Parcial #2

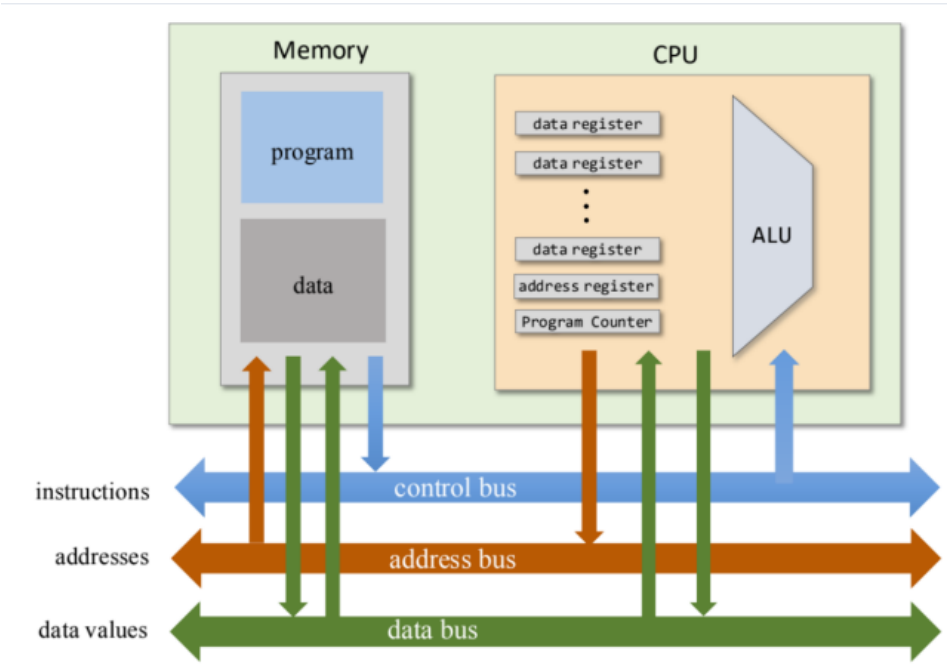
Equipo:
Yasir Enrique Blandon Varela
Juan Pablo Rúa Cartagena

Docente Académico:
Alberto Mauricio Arias Correa

UNIVERSIDAD EAFIT
Departamento de Informática y Sistemas
Organización de computadores

Medellín - Antioquia
4 de abril de 2025

- **Esquema:**



La CPU ejecuta instrucciones en cuatro etapas secuenciales: fetch (obtiene la instrucción desde memoria usando el Program Counter), decodificación (interpreta la operación y operandos), ejecución (realiza la operación con la ALU o accesos a memoria) y write-back (almacena el resultado en registros o memoria). Durante el fetch, el PC envía la dirección por el bus de direcciones y recibe la instrucción por el bus de datos; en la decodificación, se identifica el tipo de instrucción (como suma o carga); en la ejecución, la ALU opera o se accede a RAM; finalmente, en write-back, el resultado se guarda en su destino. Este ciclo se repite para cada instrucción, coordinando buses, registros y unidades funcionales.

Una forma de probar el funcionamiento es con un ejemplo especifico:

```
@10          // RAM[10] = 2
D=M          // RAM[11] = 3
@11
D=D+M
@12
M=D
```

Paso 1: Ejecución de @10

- **Fetch (Obtención de la instrucción):**

El Program Counter (PC) apunta a la dirección de memoria donde está @10.

La CPU envía esta dirección por el bus de direcciones.

La memoria devuelve el código máquina de @10 por el bus de datos.

Decodificación:

La CPU reconoce que es una instrucción @ (carga de dirección).

El valor 10 se almacena en el registro A.

Actualización del PC:

El PC se incrementa para apuntar a la siguiente instrucción (D=M).

Estado actual:

Registro, Valor

A, 10

D, ?

PC, → D=M

Paso 2: Ejecución de D=M

- **Fetch:**

El PC envía la dirección de D=M por el bus de direcciones.

La memoria devuelve el código de D=M.

- **Decodificación:**

La CPU identifica que debe leer RAM[A] (donde A = 10) y guardarlo en D.

- **Ejecución:**

La CPU coloca A=10 en el bus de direcciones y activa MEMREAD.

