[66.70] Estructura del computador

Resumen para el coloquio

Francisco Pirra

Unidad 4

Lenguaje de máquina

Bus

- Se utiliza para reducir la cantidad de interconexiones entre la **CPU** y sus subsistemas
- La **CPU** genera direcciones que se envían por el bus, mientras que la memoria las recibe, nunca se da el caso inverso

Memoria

- Conjunto de registros numerados, y cada uno almacena 1 byte
    - ***Locación de memoria****: Dirección de cada registro en memoria*
    - ***Nybble****: Conjunto de 4 bits*
    - ***Byte****: Conjunto de 8 bits*

- Si tenemos que almacenar palabras de más de un byte, hay dos formas, veremos un ejemplo de como escribir en cada uno el número 13 (en hexa: 0x3133) y la cadena 'trece' (en hexa: 0x74726563650d0a) expresados ambos en hexadecimal:
    - **Little endian** (Usado por Intel): El byte menos significativo, se almacena en la dirección más baja de memoria *(se almacena '**al revés**' de lo natural')*
      Para el ejemplo: 0x33 0x31 y 0x0a 0x0d 0x65 0x63 0x65 0x72 0x74

    - **Big endian** (Usado por Apple): El byte más significativo, se almacena en la dirección más baja de memoria (Osea se almacena en el **mismo orden**)
    - Para el ejemplo: 0x31 0x33 y 0x74 0x72 0x65 0x63 0x65 0x0d 0x0a

- Los dispositivos de entrada/salida se tratan como posiciones de memoria, por lo que la lectura y escritura en dichos dispositivos se trata de la misma forma que con la memoria
- Una memoria con espacio de direcciones de 32 bits, puede direccionar una capacidad máxima de 2^{32} bytes, osea 4 Gbytes

CPU

- Unidad de control: Ejecución de instrucciones del programa, almacenadas en la memoria principal. Está compuesta por:
    - Program counter (**PC**): Contiene la dirección de la próxima instrucción a ejecutarse
    - Instruction register (**IR**): Contiene la instrucción que se está ejecutando
    - Para ejecutar un programa se realiza el Ciclo de **fetch** (Se verá en capítulo 6)

ARC

ARC (A **RISC** Computer) es un subconjunto de la arquitectura SPARC
Características de la arquitectura ARC:

- Memoria
  - Utiliza el almacenamiento **big endian**
  - Las primeras 2048 direcciones se reservan para el sistema operativo
  - La pila o stack comienza en la dirección 2^{31}- 4, y sigue **decreciente**
  - El espacio entre la dirección 2^{31} y 2^{32}- 1, se reserva para **dispositivos de entrada/salida**. Estos dispositivos están **MAPEADOS** en memoria, por lo tanto los últimos 2GB están dedicados a los dispositivos y no pueden ser utilizados para otra cosa.
    Si se coloca más de 2GB de memoria, estos se pisaran con los 2GB de los dispositivos de entrada/salida y quedarán inutilizables

- CPU
  - Tiene un total de **32 registros**, numerados de **%r0** a **%r31** de 32 bits cada uno; además del **%pc** (Program counter) y el **%psr** (Processor Status Register)
  - En el **%psr** se indican los flags provenientes de la ALU
    - z = 1, cuando el resultado es cero
    - n = 1, cuando el resultado de la operación es negativo
    - c = 1, cuando la operación genera arrastre (Importa sin signo)
    - v = 1, cuando la operación causa desborde (Importa con signo)
  - Cada instrucción ocupa 32 bits
  - ARC es una arquitectura de *load/store.* Las instrucciones deben cargarse en el **IR** antes de ser ejecutadas, así como los operandos necesarios para ejecutarlas deben ser antes cargados en registros. En otras palabras, no se opera con la memoria principal, sino con el contenido de los registros únicamente. También se escribe a memoria solo desde los registros

- Set de instrucciones
  - Movimiento de datos
    - **ld** (Load) : Carga una palabra, de la memoria a un registro
      - Sintaxis: **ld [x], %r1** ! Copia el contenido de la dirección de memoria de x en %r1

    - **st** (store) : Guarda una palabra, de un registro a la memoria
      - Sintaxis: **st %r1, [x]** ! Copia el contenido del registro %r1 en la dirección de memoria x

    - **sethi** : Carga en los 22 bits más significativos de un registro, la constante que le pasamos con la instrucción

- Sintaxis: **sethi 0x304f15, %r1** !Carga en los 22 bits más significativos de %r1 el valor hexadecimal 304f15
- **%hi(valor)**: Devuelve los 22 bits más significativos de una constante
- **%lo(valor)**: Devuelve los 10 bits menos significativos de una constante
    - Estas dos funciones las vamos a utilizar para cargar números mayores a 13 bits en un registro, como por ejemplo una posición de memoria, ya que la instrucción **ld** no soporta más de 13 bits

- Aritmeticas y logicas
    - **and** (y): Producto lógico bit a bit
        - Sintaxis: **and %r1, %r2, %r3** !Producto entre %r1 y %r2, que se almacena en %r3

    - **or** (o): Suma lógica bit a bit
        - Sintaxis: **or %r1, 1, %r1** !Le suma 1 al bit menos significativo del registro %r1 (Como es suma lógica, entonces significa que si esta en 0 lo pone en 1, y si esta en 1 lo deja en 1)

    - **orn** (nor): Suma lógica negada bit a bit
        - Sintaxis: **orn %r1, %r0, %r1** !Suma 0 a cada bit, lo que no modifica el valor de %r1 y luego lo niega bit a bit, osea que calcula el complemento de %r1

    - **srl**: Desplazamiento lógico a derecha, de 0 a 31 posiciones
        - Sintaxis: **srl %r1, 3, %r2** !Corre a derecha 3 posiciones el contenido de %r1 y lo almacena en %r2 (Los 3 bits que quedaron “vacíos”, que son los más significativos, se completan con 0)
        - Tambien existe un desplazamiento que en vez de logico es aritmetico, el **sra**, que completa con 0 o 1 dependiendo del bit más significativo, extendiendo el signo

      Nota: Para hacer un shift hacia la izquierda se multiplica por dos, haciendo un add del número con sí mismo

    - **add**: Suma en complemento a dos, ambos operandos
        - Sintaxis: **add %r1, 5, %r1** !Suma 5 al contenido de %r1

