un espacio de 2048 palabras. A c

Dirección	Sentencias operativas	Comentario
0:	R[ir] ← AND(R[pc],R[pc]); READ;	/ Leer una instrucción de ARC desde memoria principal
1:	DECODE;	/ Salto (256 posibilidades) condicionado al código de operación
	/ sethi	
1152:	R[rd] ← LSHIFT10(ir); GOTO 2047;	/ Copiar el campo imm22 en el registro de destino
	/ call	
1280:	R[15] ← AND(R[pc],R[pc]);	/ Guardar %pc en %15
1281:	R[temp0] ← ADD(R[ir],R[ir]);	/ Desplazar el campo disp30 a izquierda
1282:	R[temp0] ← ADD(R[temp0],R[temp0]);	/ Desplazar nuevamente
1283:	R[pc] ← ADD(R[pc],R[temp0]);	/ Salto a subrutina
	GOTO 0;	
	/ addcc	
1600:	IF R[IR[13]] THEN GOTO 1602;	/ El segundo operando origen está en modo inmediato?
1601:	R[rd] ← ADDCC(R[rs1],R[rs2]);	/ Resolver ADDCC sobre registros origen
	GOTO 2047;	
1602:	R[temp0] ← SEXT13(R[ir]);	/ Obtener el campo simm13, con extensión de signo
1603:	R[rd] ← ADDCC(R[rs1],R[temp0]);	/ Resolver ADDCC sobre operandos origen en registro/simm13
	GOTO 2047;	/
	/ andcc	
1604:	IF R[IR[13]] THEN GOTO 1606;	/ El segundo operando origen está en modo inmediato?
1605:	R[rd] ← ANDCC(R[rs1],R[rs2]);	/ Resolver ANDCC sobre registros origen
	GOTO 2047;	
1606:	R[temp0] ← SIMM13(R[ir]);	/ Obtener el campo simm13
1607:	R[rd] ← ANDCC(R[rs1],R[temp0]);	/ Resolver ANDCC sobre operandos origen en registro/simm13
	GOTO 2047;	/
	/ orcc	
1608:	IF R[IR[13]] THEN GOTO 1610;	/ El segundo operando origen está en modo inmediato?
1609:	R[rd] ← ORCC(R[rs1],R[rs2]);	/ Resolver NORCC sobre registros origen
	GOTO 2047;	
1610:	R[temp0] ← SIMM13(R[ir]);	/ Obtener el campo simm13
1611:	R[rd] ← ORCC(R[rs1],R[temp0]);	/ Resolver ORCC sobre operandos origen en registro/simm13
	GOTO 2047;	
	/ orncc	
1624:	IF R[IR[13]] THEN GOTO 1626;	/ El segundo operando origen está en modo inmediato?
1625:	R[rd] ← NORCC(R[rs1],R[rs2]);	/ Resolver ORNCC sobre registros origen
	GOTO 2047;	
1626:	R[temp0] ← SIMM13(R[ir]);	/ Obtener el campo simm13
1627:	R[rd] ← NORCC(R[rs1],R[temp0]);	/ Resolver NORCC sobre operandos origen en registro/simm13
	GOTO 2047;	/
	/ srl	
1688:	IF R[IR[13]] THEN GOTO 1690;	/ El segundo operando origen está en modo inmediato?
1689:	R[rd] ← SRL(R[rs1],R[rs2]);	/ Resolver SRL sobre registros origen
	GOTO 2047;	
1690:	R[temp0] ← SIMM13(R[ir]);	/ Obtener el campo simm13
1691:	R[rd] ← SRL(R[rs1],R[temp0]);	/ Resolver SRL sobre operandos origen en registro/simm13
	GOTO 2047;	
	/ jmpl	
1760:	IF R[IR[13]] THEN GOTO 1762;	/ El segundo operando origen está en modo inmediato?
1761:	R[pc] ← ADD(R[rs1],R[rs2]);	/ Resolver ADD sobre operandos origen en registro/simm13
	GOTO 0;	

Figura 6.15 • Microprograma parcial de ARC. Las micropalabras se muestran en secuencia lógica (no numérica).