La RAM devuelve el valor 2 (porque RAM[10] = 2).

La CPU guarda 2 en el registro D.

Estado actual:

Registro, Valor

A, 10

D, 2

PC, → @11

Paso 3: Ejecución de @11

- **Fetch:**

El PC envía la dirección de @11.

La memoria devuelve el código de @11.

- **Decodificación:**

La CPU guarda 11 en el registro A.

Estado actual:

Registro, Valor

A, 11

D, 2

PC, → D=D+M

Paso 4: Ejecución de D=D+M

- **Fetch:**

El PC envía la dirección de D=D+M.

La memoria devuelve el código de D=D+M.

- **Decodificación:**

La CPU reconoce que es una operación de suma.

Los operandos son:

D (actualmente 2)

RAM[A] (donde A = 11).

- **Ejecución:**

La CPU coloca A=11 en el bus de direcciones y activa MEMREAD.

La RAM devuelve 3 (porque RAM[11] = 3).

La ALU suma D (2) + RAM[11] (3) = 5.

El resultado (5) se guarda en D.

Estado actual:

Registro, Valor

A, 11

D, 5

PC, → @12

Paso 5: Ejecución de @12

Fetch:

El PC envía la dirección de @12.

La memoria devuelve el código de @12.

- **Decodificación:**

La CPU guarda 12 en el registro A.

Estado actual:

Registro, Valor

A, 12

D, 5

PC, → M=D

Paso 6: Ejecución de M=D

- **Fetch:**

El PC envía la dirección de M=D.

La memoria devuelve el código de M=D.

- **Decodificación:**

La CPU reconoce que debe escribir D en RAM[A] (donde A = 12).

- **Ejecución:**

La CPU coloca A=12 en el bus de direcciones.

Coloca D=5 en el bus de datos.

Activa MEMWRITE para guardar el valor en RAM.

Estado final:

Registro, Valor

A, 12

D, 5

RAM[12], 5

- **Solución Taller:**

1. Explique cómo se puede implementar cualquier función booleana utilizando solamente compuertas NAND. Justifique con base en la lógica booleana.

   Cualquier función booleana se puede implementar con compuertas nand porque es una compuerta universal. Esto significa que podemos construir todas las demás operaciones básicas (not, and, or) usando solo nand, y dado que cualquier función booleana se expresa con estas operaciones, podemos implementarlas solo con nand.

✓ **Implementación de operaciones básicas con nand:**

Para demostrar que nand es suficiente, primero mostramos cómo obtener not, and y or usando solo nand:

→ **Not usando nand:**

~A = A NAND A

→ **And usando nand:**

A·B = (A NAND B) NAND (A NAND B)

→ **Or usando nand:**

A+B = (~A NAND ~B) = ((A NAND A) NAND (B NAND B))

Dado que con nand podemos construir las compuertas and, or y not, y estas tres operaciones forman una base funcional completa, es decir, cualquier expresión booleana se puede construir combinándolas, podemos afirmar que cualquier función booleana puede implementarse exclusivamente con compuertas nand.

2. ¿Cuál es la función principal de la ALU en el contexto de la arquitectura HACK? Describa tres operaciones diferentes que puede realizar y qué señales de control deben activarse para lograrlas.

La alu en la arquitectura hack es la responsable de realizar las operaciones aritméticas y lógicas sobre dos operandos de 16 bits, usualmente denominados x y y. Su función principal es realizar cálculos y evaluar condiciones para la ejecución de instrucciones en la CPU. Por otro lado, es fundamental para la ejecución de instrucciones, permitiendo realizar operaciones matemáticas y lógicas mediante señales de control. Al manipular estas señales, se pueden definir distintas operaciones, como suma, and, o negación.

✓ **Ejemplo de operaciones con señales de control:**

→ **Suma de x + y**

zx = 0
nx = 0
zy = 0
ny = 0
f = 1
no = 0
out = x + y

→ **Operación lógica and (x AND y)**

zx = 0
nx = 0
zy = 0
ny = 0
f = 0
no = 0
out = x AND y

→ **Negación de x (~x)**

zx = 0
nx = 1
zy = 1
ny = 0
f = 0
no = 0
out = ~x

3. Compare y contraste el comportamiento de los circuitos combinacionales frente a los circuitos secuenciales, especialmente en el contexto del manejo del tiempo y la necesidad del reloj (clock).