  *Nota: **andcc**, **orcc**, **orncc** y **addcc**, tienen el mismo funcionamiento que lo antes expuesto, solo que al agregar el ‘cc’ al final, la instrucción va a cagar los flags del **%psr**. Sino, no carga los flags*

- Instrucciones de control

        - **call**: Llamado de subrutina. Invoca una subrutina y almacena la dirección de la instrucción actual en **%r15**, es un “llamado y enlace”
            - Sintaxis: **call sub_r** !Invoca la subrutina que comienza en la posición sub_r de memoria

        - **jmpl**: Retorno de subrutina. Salta a una nueva dirección y almacena la dirección de la instrucción actual en el registro de destino
            - Sintaxis: **jumpl %r15 + 4, %r0** !Como el valor del contador de programa al momento de la instrucción **call** quedó almacenado en %r15, la dirección de retorno es la siguiente al call, osea %r15 + 4 (recordemos que una instrucción tiene 32 bits, osea 4 bytes)

        - **be**: Salta a la etiqueta indicada, si ambos operandos son iguales (Lo que se hace para verificar si son iguales es restar los operandos con **subcc**, descartando el resultado, para poder observar el flag **z**, el **be** verifica si el flag **z** está en cero para poder saber si la resta dio cero, entonces confirmar que son iguales)
        - **bcs**: Salta a la etiqueta si el flag **c** está en 1
        - **bneg**: Salta si el flag **n** está en 1
        - **bvs**: Salta si el flag **v** está en 1
        - **ba**: Salta a la etiqueta indicada, sin importar los flags
    - *Nota: La sintaxis de **be**, **subcc**, **bcs**, **bneg**, **bvs** y **ba**, es siempre igual. Se coloca, por ejemplo **be etiqueta_1** lo que realiza un salto a la etiqueta_1 si se cumple la condicion, o en el caso de **ba**, lo realiza siempre*

- Formato del lenguaje simbólico de **ARC**
    - Es la ‘sintaxis’ que va a tener el código que escribamos, y va a tener un orden de izquierda a derecha, que voy a escribir acá de arriba hacia abajo:
        - Rótulo: Sería una etiqueta, debe terminar con dos puntos (por ejemplo start: , lab_1: , rotulo1: , etc)
        - Nombre de la instrucción
        - Operando de origen: Registro o dirección de memoria de origen
        - Operando de destino: Registro o dirección de memoria de destino
        - Comentario: Se escribe comenzando con un signo de exclamación !

    - Registros especiales:
      Es importante notar que si bien, los registros cumplen diferentes funciones, y tenemos registros especiales, todos tienen la misma estructura logica interna de compuertas SALVO el **%r0**, que la diferencia es que como no se modifica, no tiene que estar conectado al bus que le escribe a el.
        - **%r0** : Siempre está en 0, no puede modificarse (se usa para enviar datos que se quieren desechar)
        - **%r14** : Es el puntero del stack (%sp stack pointer)
        - **%r15** : El el registro de enlace (link register)

- Formatos de instrucción en ARC
    - Es la manera en la que el ensamblador distribuye los diversos campos de una instrucción, osea como interpreta los 32 bits de la instrucción
    - En ARC solo existen 5 formatos :
        - SETHI
        - Salto (branch)
        - Llamada (call)
        - Aritmético
        - Memoria
    - Vamos a analizar, en rasgos generales, cómo usar la tabla de formatos de instrucción de ARC
        - **op**: Indica que formato utilizar
            - 00 = SETHI/branch, 01 = call, 10 = Aritmético, 11 = Memoria
            - *Para op = 00 vamos a estar en un SETHI/branch y luego el valor de **op2** nos dirá si es un SETHI (100) o un branch (010)*
        - **op3**: Indica la operación a realizar
        - **cond**: Indica el tipo de salto condicional, en base a los flags del %psr, que son 4 bits
        - **imm22**: Contiene el valor que vamos a utilizar en el SETHI
        - **disp22**: Contiene el valor para calcular la dirección de salto
        - **disp30**: Contiene el valor para dirigirnos a otra dirección
        - **rd**: Según en el formato que estemos, tiene diversos usos:
            - Para SETHI, es el registro al cual se le aplicará el SETHI
            - Para **st**, de Memoria, es el registro de origen
            - Para todo lo demás, es el registro de destino
        - **rs1**: Primer registro de origen
        - **rs2**: Segundo registro de origen
        - **simm13**: Es un valor de 13 bits, que se copia en una palabra de 32 bits y se extiende con signo hasta completar los 32 bits, si el campo **i** está en 1

- Directivas
    - Pseudo operaciones, que son instrucciones dirigidas al ensamblador, siempre comienzan con un punto
    - Son procesadas en el momento del PRE-ensamblado, no durante el proceso de compilación
    - Las más importantes son:
        - .equ : Define una constante (X .equ 9 => X vale 9)
        - .begin : Comienza el programa
        - .end : Termina el programa
        - .org : Cambia el contador de posición (.org 2048 => lo cambia a 2048)

- Subrutinas
    - Son lo que en otros lenguajes de programación llamamos funciones o procedimientos

- Si se utilizan rutinas o etiquetas definidos en módulos separados, se debe utilizar:
  - **.global <etiqueta>** en el módulo que se define
  - **.extern <etiqueta>** en el módulo que va a ser usada

- Existen tres convenciones para las llamadas a subrutinas:
  - **La primera**, simplemente coloca los argumentos que se le van a pasar a la subrutina en registros. No funciona si la cantidad de argumentos a ser transferidos excede el número de registros disponibles

    Ejemplo de esta convención

```
                ld      [x], %r1
                ld      [y], %r2
                call    add_1
                st      %r3, [z]

x:              1
y:              2
z:              0

add_1:          addcc %r1,%r2, %r3
                jmpl    %r15 + 4, %r0
```