En la programación se ve que el lenguaje simbolico de microensamblador no mas que una manera mas textual de llenar los campos del MIR. Cada palabra tiene únicamente 41 bits siempre y se debe completar los campos necesarios como el COND, ALU y JUMP ADDR. Notar que luego del DECODE, se salta a una dirección de memoria mayor a 2^{10}, ya que como se dijo antes durante el DECODE la próxima dirección se llena con un 1 en el MSB siempre (y tiene 11 bits).

```
1762: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13, con extensión de signo
1763: R[pc] ← ADD(R[rs1],R[temp0]);      / Resolver ADD sobre operandos origen en registro/simm13
      GOTO 0;
      / ld
1792: R[temp0] ← ADD(R[rs1],R[rs2]);     / Calcular dirección origen
      IF R[IR[13]] THEN GOTO 1794;
1793: R[rd] ← AND(R[temp0],R[temp0]);    / Colocar dirección origen sobre el bus A
      READ; GOTO 2047;
1794: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13 para la dirección origen
1795: R[temp0] ← ADD(R[rs1],R[temp0]);   / Calcular dirección de origen
      GOTO 1793;
      / st
1808: R[temp0] ← ADD(R[rs1],R[rs2]);     / Calcular dirección de destino
      IF R[IR[13]] THEN GOTO 1810;
1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;   / Mover el campo rd hacia la posición del campo rs2
  40: R[ir] ← RSHIFT5(R[ir]);            / desplazándolo 25 bits a la derecha.
  41: R[ir] ← RSHIFT5(R[ir]);
  42: R[ir] ← RSHIFT5(R[ir]);
  43: R[ir] ← RSHIFT5(R[ir]);
  44: R[0] ← AND(R[temp0], R[rs2]);      / Colocar la dirección de destino sobre el bus A y
      WRITE; GOTO 2047;                  /   el operando sobre el bus B
1810: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13 para calcular la dirección de destino
1811: R[temp0] ← ADD(R[rs1],R[temp0]);   / Calcular dirección de destino
      GOTO 1809;
      / Instrucciones de salto: ba, be, bcs, bvs, bneg
1088: GOTO 2;                            / Árbol de decodificación para saltos
   2: R[temp0] ← LSHIFT10(R[ir]);        / Extender el signo de los 22 bits menos
   3: R[temp0] ← RSHIFT5(R[temp0]);      / significativos de %temp0, desplazando
   4: R[temp0] ← RSHIFT5(R[temp0]);      / primero 10 bits a izquierda, luego 10 bits a derecha
   5: R[ir] ← RSHIFT5(R[ir]);            / La extensión de signo se realiza a través de RSHIFT5
   6: R[ir] ← RSHIFT5(R[ir]);            / Mover el campo COND a IR[13] utilizando tres veces RSHIFT5
   7: R[ir] ← RSHIFT5(R[ir]);            / La extensión de signo no afecta la operación
   8: IF R[IR[13]] THEN GOTO 12;         / ¿La instrucción es ba?
      R[ir] ← ADD(R[ir],R[ir]);
   9: IF R[IR[13]] THEN GOTO 13;         / ¿La instrucción no es be?
      R[ir] ← ADD(R[ir],R[ir]);
  10: IF Z THEN GOTO 12;                 / Ejecutar be
      R[ir] ← ADD(R[ir],R[ir]);
  11: GOTO 2047;                         / El salto indicado por be no se ejecuta
  12: R[pc] ← ADD(R[pc],R[temp0]);       / Ejecutar el salto
      GOTO 0;
  13: IF R[IR[13]] THEN GOTO 16;         / ¿Es bcs?
      R[ir] ← ADD(R[ir],R[ir]);
  14: IF C THEN GOTO 12;                 / Ejecutar bcs
  15: GOTO 2047;                         / El salto indicado por bcs no se ejecuta
  16: IF R[IR[13]] THEN GOTO 19;         / ¿Es bvs?
  17: IF N THEN GOTO 12;                 / Ejecutar bneg
  18: GOTO 2047;                         / El salto indicado por bneg no se ejecuta
  19: IF V THEN GOTO 12;                 / Ejecutar bvs
  20: GOTO 2047;                         / El salto indicado por bvs no se ejecuta
2047: R[pc] ← INCPC(R[pc]); GOTO 0;      / Incrementar %pc y empezar de nuevo
```

Figura 6.15 ∘ Continuación.