Característica	Circuitos Combinacionales	Circuitos Secuenciales
Definición	La salida depende únicamente de las entradas actuales.	La salida depende de las entradas actuales **y** del estado anterior.
Memoria	No tienen memoria, no retienen información.	Usan memoria (flip-flops, registros) para almacenar información.
Dependencia del tiempo	La salida cambia instantáneamente cuando cambian las entradas.	La salida cambia en función del tiempo y de señales de control (como el reloj).
Uso del reloj (clock)	No necesitan reloj porque los cambios son inmediatos.	Normalmente usan reloj para sincronizar cambios de estado.
Ejemplo de circuitos	Sumadores, multiplexores, comparadores, decodificadores.	Contadores, registros, memorias, procesadores.

Los circuitos combinacionales son rápidos y no dependen del tiempo, pero no pueden almacenar datos, mientras que los circuitos secuenciales permiten el almacenamiento de información y la sincronización de eventos, lo que los hace esenciales en sistemas complejos como CPU y memorias. El reloj es clave en los circuitos secuenciales, ya que regula cuándo ocurren los cambios de estado y garantiza una ejecución ordenada.

4. ¿Cómo se implementa el almacenamiento de datos en la memoria RAM en HACK? Mencione los componentes principales implicados y el rol del registro A.

   En la arquitectura HACK, la memoria RAM se implementa utilizando registros y un circuito de direccionamiento que permite leer o escribir en una celda específica.

   Componentes principales involucrados:

   → **Registros**

   Cada celda de la RAM es un registro de 16 bits, capaz de almacenar un valor binario de 16 bits. En una RAM de tamaño RAM8, por ejemplo, hay 8 registros de 16 bits.

   → **Decodificador de dirección**

   Se encarga de seleccionar cuál registro debe activarse para leer o escribir, según la dirección de memoria. En una RAM8, la dirección es de 3 bits (porque 2^3 = 8), lo que permite seleccionar uno de los 8 registros.

   → **Multiplexor**

Cuando se lee un valor de la RAM, el multiplexor selecciona el contenido del registro correspondiente y lo envía a la salida.

→ **Señal de control load**

Controla si un nuevo valor debe almacenarse en la dirección seleccionada. Si load = 1, el valor de entrada se guarda en el registro seleccionado.

Rol del registro A

El registro A es fundamental para direccionar la RAM: Su valor determina qué celda de memoria se leerá o escribirá. Si la CPU ejecuta una instrucción de carga como @5, el valor 5 se guarda en el registro A y se usará como dirección en la RAM. Luego, si se ejecuta una operación como D=M, la CPU leerá desde la celda RAM[5].

5. En el lenguaje ensamblador de HACK, ¿qué diferencia existe entre las instrucciones tipo A y tipo C? Dé un ejemplo de cada una y explique su función.

En el ensamblador de HACK, las instrucciones se dividen en dos tipos:

Tipo de instrucción	Propósito	Formato	Ejemplo
Instrucción tipo A	Carga una dirección en el registro A.	@valor	@5 (Carga la dirección 5 en A)
Instrucción tipo C	Realiza operaciones aritméticas/lógicas o asignaciones.	dest = comp ; jump	D = M + 1 (Guarda en D el valor de M + 1)

- Ejemplo de cada tipo:

→ **Instrucción tipo A** (@valor)

@10

Función:

- Guarda el valor 10 en el registro A.
- Si luego se usa M, se referirá a RAM[10].

→ **Instrucción tipo C** (dest = comp ; jump)

D = M + 1

Función:

- Toma el valor almacenado en la celda de RAM indicada por A (M = RAM[A]).
- Le suma 1.
- Guarda el resultado en el registro D.

