Organización del computador 1

TEMARIO COMPLETO

Objetivo de la materia:

- *Dada una instrucción a la computadora, entender cómo la misma puede leer la instrucción y ejecutarla.*
- *Responder la pregunta ¿Qué es un procesador?*
- *Poder hacer nuestro propio procesador.*

Representación de la información

La organización de un computador depende del sistema de representación numérica adoptado.

Se trabaja con el sistema binario, de donde proviene el termino bit de "**b**inary **d**igit"

Bit: Unidad de medida. Es un valor decimal. Tiene dos estados (Binario) el cual se suele representar con un 1 y un 0. Si tengo n bits genero 2^n valores. Si tengo x valores necesito $\log_2(x)$ bits.

Cambios de base: ¿Para que quiero un cambio de base? -> para convertir lo que naturalmente conocemos en base 10 a base 2 para almacenar datos en la computadora.

Sistemas de Numeración:

Un sistema de numeración es un conjunto de símbolos y un conjunto de reglas de combinación de dichos símbolos que permiten representar los números enteros y/o fraccionarios.

Se habla de un sistema de numeración posicional cuando cada digito posee un valor diferente que depende de su posición relativa.

En un sistema de numeración posicional de base b, la representación del numero se define a partir de la regla

$$(a_n\text{---}a_3a_2a_1a_0)_b = a_n*b^n + \text{---} + a_3*b^3 + a_2*b^2 + a_1*b^1 + a_0*b^0$$

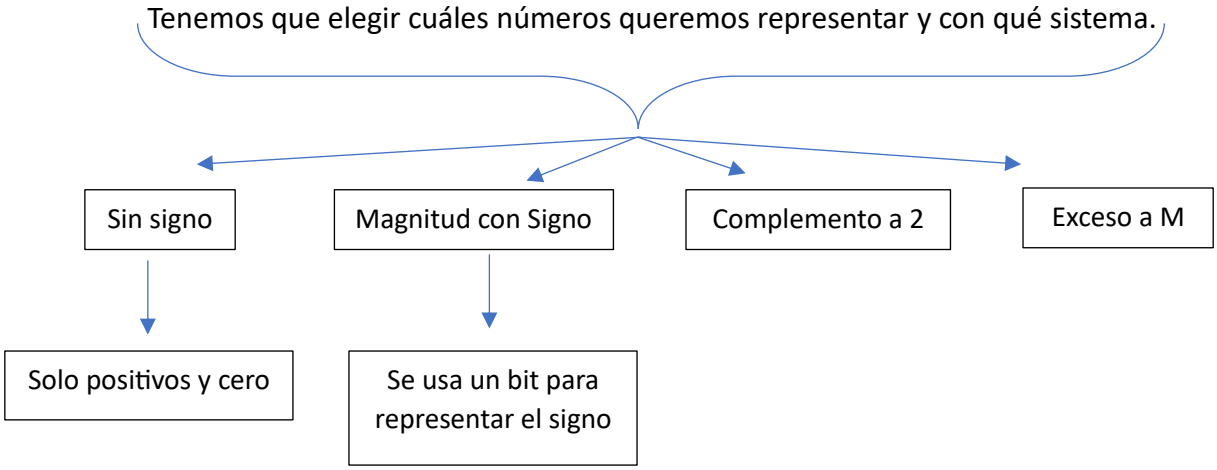
Por ejemplo:

$(25)_{10} = 2 * 10^1 + 5 * 10^0$

$(0101)_2 = 0*2^3+1*2^2+0*2^1+1*2^1 = (5)_{10}$

Precisión fija:

La computadora opera con sistemas de numeración de tamaño fijo. No se puede representar TODOS los números.



PD: Existen muchos otros tipos.