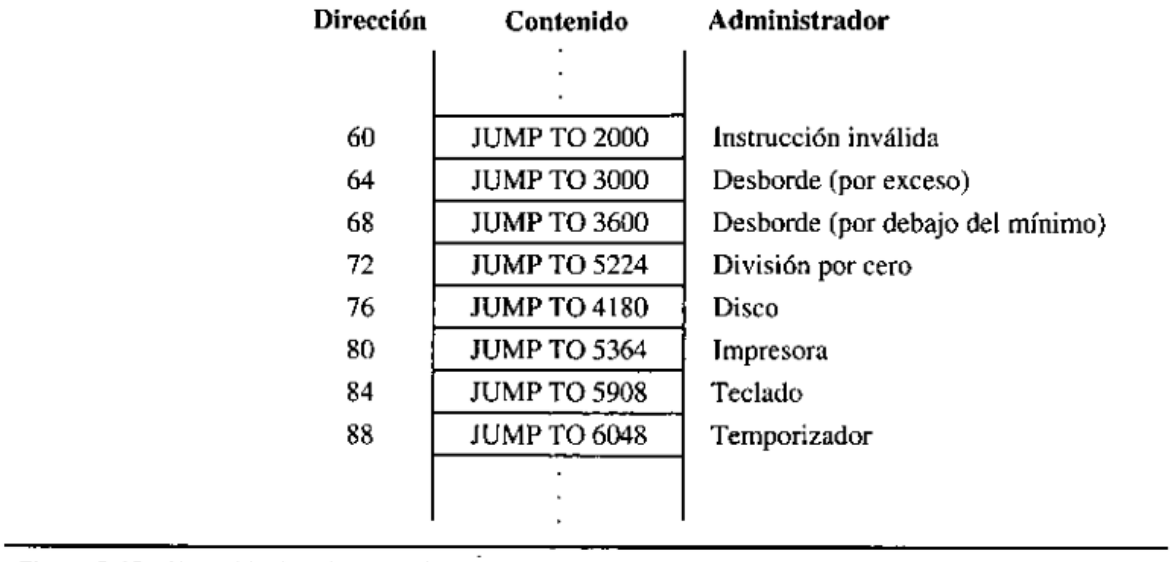
Con el objetivo de hacer posible la sincronización de la lectura/escritura de memoria (que generalmente es mas lenta que cualquier operación con registros) existe una señal de ACK proveniente de la memoria principal que indica a la unidad de control cuando la operación de memoria finalizó.

Un ejemplo de una decodificación de una instrucción: *addcc* tiene op=10 y op3=010000. Con el 1 a la izquierda y los dos ceros agregados queda: 11001000000 = 1600. Por lo tanto las microinstrucciones que ejecutan el addcc deberían comenzar en 1600. En la practica hay direcciones a las que no se debería caer nunca: se pueden usar para marcar el error o si nos quedamos sin lugar ejecutando una instrucción (ya que tenemos 4 palabras secuenciales disponibles para cada decode por los últimos 2 bits en 0).

La codificación para las instrucciones *sethi/branch/call* no tienen op3. Para mantener un mecanismo de decodificación se crean entradas duplicadas en la memoria de control (para cubrir todos los casos posibles sin importar lo que haya en el *%ir*). La otra opción es utilizar un decodificador adicional para cada tipo de instrucción que modifique como se calcula la dirección a leer de la ROM basado en el tipo de instrucción y haga los casos especiales para este tipo de instrucción

Traps e interrupciones

Un trap es un procedimiento automático de llamada generado por el hardware como consecuencia de una condición excepcional que se produce durante la ejecución de un programa. En general son errores inesperados que deben ser 'atrapados', como desbordes o división por cero. Cuando se produce, se transfiere el control a un administrador de traps, parte del OS que por ejemplo muestra un error y finaliza el programa. Normalmente existe una sección de memoria fija para los administradores de trap donde se almacena las palabras para manejar los traps, llamada *tabla de saltos*.