  - La segunda, utiliza la **zona de transferencia de datos**. La dirección de dicha zona, es entregada a la rutina en un registro predeterminado

```
                ld      [x], %r1
                ld      [y], %r2
                st      %r1, [k]        ! Guardo en la primer pos de la
                                        ! zona de transf de datos, el
                                        ! valor del registro 1
                st      %r2, [k+4]      ! Guardo en la siguiente pos
                sethi   k, %r5
                srl     %r5, 10, %r5    ! Corro a derecha 10 bits el
                                ! contenido de r5, de modo que los
                                ! 22 que le cargo el sethi pasan a ser
                                ! los 22 menos significativos… de esta
                ! forma estamos limitados a posiciones de
                ! memoria de hasta 22 bits, para usar más, se
                ! puede usar el método explicado para cargar
                ! posiciones de 32 bits
                call    add_1
                ld      [k+8], %r3      ! Obtengo el resultado
                st      %r3, [z]        ! Guardo el resultado en z

x:              1
y:              2
```

```
z:          0
k:          .dwb 3      ! Defino la zona de transf, con 3 posiciones

add_1:
            ld      %r5, %r11       ! Cargo desde la zona de transf
            ld      %r5+4, %r12     ! Cargo desde la zona de transf
            addcc   %r11,%r12, %r13     ! Opero
            st      %r13, %r5 + 8       ! Guardo en la 3er posición
            jmpl    %r15 + 4, %r0       ! Vuelvo a la rutina invocante
```

- **La tercera, utiliza una pila**. La idea es que el programa invocante empuja (push) todos sus argumentos en una pila del tipo LIFO. La rutina invocada, extrae de la pila los argumentos transferidos, a la vez que coloca en la misma los valores a retornar. Por último el programa invocante rescata de la pila los valores devueltos por la rutina.
  El registro **%sp** (stack pointer), contiene la dirección de la cabeza de la pila, que va a ser el registro **%r14**
  Para utilizar la pila, conviene definir las **macros** push y pop:

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

.macro pop arg
        ld %r14, arg
        add %r14, 4, %r14
.endmacro

        ld      [x], %r1
        ld      [y], %r2
        push    %r2     ! Mando a la pila el segundo valor
        push    %r1     ! Mando a la pila el primer valor
        call    add_1   ! Llama a la subrutina
        pop     %r3     ! Obtengo de la pila el resultado
        st      %r3, [z]        ! Guardo el resultado, en la
                                ! posición de memoria z, osea
                                ! cambia el valor de z, que hasta
                                ! ahora estaba en 0.
x:          1
y:          2
z:          0

add_1:
        pop %r11        ! Obtengo de la pila el primer valor
        pop %r12        ! Obtengo de la pila el segundo valor
        addcc %r11,%r12, %r13
```

```
push %r13      ! Mando a la pila el resultado
jmpl   %r15 + 4, %r0 ! Vuelvo a la rutina invocante
```

Puntos a tener en cuenta, definiciones y comentarios

- Las constantes que utilicemos están restringidas a 13 bits.
    - Por ejemplo, si queremos cargar lo que hay en la posición de memoria 0x4000, no podemos hacer “ld 0x4000, %r1” 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 más significativos, y luego con un **and** o **add** cargar los 10 bits menos significativos

```
sethi   %hi(temp), %r7        ! Seteo los 22 mas sign.
add     %r7, %lo(temp), %r7   ! Le agrego los 10
ld      %r7, %r8              ! Cargo el contenido de la
                              ! posición de memoria en el
                              ! registro r8
```

            - 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

- Para entender cómo funciona la “ Declaración de variables y constantes”
    - En la posición de memoria de x, o de y, coloco un valor:
        - ***x: 0xFFFFFF*** *-> int x = 0xFFFFFF* *! Defino la variable x, y le asigno ! el valor **0xFFFFFF***
        - ***y: 4*** *-> int y = 4* *! Defino la variable y, y le asigno ! el valor **4***
    - Defino que cuando escriba x en el código, luego del preensamblado, quede reemplazado por la constante elegida:
        - ***x .equ 0xFFFFFF*** *-> #define X 0xFFFFFF* *! Definir la constante x ! como **0xFFFFFF***

- Si tengo una variable definida, y quiere cargar el valor de dicha variable a un registro:

```
ld      [x], %r1      ! Carga de la posición de memoria x, el contenido a r1
ld      [y], %r2      ! Carga de la posición de memoria y, el contenido a r1
x:      1             ! Defino la variable x, con valor 1
y:      2             ! Defino la variable y, con valor 2
```

- Obtener N bits de una palabra de 32 bits:

    - Se utilizan MÁSCARAS (La forma fácil, es pasar a binario escribiendo los 32 bits, y poniendo “1” en los bits que nos interesa obtener, y “0” en los que no; luego de esto, pasamos dicho valor a hexadecimal)
        - Para obtener los 5 bits más significativos de una palabra de 32, se hace un AND con la mascara 0xF8000000 (recordando que para armar esta máscara tenes que usar 'sethi')
        - Para obtener los 5 bits menos significativos se usaría 0x0000001F
        - Para obtener los dos bytes (1 byte = 8 bits) del medio, 0x00FFFF00

- Las posiciones de memoria se pueden escribir, tanto en hexadecimal, como en decimal, la única diferencia es su mejor legibilidad

Preguntas de examen

En general, lo referido al capítulo 4 serán ejercicios prácticos, de programar en ASM

- Indique si considera que el procesador ARC es de tipo CISC o RISC y compare ambos tipos de arquitecturas
    - El procesador ARC es del tipo RISC, ya que su mismo nombre lo indica (**A** **R**isc **C**omputer)

CISC (Complex Instruction Set Computing)	**RISC** (Reduced Instruction Set Computing)
Instrucciones complejas	Instrucciones simples
Código binario de largo variable	Código binario de largo fijo
Código binario de largo variable	Direcciones de memoria múltiplos de 4
Mucha velocidad	Poca velocidad
Las funciones complejas se almacenan en transistores	Se utilizan transistores para registros de memoria

*En base al cuadro es fácil ver que ARC es RISC, ya que tiene **instrucciones simples**, **código binario fijo de 32 bits**, direcciones de **memoria múltiplos de 4** bytes ya que son de 32 bits, y las únicas dos instrucciones que pueden acceder a memoria principal son **load y store** (leer de memoria a un registro, escribir de un registro a memoria; respectivamente)*

- Un conjunto de instrucción simple (Instrucciones cómo add, or, add, etc)
    - Velocidades más altas
    - Tamaño reducido del procesador
    - Consumo de energía reducido

    - Un conjunto complejo (Instrucciones cómo senos, cosenos, raíz, etc)
        - Optimizar operaciones comunes
        - Mejorar memoria/eficiencia de caché
        - Simplificar la programación

- Explicar el concepto de bus, cuales son sus ventajas. Explicar que es Bus de sistema, y compararlo con el modelo Von Neumann
    - El bus es el sistema es un canal compartido, por el cual se comparte la información de la CPU, memoria y entrada/salida
      Está constituido por:
        - Bus de datos: Transporta la información
        - Bus de direcciones: Determina hacia dónde enviar la información
        - Bus de control: Describe la transferencia de información
    - La ventaja del bus de sistema es que reduce el número de interconexiones entre el CPU y sus subsistemas
    - El modelo de Von Neumann en cambio, tiene más conexiones y tiene la unidad de control y de salida como dos líneas distintas, así como también la ALU y la unidad de control están separadas (Osea más interconexiones)

- Explica las ventajas perseguidas por las siguientes características de los procesadores RISC:
    - Todas las instrucciones se codifican con la misma cantidad de bits
        - Simplifica la lógica para encontrar las instrucciones en memoria
        - Simplifica la codificación de las instrucciones
        - Estandariza el ancho de buses internos

    - Entradas/salidas mapeadas en memoria
        - Se requiere menos lógica interna
        - CPU más barata, rápida y fácil de construir

Capítulo 5

Compilación, ensamblaje y linking

Compilación

- Pasos de compilación:
    - Análisis lexicográfico: Reconocer los elementos básicos del lenguaje como identificadores, definiciones y delimitadores

    - Análisis sintáctico: 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 semántico:
        - 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

    - 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

- Globales (estáticas): Son accesibles durante el curso de todo el programa y están fijas en memoria en una ubicación **conocida** al momento de compilación

- Locales (automáticas): 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. Generalmente se implementan usando una pila LIFO (last in, first out)

- Direccionamiento base: Se accede a las variables usando una copia del %sp (stack pointer), llamada %fp (frame pointer), 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 automáticas, sólo se conoce su posición en memoria como un desplazamiento respecto del %fp.

Estructuras (Structs)

- Se define y declara una instancia ‘p’: struct point { int x, y, z; } p;
    - Para acceder a los elementos de la estructura lo hacemos como **p.x** o **p.y**.
    - La dirección en memoria de la estructura, coincide con la dirección de su primer elemento. Entonces en este caso, la estructura p, tendría la misma dirección que **p.x**, mientras que **p.y** tendrá la dirección de **p+4** (ya que cada elemento pesa 32 bits, osea 4 bytes por eso sumo 4) y **p.z** la dirección **p+8**

Vectores (Arrays)

- Se los define con un tipo homogéneo (osea que todos los elementos del array son del mismo tipo) y un tamaño “n” conocido al momento de compilación
- La dirección del elemento con índice ‘i’, se calcula como un offset con respecto al primer elemento:
    - Posición del i-ésimo elemento: base + (i – start)*size, dónde 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 int)

Secuencias de control

Para implementar las bifurcaciones condicionales (if, while, for) se utilizan los flags del **%psr** y los saltos condicionales
Veamos el equivalente en ASM de algunas sentencias conocidas:

- **goto**: Se suele utilizar para ir a alguna parte del programa, sin condición previa
    - En ASM:
        - **ba etiqueta_deseada** !Realiza un salto a etiqueta_deseada

- **if-else**: Si se cumple la condición, se ejecuta un código, y sino, el otro
    - En ASM:
        - **subcc %r1, %r2, %r0** !Resta ambos registros, y fija los flags
        - **bne Over** !**bne** verifica el flag **z**, si esta en 0 salta a **Over**

        - !Si son iguales, no salta a **Over** y ejecuta lo que escribimos acá
        - **ba End** !Una vez ejecutado el código, salta a **End**
        - **over:** !Aca va el código, que se ejecuta si no son iguales
        - **End**: !Aca va el codigo, luego de que termina el **if**

- **while**: Mientras se cumpla la condición, se ejecuta un código
    - En ASM:
        - **ba Test** !Salto incondicional a la etiqueta **Test**
        - **True: add %r3, 1, %r3** ! Le suma 1 al registro **%r3**
        - **Test: subcc %r1, %r2, %r0** ! Resta **%r1** y **%r2**
        - **be True** ! Si el flag **z** está en 1, salta a **True**