- **Magnitud con Signo:**
  - El bit mas a la izquierda es usado como indicador de signo. 1 implica que es negativo, 0 que es positivo.
  - Tiene un problema. Se puede representar dos veces el cero.
  - Sea una representación en formato de Magnitud con Signo que nos permite codificar un número entero en binario con 8 bits (un byte). Esto nos otorga 1 bit para el signo y 7 bits para la magnitud. Con 8 bits, podemos representar, en teoría al menos $2^8 = 256$ números. Pero utilizando un bit para indicar el signo, se podrán representar: el cero más 127 números positivos (bit de signo en 0), y 127 números negativos (bit de signo en 1); 255 números en total.
  - Es más complejo operar aritméticamente. Para realizar una suma, por ejemplo, primero hay que determinar si los dos números tienen el mismo signo, y en caso de que sea así, realizar la suma de la parte significativa, pero en caso contrario, restar el mayor del menor y asignar el signo del mayor.

La suma es igual que el sistema decimal, incluyendo el concepto de acarreo. En caso de llegar con un acarreo al bit mas a la izquierda, nos encontramos en una situación de **overflow.**

Ejemplo:

1				**1**	**1**	**1**	**1**	
0		1	0	0	1	1	1	1
0	**+**	1	1	0	0	0	1	1
0		0	1	1	0	0	1	0

$(79)_{10} = (1001111)_2$

$(99)_{10} = (1100011)_2$

79+99= 178. La suma dio como resultado $(0110010)_2 = 50$ lo cual es incorrecto. Esto ocurre porque usamos un sistema de 8 bits (8 dígitos). El 79 y el 99 son números que podemos representar en 8 bits usando base 2 pero, en este caso, la suma de ambos valores genera un valor que supera el máximo valor representable, estamos en el caso denominado “overflow”.

- Complemento a 2:
    - El complemento de un número se obtiene restando dicho número al número más grande que puede representarse con el tamaño de numeral con que contamos
    - La idea es usar los números “más altos” como números negativos.
    - ¿Cómo pasar un dato en bits a complemento a 2?

      Ejemplo: Quiero pasar el -3 a complemento a 2

      <u>Forma 1:</u>

      1) Calcular el valor en positivo -> 3 = 011
      2) Dar vuelta bit a bit -> 011 = 100
      3) Luego le sumo 1 -> 100 + 1 = 101
         Rta: 101

      <u>Forma 2:</u>

      1) Busco 3 en bits -> 3=011
      2) Luego busco 2^3 en bits -> $2^3 = 8 = 1000$
      3) Luego resto 1000 - 011 = 5 = 101
         Rta = 101

- ¿Cómo paso un valor en complemento a 2 a un valor en decimal?
  Ejemplo: Quiero pasar el 101 a decimal
  1) Le resto 1 -> 101 -1 = 100
  2) Doy vuelta bit a bit -> 100 = 011
  3) Calculo el valor decimal -> 011 = 3 (recuerdo que es en base 2)
  4) Como el valor en complemento a 2 tenía un 1 al principio ya se que en decimal es un numero negativo.
     Rta= -3

Circuitos combinatorios – Logita digital

- Jerarquía de maquina:

Nivel 6	Usuario	Programas ejecutables
Nivel 5	Lenguaje de alto nivel	C++, Java, Python, etc.
Nivel 4	Lenguaje ensamblador	Assebly code
Nivel 3	Software del sistema	Sistemas op, bibliots, etc.
Nivel 2	Lenguaje de maquina	Instruction ser architecture
Nivel 1	Unidad de control	Microcódigo hardware
Nivel 0	Lógica digital	Circuitos, memorias, etc.

  - Cada nivel funciona como una maquina abstracta que oculta la capa anterior.
  - Cada nivel es capaz de resolver determinado tipo de problema.
  - La capa inferior es utilizada como servicio.

- Organización:
  - Organización de un computador refiere al diseño especifico de sus componentes, y como estos se coordinan para llevar a cabo una tarea.
  - Un factor decisivo es como se estructuran los componentes, es como está representada la información.

- Lógica Digital:
  - Los circuitos operan con dos valores eléctricos.
    - Operadores booleanos: Los operadores booleanos están descriptos por un conjunto de axiomas. Pueden ser interpretados como tablas de verdad.

- Circuitos booleanos:

▸ Circuitos combinatorios ≡ Funciones Booleanas:

El resultado **solo** depende de sus entradas.