Dirección	Contenido	Administrador
	. . .	
60	JUMP TO 2000	Instrucción inválida
64	JUMP TO 3000	Desborde (por exceso)
68	JUMP TO 3600	Desborde (por debajo del mínimo)
72	JUMP TO 5224	División por cero
76	JUMP TO 4180	Disco
80	JUMP TO 5364	Impresora
84	JUMP TO 5908	Teclado
88	JUMP TO 6048	Temporizador
	. . .	

Figura 6.18 • Una tabla de saltos para las rutinas de atención de interrupciones y administradores de *traps*.

Las interrupciones son situaciones similares pero se producen luego de eventos llamados "excepciones" o "interrupts". Generalmente se los utiliza para la entrada o para marcar problemas con los dispositivos. La diferencia es que los traps son sincrónicos mientras que las interrupciones son asincrónicas. Un trap ocurrirá siempre en la misma parte de un programa. Un interrupt ocurrirá cuando el usuario decida apretar una tecla, por ejemplo.

Nanoprogramacion y nanoalmacenamiento

Se puede reducir el tamaño de la memoria de control cuando hay mucha repetición de palabras iguales, colocando las copias de las palabras en un único lugar e indexándolas. El microprograma tiene la misma cantidad de palabras pero cuando se utiliza *nanocódigo*, en la memoria de control se almacenan punteros (los índices de las palabras únicas) en lugar de palabras enteras de 41 bits. Si bien permite liberar superficie de memoria en la CPU, también implica una pérdida en velocidad asi que se deben tener en cuenta esto para tomar la decisión.

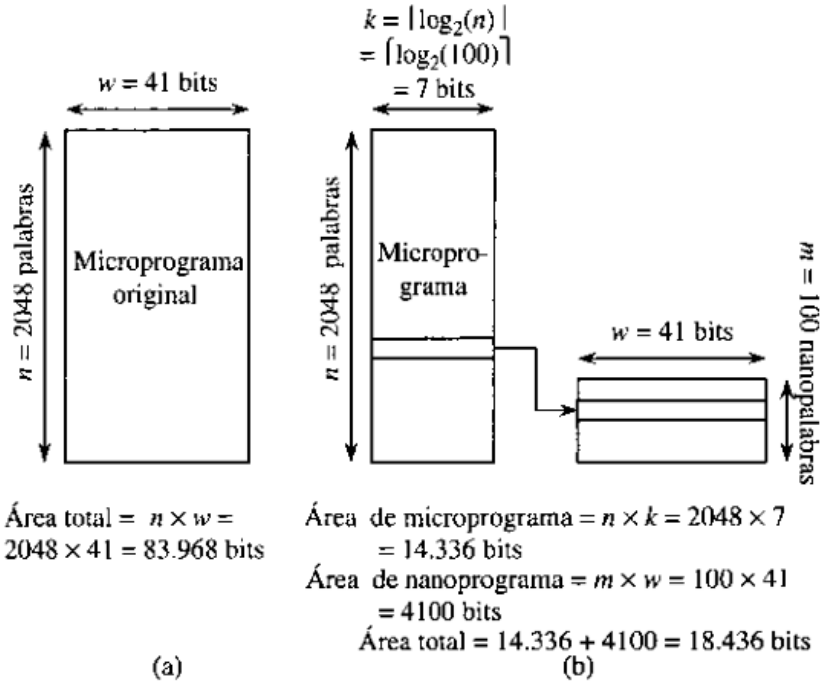


Figura 6.19 • (a) Microprogramación versus (b) nanoprogramación.

Episodio 7

Las jerarquías de memoria

En general hay distintos tipos de memorias, ya que se trata de balancear el costo vs. su velocidad de lectura/escritura. Típicamente, en una computadora se incluyen partes de memoria de cada una de los escalones de la pirámide de la jerarquía de memorias, donde la pirámide representa la cantidad de espacio que se dedica en la computadora para cada tipo de memoria:

Figura 7.1 • La jerarquía de las memorias.

Memoria RAM (*Random-access memory – memoria de acceso aleatorio)*

La memoria RAM es la que permite el acceso aleatorio es decir, a cualquier posición en tiempo constante independientemente de su tamaño. Se comienza estudiando como se puede crear una celda de 1 bit, para luego crear bytes, palabras y memorias enteras.