6. Analice el siguiente fragmento de código en lenguaje ensamblador HACK y explique línea por línea lo que realiza. ¿Qué haría este código si se pone un valor 5 en R0?

```
@i          // Carga la dirección de la variable i en el registro A
M=1         // RAM[i] = 1 (inicializa i = 1)

@sum        // Carga la dirección de la variable sum en A
M=0         // RAM[sum] = 0 (inicializa sum = 0)
(LOOP)      // Etiqueta para el inicio del bucle
@i          // A = dirección de i
D=M         // D = valor de i (D = RAM[i])

@R0         // A = 0 (dirección de R0)
D=D-M       // D = i - R0 (resta el valor de R0 para verificar condición)

@STOP       // A = dirección de la etiqueta STOP
D;JGT       // Si D > 0 (i > R0), salta a STOP (termina el bucle)
@sum        // A = dirección de sum
D=M         // D = valor actual de sum (D = RAM[sum])

@i          // A = dirección de i
D=D+M       // D = sum + i (acumula la suma)

@sum        // A = dirección de sum
M=D         // RAM[sum] = D (actualiza sum con el nuevo valor)

@i          // A = dirección de i
M=M+1       // RAM[i] += 1 (incrementa el contador i)

@LOOP       // A = dirección de LOOP
0;JMP       // Salto incondicional a LOOP (repite el bucle)
(STOP)      // Etiqueta de terminación
```

¿Qué hace este código con R0 = 5?

Este código calcula la suma de los primeros R0 números naturales.

→ Ejemplo con R0 = 5:

1+2+3+4+5 = 15

Después de ejecutarse, la variable sum tendrá 15.

7. ¿Qué papel juega el clock en el diseño de memorias y registros? ¿Por qué es importante definir una longitud de ciclo adecuada?

   El clock es fundamental en el diseño de memorias y registros porque controla cuándo se almacenan y actualizan los datos, además, marca el paso de los ciclos en los cuales se realizan las operaciones y las instrucciones dadas.

   El ciclo de reloj es el tiempo entre dos pulsos de clock consecutivos. Si el ciclo es demasiado corto, los circuitos no tienen tiempo suficiente para procesar datos lo cual puede provocar inestabilidad y errores, pero si el ciclo es demasiado largo, el sistema funciona correctamente, pero más lento de lo necesario y se desperdicia rendimiento.

8. En el contexto de un ensamblador para la arquitectura HACK, explique brevemente cómo se traducen las instrucciones simbólicas a binario. ¿Qué estructuras de datos son necesarias para manejar los símbolos y las etiquetas en este proceso?

   En HACK, hay dos tipos de instrucciones, cada uno de los registros cuenta con 16 bits para el caso del tipo A el ultimo bit [15] siempre en 0 quiere decir que es una instrucción tipo a y para el caso del tipo C 111 + a + cccccc + ddd + jjj

   - **Instrucciones tipo A (@valor)**

   Se usan para cargar una dirección o valor en el registro A.

   Su formato binario es:

   - **0vvvvvvvvvvvvvv**

   Donde vvvvvvvvvvvvvv es el número en binario de 15 bits.

   - **Instrucciones tipo C (dest = comp ; jump)**

   Se usan para cálculos y operaciones lógicas.

   Su formato binario es:

   - **111accccccdddjjj**

   a → Determina si se usa A o M.

   ccccc → Código de la operación.
   ddd → Código del destino.
   jjj → Código del salto.

9. El siguiente bloque de instrucciones HACK y determine: a) Qué valores específicos toman los registros D, A y la RAM en la dirección 100. b) Qué operaciones realiza la ALU en cada instrucción tipo C.

Paso 1: cargar 100 en A

@100
A = 100

Paso 2: guardar A en D

D = A
ALU realiza: out = A
D = 100

Paso 3: guardar D en RAM[17]

@17
M = D
A = 17
RAM[17] = 100

Paso 4: cargar RAM[17] en D

@17
D = M
A = 17
D = RAM[17] = 100
ALU realiza: out = M

Paso 5: sumar D + A

@5
D = D + A
A = 5
D = 100 + 5 = 105
ALU realiza: out = D + A

Paso 6: Guardar D en RAM[100]

@100
M = D
A = 100
RAM[100] = D = 105

Registro / Memoria	Valor Final
D	105
A	100
RAM[100]	105
RAM[17]	100

Instrucción	**Operación ALU**	**Resultado**
D = A	out = A	D = 100
M = D	out = D	RAM[17] = 100
D = M	out = M	D = 100
D = D + A	out = D + A	D = 105
M = D	out = D	RAM[100] = 105

10. En el lenguaje ensamblador de HACK, ¿por qué se requiere una instrucción tipo A antes de acceder a una posición de memoria usando M? ¿Qué implicaciones tiene esta estructura para la ejecución de instrucciones condicionales y cómo se relaciona con la arquitectura de la ALU?