  - Los circuitos electrónicos implementan funciones booleanas y mientras mas simple la función, más pequeño será el circuito siendo a su vez mas barato, con un menor consumo y hasta mas rápido. El algebra de Boole nos permitirá la reducción de circuitos.
  - Los circuitos electrónicos están formados por compuertas que son dispositivos electrónicos que producen un resultado en función de su entrada.
  - Una compuerta está formada por uno o más transistores.

- Algebra de Boole:

- George Boole, desarrolló un sistema algebraico para formular proposiciones con símbolos.
- Su álgebra consiste en un método para resolver problemas de lógica que recurre solamente a los valores binarios:
    - verdadero / falso.
    - On / off.
    - 1 / 0.
- Tres operadores:
    - AND (y).
    - OR (o).
    - NOT (no)
- Las variables Booleanas solo pueden tomar los valores binarios: 1 o 0.
- Una variable Booleana representa un bit.
- Propiedades:

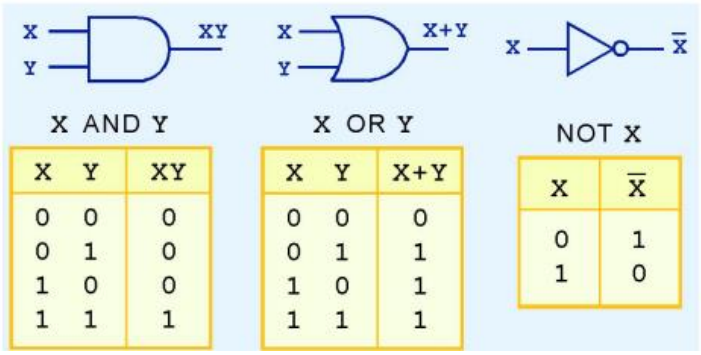
Identidad	$1.A = A$	$0 + A = A$
Nula	$0.A = 0$	$1 + A = 1$
Idempotencia	$A.A = A$	$A + A = A$
Inversa	$A.\overline{A} = 0$	$A + \overline{A} = 1$
Conmutativa	$A.B = B.A$	$A + B = B + A$
Asociativa	$(A.B).C = A.(B.C)$	$(A + B) + C = A + (B + C)$
Distributiva	$A + B.C = (A + B).(A + C)$	$A.(B + C) = A.B + A.C$
Absorción	$A.(A + B) = A$	$A + A.B = A$
de Morgan	$\overline{A.B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}.\overline{B}$

- <u>Compuertas lógicas:</u>

Definición: Una compuerta es un dispositivo electrónico que produce un resultado en base a un conjunto de valores de entrada.

- Producen una salida específica al (casi) instante que se le aplican valores de entrada.
- Implementan funciones booleanas.
- La aritmética y la lógica de la CPU se implementan con estos circuitos

Las compuertas más simples se corresponden exactamente con los operadores booleanos elementales que vimos anteriormente.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

X	$\overline{X}$
0	1
1	0

- Una compuerta muy útil: el OR exclusivo => **XOR**. La salida es 1 cuando los valores de entrada difieren.

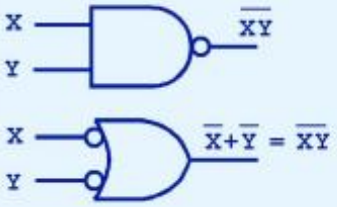
X XOR Y

X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

- **NAND y NOR** son dos compuertas lógicas combinadas. Con la identidad de Morgan se pueden implementar con AND u OR. Son más baratas y cualquier operación básica se puede representar usándolas cualquiera de ellas (¡sin usar la otra!).