Funcionalmente, la celda debe tener 1 entrada de selección llamada *Chip Select* (CS) + 1 entrada que determina si la operación es de lectura (0) o escritura (1) y una línea bidireccional para leer/escribir el dato almacenado, según corresponda. Una posible descripción funcional es:

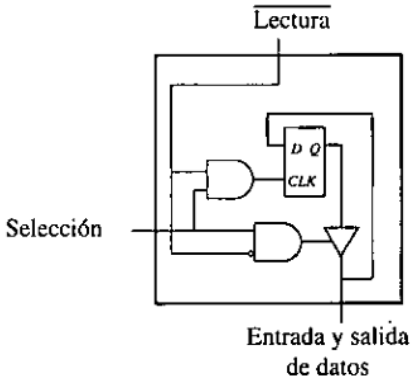


Figura 7.2 • Descripción funcional del comportamiento de una celda de memoria de acceso aleatorio.

Los circuitos de memoria basados en FF como el de la figura se conocen como **SRAM** (memoria estática), debido a que el contenido de cada bit se mantiene en tanto haya alimentación eléctrica.

Por otro lado, los circuitos de **DRAM** (memoria dinámica) utilizan un capacitor que almacena una pequeña cantidad de carga eléctrica para determinar si el bit representa un 0 o un 1. Los capacitores son mucho mas chicos que los FF, por eso se puede almacenar mucha mas memoria en la misma cantidad de superficie de DRAM que de SRAM. Sin embargo,

dado que las cargas de los capacitores se disipan con el tiempo, la carga de sus celdas tiene que ser reestablecida o *refrescada* periódicamente y con frecuencia. Es normal que las memorias dinámicas tengan circuitos extra de detección de errores debido a las descargas espontáneas que pueden sufrir los capacitores por interacción los rayos gamma del ambiente (son raros pero pasa).

Organización de un circuito integrado y construcción de memorias

Un circuito de memoria entero consiste de varias celdas que forman palabras que se leen conjuntamente y están indexadas para ser accedidos por dirección. Por eso, esquemáticamente cualquier circuito de memoria debe tener las siguientes entradas/salidas:

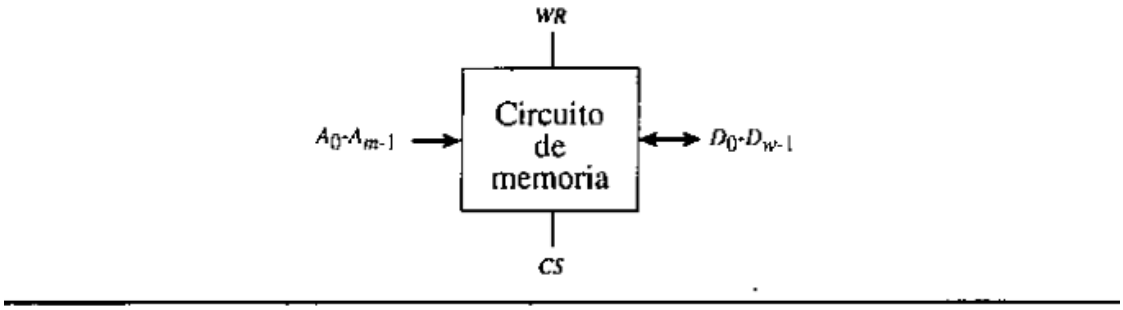


Figura 7.3 • Distribución simplificada de conexiones en un circuito integrado de memoria de acceso aleatorio.

Para una memoria con un espacio de direcciones de *m* bits, se debe elegir la palabra de *w* bits sobre la que se opera, indicado por las líneas A. La línea WR indica si la operación es de escritura o lectura, mientras que las líneas de dato son las indicadas como D. La línea de CS parece redundante pero permite construir memorias mas grandes a partir de memorias chicas (de manera similar a como con varios *Full-adder* se podía hacer un sumador de palabras arbitrarias). En este caso, el circuito tendría 2^m x *w* bits de capacidad.

Notar que cualquier circuito eléctrico tiene retardo en su operación (compuertas, propagación, etc), por lo que cuando se lee se tiene que esperar a que todos los datos estén listos por un tiempo conocido, al igual que cuando se escribe se debe mantener los datos estables hasta que la escritura haya terminado.