    Debido a que M que es la variable que toma el valor de la dirección de A depende de esta no es posible acceder directamente ya que se necesita saber por medio de A a que dirección se trata de llegar, ya que la alu trata de operar solo con D y A.

    Esto ayuda en saltos condicionales a que el código sea más explícito en el sentido de saber qué hace cada instrucción, pero trae como desventaja que no se puede combinar instrucciones y finalmente tiene que ver todo con la alu ya que esta no puede acceder directamente a la memoria RAM y solo recibe datos por medio de A y D para separar los datos de instrucciones. La alu en HACK solo opera con los registros D, A, y M (que es RAM[A]). Como M depende de A, es obligatorio establecer A antes de operar con M.

11. Dado un conjunto de instrucciones HACK que manipulan bucles y condiciones (@xx,D;JGT, 0;JMP, etc.), explique cómo se implementa un ciclo while(x > 0) utilizando registros D y A. Discuta qué operaciones realiza la ALU implícitamente y cómo se controlan mediante los campos: comp, dest y jump.

    Si suponemos que x está almacenado en RAM[0], y queremos ejecutar un ciclo mientras x > 0, decrementándolo en cada iteración.

```
(LOOP)      // Etiqueta de inicio del bucle
 @0         // Cargar dirección de RAM[0] en A
 D = M      // D = RAM[0] (Carga x en D)
 @END       // Dirección donde termina el bucle
 D;JLE      // Si x <= 0, salir del bucle
 @0         // Cargar dirección de RAM[0] en A
 M = M - 1  // RAM[0] = RAM[0] - 1 (Decrementar x)
 @LOOP      // Volver a la etiqueta LOOP
 0;JMP      // Salto incondicional al inicio del bucle
(END)       // Punto de salida del bucle
```

12. Diseñe una rutina en lenguaje ensamblador HACK que: • Lea dos valores almacenados en RAM[10] y RAM[11] • Calcule el máximo de los dos usando instrucciones tipo C • y almacene el resultado en RAM[12]. Explique detalladamente cómo las operaciones de la ALU son utilizadas y controladas desde las instrucciones ensamblador

```
@10
 D=M
 @11
 D=D-M              // D = RAM[10] - RAM[11]
 @RAM10_MAYOR
 D;JGT              // Salta si RAM[10] > RAM[11]
 @11
 D=M                // D = RAM[11] (máximo)
 @12
 M=D
 @FIN
 0;JMP
(RAM10_MAYOR)
 @10
 D=M                // D = RAM[10] (máximo)
 @12
 M=D
```

13. Analice la siguiente instrucción tipo C: D=D-M;JLE a) ¿Qué entradas están siendo usadas por la ALU? b) ¿Cuál es el resultado que la ALU genera si D = 10 y M = 20? c) ¿Qué efecto tiene el campo jump si el resultado es negativo?

- **Entradas usadas por la ALU:**

  D (contenido del registro D).

  M (contenido de la memoria en RAM[A]).

La ALU realiza la operación D - M.

- **Resultado de la ALU si D = 10 y M = 20:**

  D − M = 10 − 20 = −10
  La ALU genera -10 como salida.

- **Efecto del campo jump si el resultado es negativo:**

  JLE significa que el salto ocurre si el resultado de la ALU es negativo o cero. Como -10 < 0, la condición se cumple y la ejecución salta a la dirección indicada en A en la siguiente instrucción.

14. Explique el papel del Data Flip-Flop (DFF) en la implementación de memorias secuenciales. ¿Por qué no es posible construir una celda de memoria RAM confiable usando únicamente compuertas lógicas combinacionales?

    El data flip-flop es muy importante ya que es el elemento básico de almacenamiento en memorias secuenciales. Su función es guardar un bit de información hasta que llegue una señal de reloj que permita actualizar su estado.

    No es posible construir una celda de memoria RAM confiable solo con lógica combinacional porque los circuitos combinacionales no tienen estado: su salida depende solo de la entrada actual y no pueden recordar valores pasados. En cambio, los circuitos secuenciales con los data flip-flops pueden almacenar datos a lo largo del tiempo, permitiendo que la RAM retenga información entre ciclos de reloj.