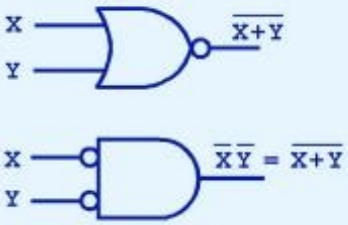
X NAND Y

X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0



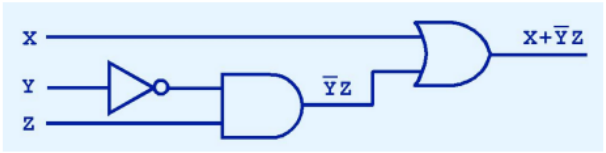
X NOR Y

X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



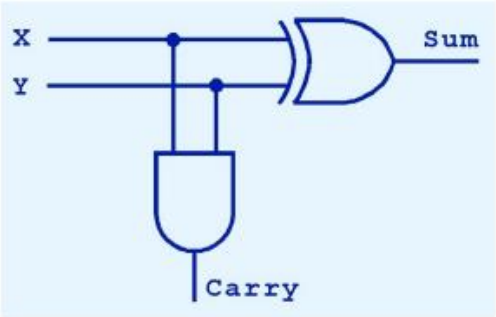
- Combinando compuertas se pueden implementar funciones booleanas.

- Este circuito implementa la siguiente función: $F(x, y, z) = x + \bar{y}z$

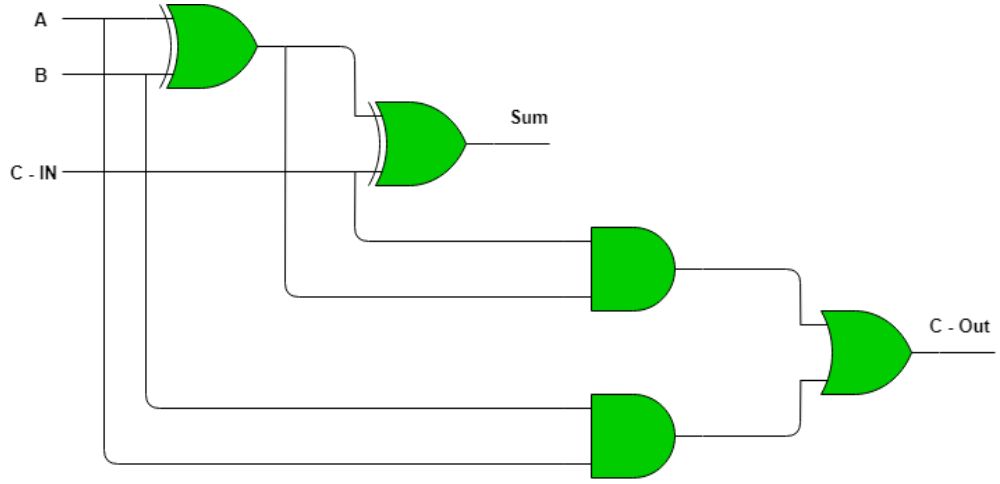


- **Half Adder:** F (X, Y) = X +Y (suma aritmética). A este circuito se lo llama semisumador (half-adder). Podemos usar un XOR para la suma y un AND para el carry.

X	*Y*	*Suma*	*carry*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



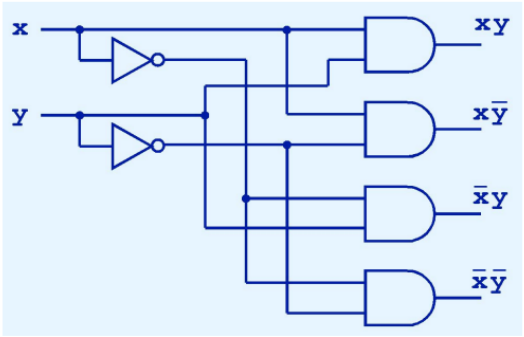
- **Full Adder:** Si debemos sumar numeros de mas de 1 bit es necesario que el adder pueda aceptar el acarrero y de los bits anteriores.



- **Decodificadores:**
  Los decodificadores **de n entradas pueden seleccionar una de 2^n salidas**. Son ampliamente utilizados. Por ejemplo: Seleccionar una locación en una memoria a partir de una dirección colocada en el bus memoria. **Cada combinación de las líneas de entrada corresponderá a una UNICA salida**.