Para ver un ejemplo, si queremos armar una memoria de 4 palabras (espacio de direcciones de 2 bits) de 4 bits cada una, una posible configuración es la siguiente:

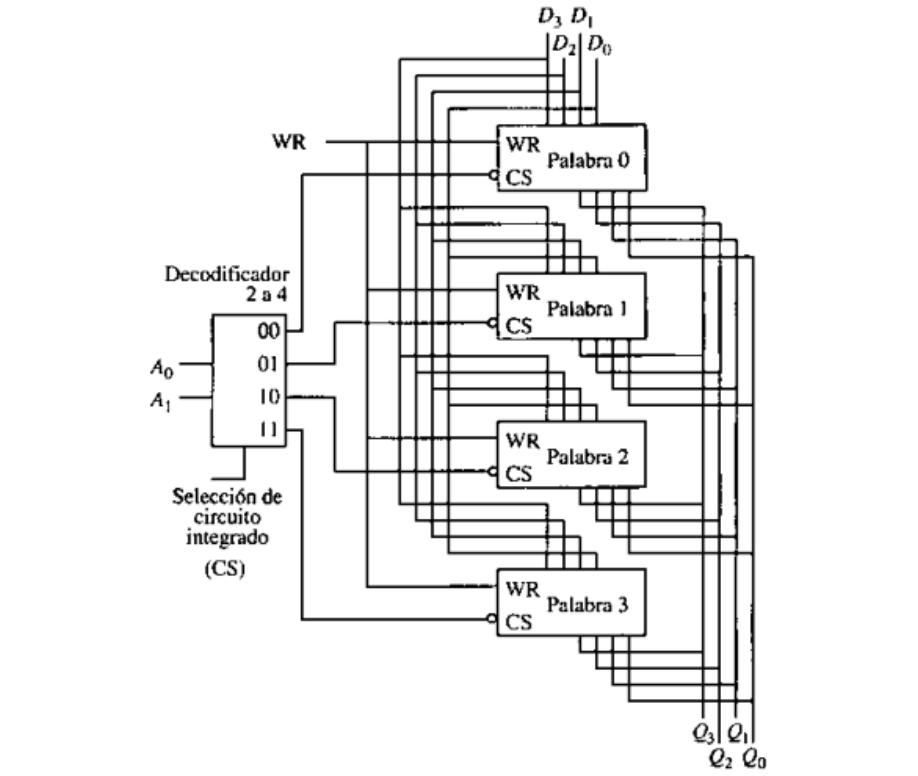


Figura 7.4 • Una memoria de cuatro palabras con cuatro bits por palabra y organización 2D.

El decodificador se encarga de seleccionar la palabra correcta dada la dirección codificada. Por otro lado, para simplificar las cosas se piensa a las palabras como registros con una entrada (D) y una salida (Q), ambas de 4 bits, el ancho de la palabra en esta memoria en particular. Las salidas de los registros están interconectadas porque como se explicó antes, se utilizan tristates en su estado 'z' cuando no se selecciona la palabra (CS=0) para evitar conflictos de superposición de señales.

Como la mayoría de los circuitos de memoria son prácticamente cuadrados, una estructura de decodificación alternativa reduce la complejidad del decodificador por dos al realizar la decodificación de filas y columnas en partes separadas. A esta organización se la conoce como 2-1/2D y es la mas utilizada en la práctica.