15. El diseño de memoria RAM requiere una sincronización precisa con el reloj. Suponga que el ciclo de reloj es más corto que la suma de los retardos de propagación de los componentes internos (DFF, multiplexores, buses). a) ¿Qué consecuencias tendría esto sobre la integridad de los datos almacenados o leídos? b) ¿Cómo se mitiga este problema desde el diseño lógico?

    Si el ciclo de reloj es más corto que la suma de los retardos de propagación de los componentes internos, la memoria RAM no tendrá suficiente tiempo para estabilizar los datos antes de la siguiente transición del reloj.

    Esto puede provocar errores en la lectura y escritura, ya que los valores almacenados podrían ser incorrectos o inconsistentes debido a cambios prematuros en los registros y buses. Para mitigar este problema desde el diseño lógico, se debe elegir un período de reloj adecuado que sea mayor que la suma de los retardos de propagación de todos los componentes involucrados.

Además, se pueden implementar técnicas como la sincronización de señales, el uso de registros intermedios para retener datos estables y la optimización del diseño físico de los circuitos para reducir la latencia de propagación.

16. Analice el siguiente escenario: un registro de 16 bits debe almacenar un valor externo solo cuando la señal load está activa. Explique, paso a paso, cómo el comportamiento del registro se ve afectado por: a) La señal de clock (tick-tock), b) El valor de load, c) El retardo de propagación interno de los DFFs

    El comportamiento del registro de 16 bits está sincronizado con la señal de reloj, lo que significa que solo actualiza su valor en el flanco de subida del clock. Durante el resto del ciclo, el valor almacenado se mantiene sin cambios. La señal load controla cuándo el registro debe capturar un nuevo valor: si load = 1 en un flanco de reloj, el registro almacena el valor externo; si load = 0, el registro mantiene su valor anterior, sin importar la entrada.

    Además, el retardo de propagación interno de los flip-flops tipo D introduce un pequeño retraso entre el momento en que el reloj hace la transición y cuando el nuevo dato es efectivamente almacenado. Este retardo debe ser considerado en el diseño del sistema, ya que, si el reloj es demasiado rápido, los datos pueden no propagarse correctamente, causando errores en el almacenamiento.

17. a) Analice el código y explique su funcionamiento. b) Rediseñe el código para que el cursor no deje rastro a medida que se mueve por pantalla.

    Para evitar que el cursor deje rastro al moverse, primero debemos borrar la posición anterior antes de dibujar la nueva. Esto se logra almacenando la dirección previa del cursor y escribiendo 0 en esas posiciones antes de actualizar RAM[1] con la nueva posición.

    // Programa:

    @SCREEN

    D=A

    @1

    M=D // RAM[1] = 16384 (posición inicial del cursor)

    @1

    D=M

    @3

```
M=D // RAM[3] almacena la posición anterior
@0
D=M
@1
M=M+D // RAM[1] = RAM[1] + desplazamiento (nueva posición)
@0
D=A
@2
M=D // RAM[2] = 0
(ERASE_LOOP)
@2
D=M
@16
D=D-A
@DRAW_CURSOR
D;JGE // Si contador >= 16, pasar a dibujar el cursor
@3
D=M
@2
A=D+M // Dirección de pantalla anterior
M=0 // Borra el cursor anterior
@2
M=M+1
@ERASE_LOOP
0;JMP
(DRAW_CURSOR)
@0
```

```
D=A

@2

M=D // Resetear contador

(DRAW_LOOP)

@2

D=M

@16

D=D-A

@END

D;JGE // Si contador >= 16, finalizar

@1

D=M

@2

A=D+M // Dirección de pantalla actual

M=-1 // Dibujar nueva línea vertical

@2

M=M+1

@DRAW_LOOP

0;JMP

(END)

@END

0;JMP // Bucle infinito
```

Explicación de los cambios:

- Almacena la posición anterior en RAM[3] antes de actualizar la nueva.

- Borra la línea vertical en la posición anterior escribiendo 0 en las mismas 16 palabras de memoria.

- Luego, dibuja la nueva línea vertical en la posición actual sin dejar rastro.