Decodificadores: ejemplo

- Decodificador 2-a-4:



- **Codificador:**
  Una y solo una línea en alto de las entradas corresponderá a una **combinación en la salida.**

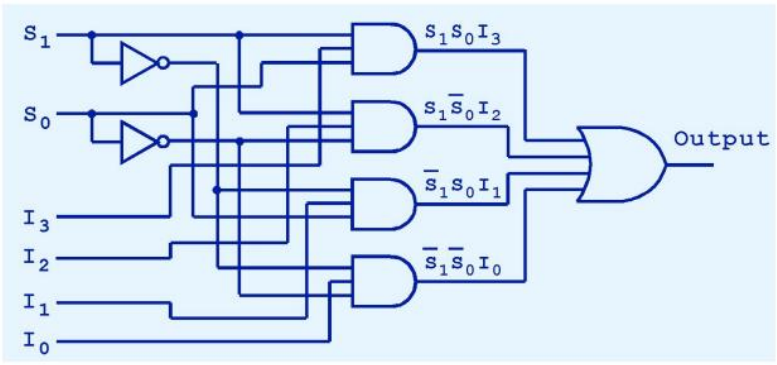
- **Multiplexores:**
  Selecciona una salida a de varias entradas. Seleccionan **una de n entradas** a partir de **$\log_2$ n líneas de control**, es decir:

  La entrada que es seleccionada como salida es determinada por las líneas de control.

  Para seleccionar entre n entradas, se necesitan $\log_2$ n líneas de control.

Multiplexor: ejemplo

- Multiplexor 4-a-1:



- **Demultiplexores:**
  Hacen lo opuesto, dada **una entrada**, seleccionan una de las **n salidas basándose en $\log_2$ n líneas de control.**

Circuitos secuenciales

Computar está vinculado a la posibilidad de almacenar, aun cuando se desee aplicar una función booleana, será necesario almacenar los valores de entrada y el resultado. Si pretendemos implementar un modelo de cómputo que organiza operaciones secuencialmente (por ejemplo, la multiplicación como sumas sucesivas).

Es decir, surge la necesitad de circuitos que puedan “recordar” su estado.

A estos circuitos se los denomina **secuenciales.**

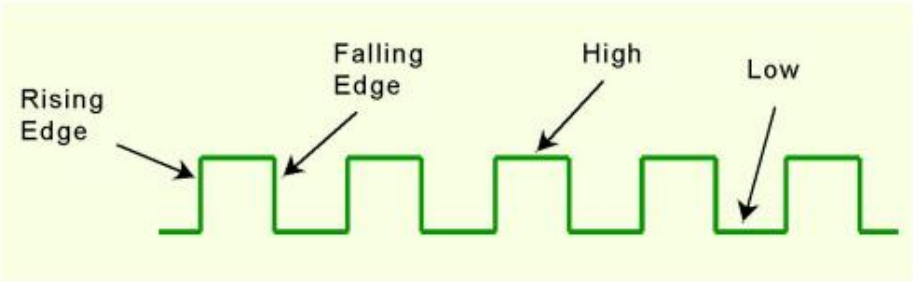
Circuitos sincrónicos:

Cuando se diseña un circuito digital, se debe considerar el comportamiento físico de los circuitos electrónicos. Los circuitos sincrónicos funcionan sobre la base del tiempo. Es decir, las salidas dependen no solo de las entradas. Comienza a jugar el estado en que estaba el circuito y el tiempo. Las salidas de un circuito secuencial dependen de las entradas y del estado anterior a este, por lo que es necesario ordenar los eventos de observación.

Relojes:

En general, necesitamos una forma de ordenar los diferentes eventos que producen cambios de estados. Para esto usamos relojes: Un reloj (clock) es un circuito capaz de producir señales eléctricas oscilantes, con una frecuencia uniforme. Los cambios de estado se producen en cada tick de reloj. Los circuitos pueden tomar diferentes partes de la señal de clock para sincronizarse:

- Cambio de flanco: se detecta cuando hay un cambio en la señal, puede ser flanco ascendente (rising edge) o descendente (falling edge).
- Nivel: se verifica que la señal alcance cierto nivel, puede ser alto (high, es decir, un 1) o bajo (low, es decir, un 0).