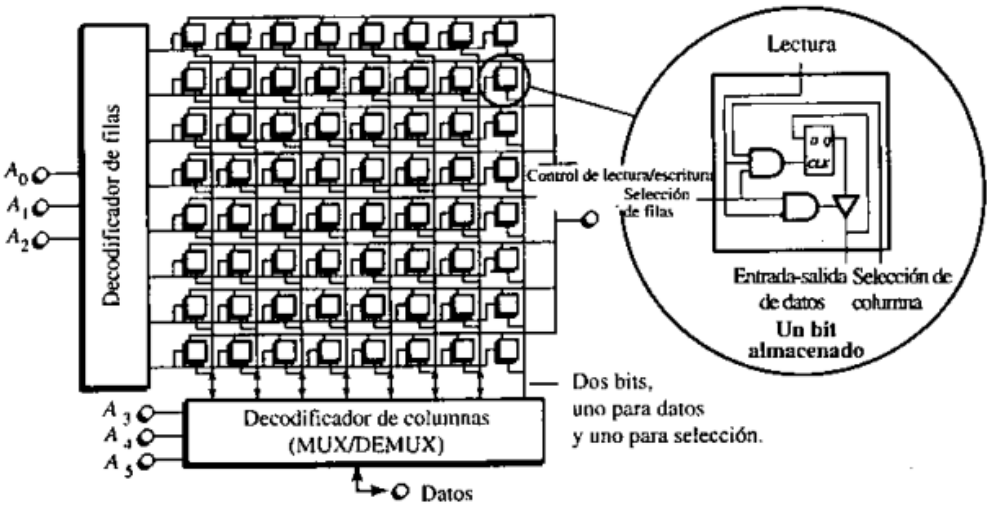


Figura 7.6 • Organización 2-1/2D de una memoria de 64 palabras por 1 bit.

Para localizar correctamente la palabra buscada se debe multiplexar la dirección de la fila y de la columna por separado, respectivamente con una señal que indique qué operación se está realizando: un RAS (Row address strobe) o un CAS (Column address strobe). Además, puede haber otras terminales para funciones extra como el refresco de la memoria dinámica.
Una manera de mejorar los tiempos de acceso es el **entrelazamiento**, donde se accede a varias direcciones de memoria simultáneamente.
Para construir memorias mas grandes a partir de mas chicas tenemos dos posibilidades: incrementar la cantidad de palabras o incrementar el tamaño de la palabra. Las dos se ilustran en las figuras:

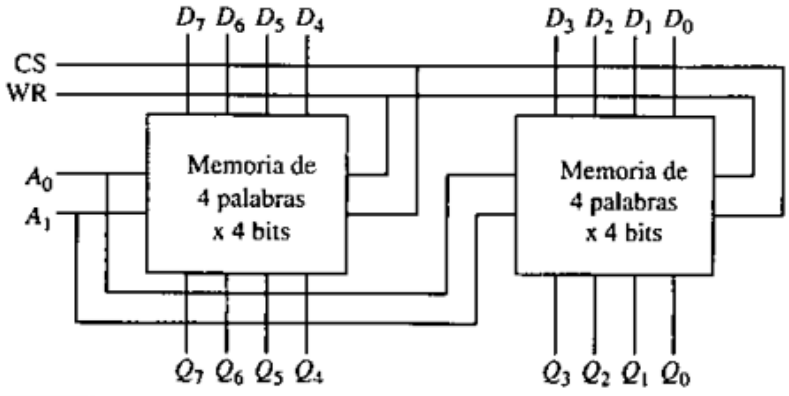


Figura 7.7 • Dos memorias de cuatro palabras por cuatro bits utilizadas para crear una memoria de cuatro palabras de ocho bits.