- **for**: Se implementa de forma similar al **while**

Ensamblado

- Traducción del programa en lenguaje simbólico (assembler) a código binario.
- Es un proceso **directo** ya que las instrucciones en ASM se corresponden directamente con instrucciones del procesador.
- **Prestaciones de un ensamblador**
    - Permitir al programador especificar ubicación en memoria 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 etiquetas para direcciones de memoria constantes
    - Proveer macros y otras opciones, como la dirección de comienzo del programa, la posibilidad de hacer aritmética básica y lógica pre-ensamblado

- **Tabla de símbolos**
  La tabla de símbolos en ARC consta de 2 campos:
    - **Símbolo**: Todos aquellos que luego tengan dos puntos (etiquetas, variables, etc)
    - **Valor**: La posición en la que se encuentra cada símbolo, cada “renglón” que contamos, le suma 4 al valor de la posición. Tener en cuenta que las pseudoinstrucciones, osea las que procesa el pre ensamblador, no generan código, por lo que no se les asigna dirección, asique no suman nada.
    - Los valores se cuentan en **BYTES** (Recordemos que 1 byte = 8 bits), por lo tanto, incrementamos 4 bytes, lo que equivale a 32 bits, osea el tamaño de una instrucción

- **Ensambladores de dos pasadas**:
    - Primer pasada (**Armado de tabla de símbolos**):
        - Se determinan 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

        - Expande las macros e inserta cualquier definición de identificadores (etiquetas y constantes) en la tabla de símbolos, junto con su ubicación en memoria (según el location counter)

    - Segunda pasada:
        - Se usa para permitir el uso de símbolos dentro de un programa, con anterioridad a que sean definidos.
        - Cada instrucción es convertida a código de máquina
        - Cada identificador es reemplazado por su ubicación en memoria según indica la tabla de símbolos
        - Cada línea es procesada completamente antes de avanzar a la siguiente
        - Genera el código objeto y el listado

- Las salidas del ensamblador
    - **Archivo de texto (Listado)**
        - Tabla de símbolos
        - Listado
            - Código objeto
            - Location counter
            - Instrucciones

    - **Archivo con código objeto**:
        - Código de máquina
        - Dirección de primera instrucción a ejecutar
        - Símbolos declarados en otros módulo: externos
        - Símbolos globales: accesibles desde otros módulos
        - Librerías externas que son utilizadas por el módulo
        - Información sobre la relocalización del código
            - Cuando se combinan módulos, la mayoría deben reubicarse en memoria. Algunas direcciones se marcan como reubicables y otras no. Una excepción conocida sería la entrada/salida mapeada en memoria. Esta información se coloca en un diccionario de reubicación en el módulo ensamblado final, para el linker/loader

Linker

- Combina dos o más módulos que fueron ensamblados separadamente
- Resuelve referencias de memoria externa al módulo
- Relocaliza los módulos combinándolos y reasignando las direcciones internas
- Define en el módulo a cargar la dirección de la primer instrucción a ser ejecutada

Reubicación

- Si dos módulos se ensamblan por separado, no hay forma de que un programa ensamblador pueda descubrir el conflicto entre las direcciones de memoria durante el proceso de traducción, por lo cual se definen como reubicables a los símbolos que puedan admitir que su dirección se modifique durante el proceso de enlace
- **No son relocalizables**:
    - Direcciones de entrada/salida
    - Rutinas del sistema
- **Sí son relocalizables**:
    - Posiciones de memoria relativas a un .org (p.e.: x o y)

Loader

- Carga los diversos segmentos de memoria, con los valores apropiados e inicializa ciertos registros como el %sp (stack pointer) y el %pc (program counter) a sus valores iniciales
- Cuando hay varios módulos en ejecución, el loader debe reubicar estos módulos en el momento de la carga, para lo cual suma un desplazamiento a todo el código reubicable del módulo en cuestión
- Otra forma es usando la MMU (Memory Management Unit), que realiza la reubicación durante la carga, utilizando un registro base con respecto al cual todas las demás direcciones se reubican

Desde el diseño a la ejecución

Lenguaje de alto nivel **->(Se compila)** Assembler **->(Se ensambla)** Código de máquina, módulos **->(Linkeo)** Código Ejecutable **->(Loader)** Programa en ejecución

Preguntas de examen

- Indicar la información que reciben los programas ensambladores y la salida que generan. Repetir análisis para linkers y loaders. Señalar los procesos o aplicaciones que invocan la ejecución de cada uno de ellos, y en qué instancia del proceso de compilación/ejecución del programa intervienen

    - **Ensamblador**: Encargado de hacer la traducción de instrucciones de Assembler en código de máquina. Esta es una traducción 1-1
        - **Entrada**: Los archivos con código fuente con instrucciones Assembler
        - **Salida**:Archivos con código de máquina, uno por cada archivo con código fuente que fue suministrado
        - **Invocado**: El programador, al querer ensamblar su código

    - **Linker**: Encargado de juntar todos los distintos archivos generados por el Ensamblador, y agregar todas las librerías (estáticas) necesarias, para generar un único archivo.
        - **Entrada**: Múltiples archivos con código de máquina
        - **Salida**: Un único archivo que contiene todo el código ensamblado, con sus distintos módulos ya relocalizados
        - **Invocado**: Luego de que todos los módulos hayan sido ensamblados

    - **Loader**: Encargado de cargar un ejecutable a memoria. Se agregan todas las librerías (dinámicas) y se las relocaliza. Luego, se inicializan los registros especiales (Stack Pointer, Frame Pointer)
        - **Entrada**: Archivo binario a ejecutar
        - **Salida**: Se encuentra cargado el archivo binario en memoria, con todas las referencias y direcciones a librerías dinámicas ya resueltas.
        - **Invocado**: El sistema operativo, previo a la ejecución de un archivo binario

- Explique en qué consiste ejecutar un programa almacenado en un archivo ejecutable, que problemas pueden surgir al hacerlo, de qué modo esto está resuelto y en base a qué información.
  Detalle los pasos previos a la creación del archivo ejecutable, que son responsables de que esta información esté disponible al tiempo de iniciar la ejecución
    - Ejecutar un programa es tarea del **loader** y para ello:
        - Carga los diversos segmentos de memoria, con los valores apropiados e inicializa ciertos registros como el %sp (stack pointer) y el %pc (program counter) a sus valores iniciales
        - Los pasos previos a la creación del ejecutable son el **ensamblado** y **linkeado**