Realimentación:

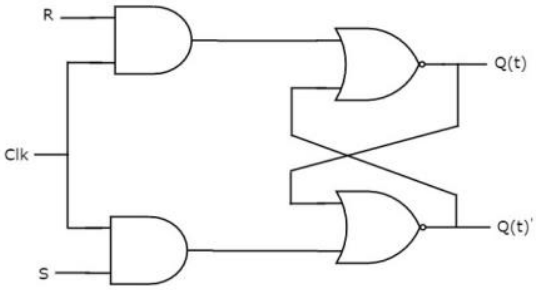
Para retener sus valores, los circuitos secuenciales recurren a la realimentación (feedback). La realimentación se produce cuando una salida se conecta a una entrada. Al introducir retroalimentación, La salida “Q” no puede tomar cualquier cosa. “Q” tomara el valor de “Q’” en el estado anterior.

Flip-Flops:

Los flip-flops son circuitos secuenciales que permiten almacenar el estado de un bit, es decir, guardar un valor y sirven a los efectos de la construcción de memorias.

Se denominan flip-flops porque conservan su valor hasta que se lo reemplaza por uno diferente.

- ***Flip-Flop RS (Reset – set):***
  Uno de los circuitos secuenciales más básicos es el flip-flop SR. Solo cuentan con líneas de control que se denominan asincrónicos puesto que el estado puede ser modificado cada vez que cambia R o S.



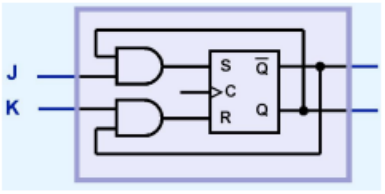
S	R	$Q(t)$	$Q(t+1)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	**indefinido**
1	1	1	**indefinido**

Q(t) es la salida en el tiempo t. Q(t+1) es el valor de Q en el próximo ciclo de clock. Notar los dos valores indefinidos, cuando las entradas S y R son 1, el flip-flop es inestable.

- **Habilitación de datos (Enable):**
  Es común que en los circuitos requieras una línea de habilitación de datos que permite determinar cuándo leer/grabar información a pesar del valor presente en la entrada.
  Las compuertas AND proveen un mecanismo sencillo para implementar una señal de ENABLE.

- **Flip-Flop JK:**

Si aseguramos que las entradas al SR no estarán nunca las dos en 1, el circuito se volvería estable. El flip-flop modificado se denomina JK, en honor de Jack Kilby. Es posible realizar esta modificación tal que:



J	K	$Q(t+1)$
0	0	$Q(t)$ no hay cambios
0	1	0 (reset, ponemos cero)
1	0	1 (set, ponemos uno)
1	1	$\overline{Q}(t)$

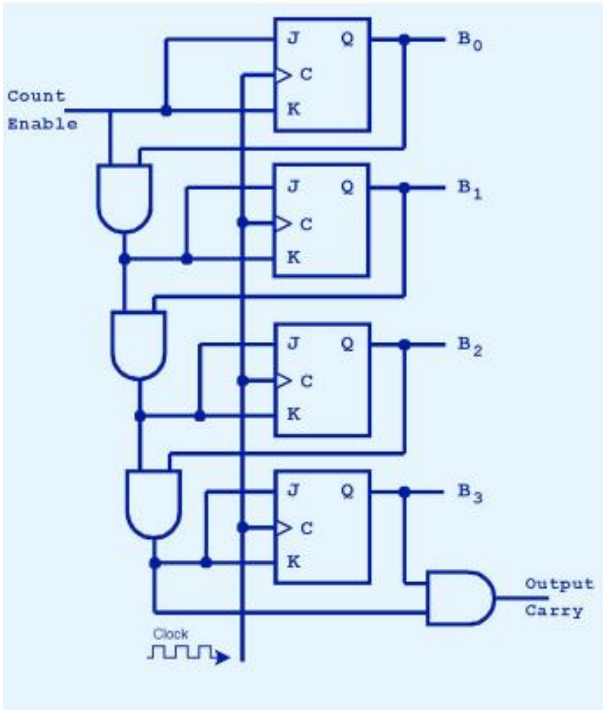
Queda definido, pero en un estado inestable (empieza a oscilar)