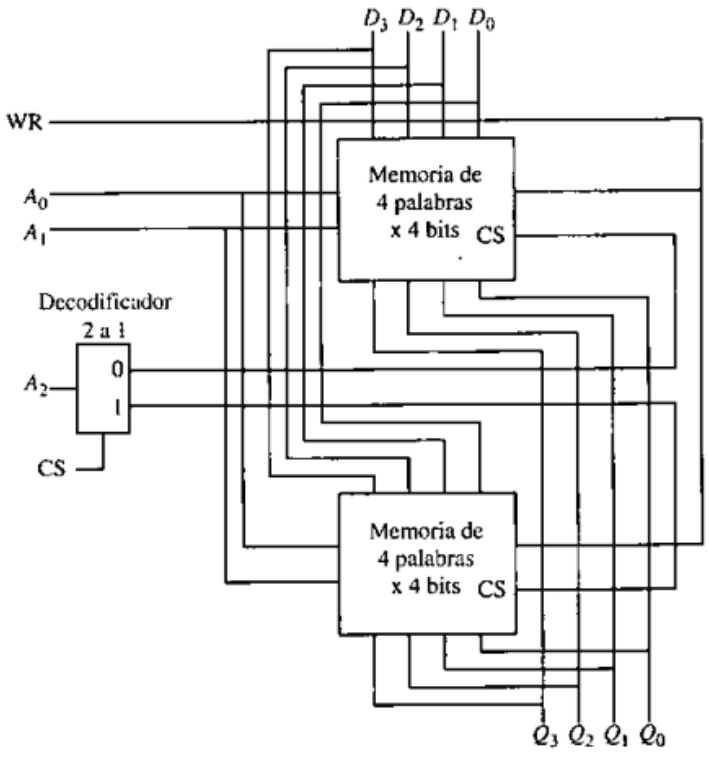


Figura 7.8 • Implementación de una memoria de ocho palabras de cuatro bits con dos memorias de cuatro palabras de cuatro bits cada una.

Memoria ROM (Read-only memory)

Como se vio en el caso de la ALU, cuando se quiere guardar información que a la que no se escribe, se la puede representar mediante una función lógica cuya salida es determinada respecto de su entrada (en este caso la dirección nos devuelve siempre el mismo dato). Se puede implementar utilizando los circuitos típicos de compuertas lógicos o se

pueden usar memorias programables (P-ROM) que se escriben una única vez, EPROM (memorias programables re-escribibles, que pueden ser borradas completamente como las UVPROM, mediante rayos UV). También existen las memorias EEPROM (borrables eléctricamente), como las memorias flash.

Memoria cache

Según el *principio de localidad*, hay una alta probabilidad de que un programa haga referencia a la misma posición de memoria repetidas veces (localidad temporal) o a posiciones de memoria cercanas (localidad espacial). Para tomar ventaja de este hecho, cada vez que se accede a la memoria RAM se hace una copia (si no existe) en una memoria mas rápida y pequeña (el cache), reduciendo los tiempos de acceso si se cumple el principio de localidad. Aunque seria mas lento en el caso de que se accedan repetidamente a direcciones diferentes de memoria (implicando hacer una copia al cache cada vez), el principio de localidad se cumple en un porcentaje de veces muy alto y como regla general es beneficioso para la velocidad de ejecución.
Los sistemas modernos poseen distintos niveles de memoria cache (L1, L2 y hasta L3), mejorando aun mas las velocidades de acceso. Otra ventaja de las memorias cache es que están físicamente mas cerca de la CPU, mejorando los tiempos de demora por propagación. Además, se pueden usar buses mas veloces, como ilustra la figura:

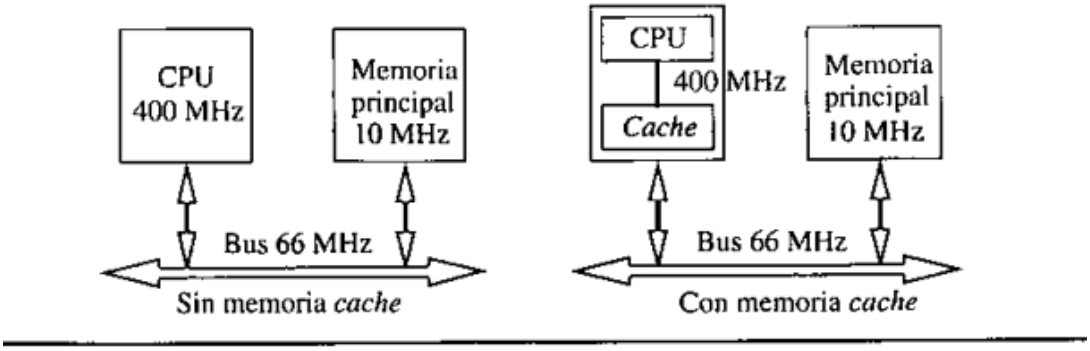


Figura 7.12 • Ubicación de una memoria *cache* en un sistema de computación.

Las frecuencias indicadas en Hz representan la velocidad máxima de CLK con la que puede operar cada parte del sistema, por lo que si se deben comunicar efectivamente el componente con el CLK mas lento es el que va a determinar la velocidad final de la operación. En este caso, si se evita el acceso a memoria principal mediante un acceso únicamente al cache para la operación de memoria, entonces se puede realizar la operación con un CLK de 400 Mhz.