        - Cuando hay varios módulos en ejecución, lo más probable es que no se puedan colocar todos contiguamente en memoria por lo que el loader debe reubicar estos módulos en el momento de la carga, y para ello hay dos formas:
            - Sumar un desplazamiento a todo el código reubicable del módulo en cuestión
            - Usando la MMU (Memory Management Unit), que realiza la reubicación durante la carga, utilizando un registro base con respecto al cual todas las demás direcciones se reubican
        - La información requerida para reubicar un módulo se almacena en la tabla de símbolos contenida en el módulo ensamblado (Archivo binario)

Capítulo 6

Trayecto de datos y control

Fundamentos de la microarquitectura

La microarquitectura está constituída por

- Unidad de control
- Registros
- ALU

Se distinguen dos enfoques de microarquitecturas distintos:

- Unidades de control **microprogramadas**
- Unidades de control **cableadas**

La funcionalidad de la microarquitectura, se basa en el ciclo de **fetch**:

1. Búsqueda en memoria de la próxima instrucción a ejecutar
2. Decodificación del código de operación
3. Búsqueda de operandos en memoria (Si los hay)
4. Ejecución de la instrucción y almacenamiento de resultados
5. Vuelve a comenzar el ciclo

Trayecto de datos

Compuesto por:

- 32 registros **(%r0 a %r31)**: Accesibles por el usuario directamente

- **Program counter (%pc)**: Solo se tiene acceso a través de call y jmpl, contiene la posición de la próxima instrucción (como solo puede obtener valores múltiplos de 4, los últimos dos bits, osea los menos significativos, pueden estar conectados a cero siempre)

- Registros temporales **(%temp0 a %temp3)**: No accesibles al usuario, son para interpretar instrucciones ARC

- **Registro de instrucciones (%ir)**: Contiene la instrucción **en ejecución**, y no es accesible por el usuario

- ALU (Unidad Aritmético Lógica)

La unidad aritmetico-logica (ALU)

Puede realizar 16 operaciones sobre los buses A y B, el resultado de cada operación se coloca en el bus C (a menos de que este bloqueado por el multiplexor que maneja el bus C, debido a colocar en él una palabra proveniente de la memoria de la máquina).

Operaciones sobre los buses (Importante para microcódigo):

- ANDCC(A, B): Toca los flags, producto **lógico** bit a bit entre los operandos
- ORCC(A, B): Toca los flags, suma **lógica** bit a bit entre los operandos
- NORCC(A, B): Toca los flags, suma **lógica** bit a bit, y el resultado negado bit a bit
- ADDCC(A, B): Toca los flags, suma aritmética entre los operandos, en complemento
- SRL(A, B): Desplaza a **derecha**, el contenido A, tantos bits como indique B
- AND(A, B): Producto **lógico** bit a bit entre los operandos
- OR(A, B): Suma **lógica** bit a bit entre los operandos
- NOR(A, B): Suma **lógica** bit a bit, y el resultado negado bit a bit
- ADD(A, B): Suma aritmética entre los operandos, en complemento
- LSHIFT2(A): Desplaza a **izquierda** el contenido de A, en 2 bits
- LSHIFT10(A): Desplaza a **izquierda** el contenido de A, en 10 bits
- SIMM13(A): Recupera los 13 bits menos significativos de A y pone 0 en los demás
- SEXT13(A): Extiende el signo de los 13 bits menos significativos de A, a 32 bits
- INC(A): Incrementa en 1 el valor de A
- INCPC(A): Incrementa en 4 el valor de A (suele usarse para el %pc)
- RSHIFT5(A): Desplaza a derecha 5 bits, y completa a izquierda extendiendo el signo

Entradas de la ALU:

- Dos entradas A y B de 32 bits (Que se relacionan con los buses)
- Una entrada F de control de 4 bits (Para elegir la operación a realizar)

Saludas de la ALU:

- Una salida de datos C de 32 bits
- Una salida de condicion de 4 bits (N, V, C, Z)
- Una señal que actualiza los bits del registro **%psr** con los bits de condición

Registros

Los registros están implementados con circuitos biestables D activados por flanco descendente.
Características:

- Cada registro tiene 32 bits de ancho
- La entrada del registro está conectada al bus C y para la entrada de CLK tenemos una compuerta **and** entre el **CLK** (señal de clock) y el decodificador C (lo que asegura que solo se escriba lo que hay en C, si el deco lo indica)

- Luego las salidas están conectadas a buffers triestados, las cuales conectan con los buses A y B
- El registro **%r0** no se conecta con el bus C, por lo que puede construirse usando solo buffers de tres estados, sin necesidad de utilizar la compuerta **and**
- Los decodificadores A, B y C tienen entradas de 6 bits y seleccionan 1 solo registro para cada uno de los buses A, B y C
- La salida 0 del decodificador C no esta usada (por lo de %r0) y además como solo hay 38 registros, y con 6 entradas tengo 2^6=64 posiciones, entonces las posiciones mayores a 37 quedan disponibles

Sección de control

Llamaremos **memoria de control** a la memoria incluida en el corazón de la unidad de control, una memoria de solo lectura **ROM** de 2048 palabras de 41 bits y contiene todas las líneas a controlarse.
Cada palabra de 41 bits es una **microinstrucción**.
La ejecución de microinstrucciones se controla a través de:

- **MIR** (Microprogram Instruction Register)
- Registro de estado (**%psr**), posee los flags
- **CBL** (Control Branch Logic) y multiplexor de direcciones de memoria de control

Formato de la palabra de microcódigo (Como se divide el **MIR**):

- A: Registro del bus A
- AMUX: Selecciona la entrada del decodificador A:
    - AMUX=0, el deco obtiene como entrada el registro A
    - AMUX=1, el deco obtiene como entrada el registro **rs1** del **%ir**

- B: Registro del bus B
- BMUX:Selecciona la entrada del decodificador B:
    - BMUX=0, el deco obtiene como entrada el registro B
    - BMUX=1, el deco obtiene como entrada el registro **rs2** del **%ir**

- C: Registro del bus C
- CMUX: Selecciona la entrada del decodificador C:
    - CMUX=0, el deco obtiene como entrada el registro C
    - CMUX=1, el deco obtiene como entrada el registro **rd** del **%ir**

- RD y WR: Indican si la operación es de lectura o escritura:
    - RD=0, WR=1: Escritura
    - RD=1, WR=0: Lectura
    - RD=0, WR=0: No se lleva acabo operación de lectura ni escritura
    - RD=1, WR=1: No permitido que valgan ambos 1

- ALU: Determina cual va a ser la operación que se ejecuta
- COND:

    - 000: NEXT ADDR (No hace ningún salto, y se lee el next del mux de direcciones)
    - 001,010…,101: JUMP ADDR condicionales
    - 110: JUMP ADDR
    - 111: DECODE (Decodifica la instrucción en un solo paso, a través del uso de los campos op, op2 y op3 de la instrucción)
- JUMP ADDR: Dirección del salto, en la memoria de control (Como hay 2^{11} micropalabras en la memoria de control, se usan 11 bits)

Microprograma

- Es el encargado de interconectar el hardware con el software
- Se lo suele llamar firmware y su propósito es interpretar el conjunto de instrucciones visibles al usuario
- La memoria de control posee 2048 palabras, y cada sentencia del microprograma está antecedida por el número decimal que indica su dirección en la memoria
- La primer tarea es cargar la instrucción apuntada por el contador de programa **PC** (que fue inicializado con la dirección de comienzo del microprograma) al registro de instrucciones **IR**
- Luego, se sigue por siguiente sentencia (en sentido creciente), salvo que se detecte una operación de **GOTO** o **DECODE**
- Cada línea del microprograma en **ARC** corresponde a una palabra en la memoria de control, esto hace que se pueda ensamblar, línea a línea, en único paso

Microcódigo asociado al ciclo de fetch

0: R[ir] <- AND (R[pc],R[pc]); READ; /Cargo el registro **%ir**, con lo que apunta el **%pc**
1: DECODE /Busco en la **ROM**, el conjunto de microcódigo que ejecuta la operación deseada
/Acá irán las operaciones, por ejemplo el código correspondiente al addcc que veremos más adelante

/Luego de ejecutar la instrucción, en general hace un " **GOTO 2047** "

2047: R[pc] <- INCPC(R[pc]); GOTO 0; /Incremento en 4 el program counter, y vuelvo a iniciar el ciclo

Ejemplos de microcódigo para interpretar instrucciones

Nota a tener en cuenta sobre microcódigo:

- R[x] : Representa " el registro x", entonces cambiando la x por algún número, o nombre de los registros del trayecto de datos obtendremos, por ej: R[1] = %r1, R[ir] = %ir, etc

- Hasta no poner otra posicion, osea un numero y dos puntos (por ej 1600:), sigue siendo todo parte de una misma micropalabra. Para entender esto mejor, podemos decir que el microcódigo que escribimos para realizar el “addcc” conlleva 5 micropalabras
- No importa el orden donde escribamos el “ if “, siempre se ejecuta toda la línea, salvo lo que va luego del “ then ”

Addcc (Suma y toca flags)

1600: If R[IR[13]] then GOTO 1602; /Chequea el campo 13 del registro **%ir** para ver si el segundo operando está en modo inmediato, de ser así salta a procesar la constante que le vamos a “addear” al registro
1601: R[rd] = ADDCC(R[rs1],R[rs2]); /Si la condición del **if** fue **falsa**, significa que es un GOTO 2047; **ADDCC** entre dos registros, por lo tanto realiza el **ADDCC** sobre los registros dados y lo guarda en el registro de destino
1602: R[temp0] = SEXT13(R[ir]); / **SEXT13** realiza la extensión de signo de los 13 bits menos significativos de la palabra contenida en el registro **%ir**, por lo que obtiene el campo **simm13** almacenado en el **%ir**, y lo almacena en el registro temporal
1603: R[rd] = ADDCC(R[rs1],R[temp0]); /Realiza ADDCC con registro y la constante a la GOTO 2047; cual extendimos el signo, para que ocupe los 32 bits y podamos trabajarla
2047: R[pc] = INCPC(R[pc]); GOTO 0; /Finalmente aumentamos en 4 el valor del %pc y volvemos a comenzar el ciclo

ld (Load)

1792: R[temp0] = ADD(R[rs1],R[rs2]); /Calcula dirección a leer con dos registros puntero
If R[IR[13]] Then GOTO 1794; /Si es con (registro + constante) salta
1793: R[rd] = AND(R[temp0],R[temp0]); /Coloca la dirección en el bus A
READ; GOTO 2047; /Lee el dato al registro rd y termina
1794: R[temp0] = SEXT13(R[ir]); /Obtiene el campo simm13 para la dirección a leer
1795: R[temp0] = ADD(R[rs1],R[temp0]); /Calcula la dirección y salta
GOTO 1793;
2047: R[pc] = INCPC(R[pc]); GOTO 0; /Finalmente aumentamos en 4 el valor del %pc y volvemos a comenzar el ciclo

Traps e interrupciones

- **Trap**: Procedimiento automático de llamada generado por el hardware como consecuencia de una condición excepcional que se produce durante la ejecución de un programa, como por ejemplo una instrucción ilegal
- **Interrupciones**: Situaciones que se producen luego de algún evento circuital considerado como excepción, como por ejemplo el accionamiento de una tecla del teclado, una falla de alimentación, etc

Control cableado

Es una alternativa a la unidad de control microprogramada, se implementa a través del uso de flip flops y compuertas lógicas.
Los pasos que seguían los microprogramas, se reemplazan por estados de una máquina de estados finitos.
Para administrar la complejidad del control cableado, se utiliza un lenguaje llamado HDL (Hardware Description Language), el cual está formado por tres secciones:

- Preámbulo
- Sentencias enumeradas
- Epílogo

La sección de control maneja la forma de realizar la transiciones entre una secuencia y otra, además está relacionada con la generación de las salidas y el cambio de los valores de cualquier elemento de memoria.
Está implementada con 4 flip flops, uno para cada sentencia. Luego están interconectados con una solución de circuitos combinatorios sencillos.
Por último, la sección de datos, es igual a la de la forma microprogramada.

Preguntas de examen

- Durante la ejecución de un microprograma las sucesivas microinstrucciones van siendo cargadas en el MIR. Explique la lógica digital que define la dirección de la microinstrucción a ser cargada
    - **MIR** (Microprogram Instruction Register): Con un largo de 41, esta “seccionado” en 11 campos distintos, los cuales se relacionan con el diagrama de la microarquitectura de ARC
    - Se detalla más arriba, cada campo del **MIR** a que va conectado, y cómo influye en la ejecución del microprograma

- Explique qué entiende por nanoprogramación. Que problema se busca solucionar a través de ella? Indicar ventajas y desventajas
    - Cuando en un microcódigo tenemos gran **reiteración** de las mismas palabras, puede **ahorrarse gran espacio** de memoria utilizando nanoprogramacion, y esto es, armando un **nanoprograma** con las palabras del microcódigo que nos interesa sustituir, y desde el microcódigo solo tendremos un índice a las palabras almacenadas en nuestro nanoalmacenamiento.
    - La máquina funcionará más lentamente, pero ocupando un espacio menor

- El trayecto de datos incluye tanto multiplexores como decodificadores, explique las características y función desempeñada por cada uno de ellos

    - **Multiplexores (MUX)**:
        - Controla si el registro que se almacena viene de la ALU o del programa principal
        - Se le conectan 2^n entradas y tiene solo una salida
        - Para controlar el valor de que entrada va a salir, se utilizan entradas de control (Con **n** entradas de control, manejo **2^n** entradas de datos)
    - **Demultiplexor (DEMUX)**:
        - Cumplen exactamente la función inversa que los multiplexores, con 1 sola entrada, se elige a cuál de las **2^n** salidas mandar el valor de dicha entrada
    - **Decodificadores**
        - Controlan cual registro se activa para escribir en el bus, solo 1 registro a la vez
        - Se utilizan para activar otros dispositivos, tienen **n** entradas, las cuales controlan cual de los **2^n** aparatos conectados en las salidas se activa

- Explique qué entiende por ciclo de búsqueda-ejecución (ciclo de fetch).
  Explique el modo por el cual este es implementado a través del firmware de una microarquitectura ARC. Detalle el microcódigo asociado.
    - El ciclo de fetch está descripto en una sección anterior
    - El firmware es codificado a través del uso del lenguaje microensamblador, y no es accesible al usuario
    - El microcódigo asociado al ciclo de fetch, asociado al firmware es el que está en la ROM

- Indicar el principio de funcionamiento de un desplazador rápido del tipo “barrel shifter” y compararlo con la alternativa de implementar la misma función con un registro de desplazamiento
    - Barrel shifter:
        - Es un circuito digital que permite desplazar, una cantidad de bits deseada, a una palabra en un solo ciclo de clock
        - Se implementa como una secuencia de multiplexores
        - La cantidad de multiplexores para una palabra de **n** bits es **$n*\log_2(n)$**
    - La diferencia entre el barrel shifter y un registro de desplazamiento (shift register), es que el barrel puede mover bits, de cualquier posición a cualquier otra en un solo clock, mientras que con un registro de desplazamiento solo se puede mover 1 bit a derecha o a izquierda por cada ciclo de clock.
      La desventaja, es que el barrel conlleva muchas más compuertas que el shift register

- Muchas rutinas de microcódigo se encuentran duplicadas en la memoria de control (Control Store o CS). Justifíquelo. Indique de qué manera esta repetición de código puede ser evitada

    - Las rutinas de microcódigo más comúnmente duplicadas, corresponden a instrucciones de salto que siempre son iguales, y también a códigos correspondientes a instrucciones ilegales.
    - La forma de evitar la repetición de código, es utilizando nanoprogramacion

- Explique de qué manera la microarquitectura ARC microprogramada lee la instrucción de Assembly guardada en memoria principal, determina el microcódigo que la realiza y una vez terminada, avanza a la siguiente instrucción en memoria principal
    - Mediante el microcódigo asociado al ciclo de fetch

- Explique de qué manera la microarquitectura ARC microprogramada controla el flujo del microprograma
    - Para responder esta pregunta, hay que realizar el dibujo completo de la microarquitectura ARC (Figura 6.10 del Murdocca)

Capítulo 7

Memorias

Jerarquías de la memoria

Tenemos distintos tipos de memorias, y las podemos clasificar en relación a su velocidad de acceso y el precio de la misma. La relacion sera, a mayor velocidad, mayor precio.
Los tipos de memoria, ordenados de más cara y rápida a mas barata y lenta:

- Registros
- Cache
- Memoria principal
- Memoria secundaria
- Almacenamiento off-line (La más barata y lenta)

RAM (Random Access Memory)

El tiempo y procedimiento para el acceso es independiente de la dirección accedida.

- **RAM dinámica**
    - DRAM fue patentada en 1968
    - El 1 o 0 lógico, es un capacitor cargado o descargado
    - Significativamente **más lenta que RAM estática**
    - Significativamente **más barata que RAM estática**:
        - (1 transistor + 1 capacitor) VS (6 transistores)
    - Permite lograr alta densidad de elementos de memoria
    - Se usa para grandes volúmenes de memoria

- Criterios para reducir el tiempo de acceso
    - Bancos entrelazados
        - Accede a un banco, mientras en el otro se refresca la información
        - Enmascara el tiempo de refresco
        - Mejora rendimiento si posic. sucesivas están en bancos diferentes
    - Mayor cantidad de bits leídos en paralelo
        - Ancho del bus
    - Mayor velocidad del clock (Utilizando memoria sincrónica SDRAM)
        - Mayor consumo de potencia
        - Proclive a errores de almacenamiento de bits
    - Con igual velocidad de clock, leer en flanco ascendente descendente (DDR)

Caché (90% del tiempo de ejecución corresponde al 10% del código)

Evita el cuello de botella producido por la diferencia entre la velocidad del CPU y la velocidad de memoria principal.
La caché es mas rapida que la memoria principal, ya que está construida con electronica mas rapida (SRAM), ocupa más espacio, disipa más potencia.
Posee un árbol de codificación pequeño y como su cercanía al CPU es fisica y logica, no necesita un bus compartido para comunicarse.

- **Principio de localidad**
  - Localidad **temporal**: Si accedo a una dirección, en poco tiempo volveré a accederla
    - Sucede en Iteraciones, procedimientos recursivos
  - Localidad **espacial**: Si accedo a una dirección, las direcciones cercanas tienen mayor probabilidad de ser accedidas
    - Sucede cuando se almacenan los datos en posiciones contiguas
- **Características**
  - Memoria muy rápida
  - Poca capacidad
  - “Cercana” al CPU
- **Funcionamiento**
  - Memoria dividida en bloques
  - Al acceder a un dato en memoria principal bajo el bloque completo al cache
  - En el próximo acceso, verifico si la posición buscada está en la caché y sino cargo otro bloque
- **Estructuras del Caché**
  - Caché especializado
    - Caché de datos
    - Cache de instrucciones
  - Caché multinivel
    - Caché más grandes, son más lentos
    - Cache grandes, mayor índice de aciertos, entonces varios niveles

Preguntas de examen

- Analizar y comentar las siguientes afirmaciones:
  - La existencia del caché es transparente al programador

        - El programador no toma en cuenta la existencia de la caché al hacer un programa, lo que hace que se pueda abstraer del funcionamiento del mismo
    - El aprovechamiento del cache es óptimo si el programa maneja muchas variables en memoria
        - La caché se base en principios de localidad temporal y espacial. Esto es, una variable que se usó recientemente es altamente probable que vuelva a ser usada. Y si se usa una variable, es muy probable que se use una variable que se encuentra en una posición cercana de memoria.
          Pr tal motivo, la caché explota esto t**rayendo la variable que le pedís y las que estén en su vecindad**. Si un programa accede siempre a variables cercanas, es altamente probable que esté en caché, y por lo tanto tenga un menor tiempo de acceso, dando así un mejor rendimiento.
    - Un procesador con caché podría ser más lento que otro sin caché en determinadas condiciones
        - Como la caché tiene un tamaño limitado, es probable que termine perjudicando el rendimiento. Las caches tienen distintas políticas para hacer los reemplazos de los bloques. Si un programa accede a una variable que va a caer en el bloque X y después hace un salto y accede a otra variable que también mapea al bloque X, éste va a ser reemplazado. Entonces, si constantemente está accediendo a estas dos variables que mapean al mismo bloque dentro de la caché, la performance va a ser peor que si no existiera cache. Como la caché introduce el costo de búsqueda y actualización de bloques, ese tiempo se va a ver sumado a los de acceso de memoria principal.

- Indique la función de las señales RAS Y CAS en un módulo de memoria DIMM
    - *Row Address (RAS) / Column Address (CAS);*
        - *Son los tiempos que hay que esperar una vez que se manda el número de fila (Row) hasta que se pueda mandar el número de columna (Column), RAS. Después, el tiempo que hay que esperar para que dato esté disponible en la salida del módulo de RAM, CAS.*