- **Flip – Flop D:**

Otra modificación al flip-flop SR es el denominado flip-flop D. Retiene el valor de la entrada al pulso de clock, hasta que cambia dicha entrada, pero al próximo pulso de clock. El flip-flop D es el circuito fundamental (celda) de la memoria de una computadora (1 bit). En este caso el circuito es estable en todos los estados.

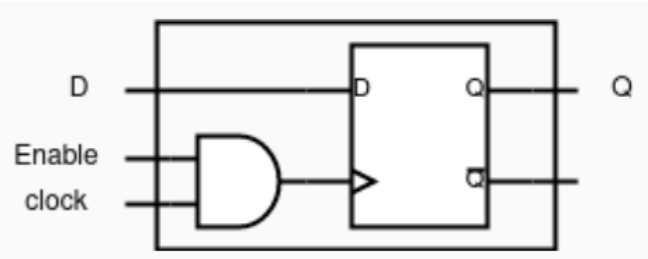
- **Contadores:**

Un contador binario es otro ejemplo de circuito secuencial. El bit de menor orden se complementa a cada pulso de clock. Cualquier cambio de 0 a 1, produce el próximo bit complementado, y así siguiendo a los otros flip-flops.

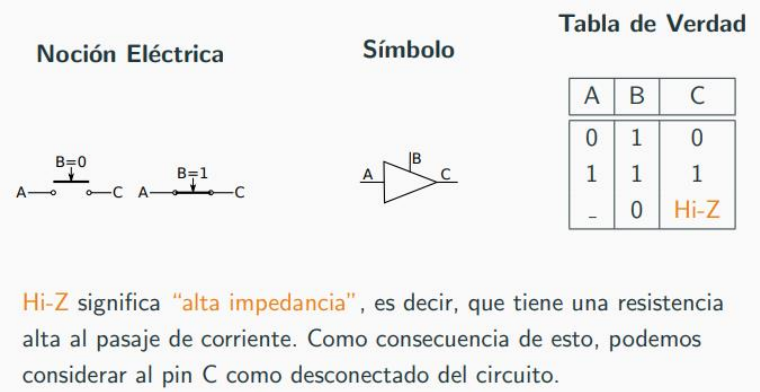


- **Registros y memorias:**

Un flip-flop D puede almacenar un bit, pero solo durante un clock. Debemos poder elegir, con una entrada adicional de control, por cuanto tiempo queremos almacenar -> **ENABLE**



- **Componentes de tres estados:**



A	B	C
0	1	0
1	1	1
_	0	Hi-Z

ISA: Instruction Set Architecture

- **¿Cuál es la diferencia entre Organización y Arquitectura?**

La arquitectura es a lo que se enfrenta el usuario. La organización es como se implementa el programa.

- **¿Qué es una Arquitectura?**

*En informática , una **arquitectura de conjunto de instrucciones** (**ISA**), también llamada **arquitectura informática ,** es un modelo abstracto de una computadora .

En general, una ISA define las instrucciones admitidas , los tipos de datos , los registros , el soporte de hardware para administrar la memoria principal , las características fundamentales (como la consistencia de la memoria , los modos de direccionamiento , la memoria virtual) y el modelo de entrada/salida de una familia de implementaciones. del ISA.

Una ISA especifica el comportamiento del código de máquina que se ejecuta en implementaciones de esa ISA de una manera que no depende de las características de esa implementación, proporcionando compatibilidad binaria entre

implementaciones. Esto permite múltiples implementaciones de un ISA que difieren en características como el rendimiento , el tamaño físico y el costo monetario (entre otras cosas).

*data sacada de Wikipedia

- **Tipos de arquitectura:**

  Stack architecture: se cuenta con una pila y las operaciones se realizan sobre los elementos almacenados en ella accesibles desde el tope.

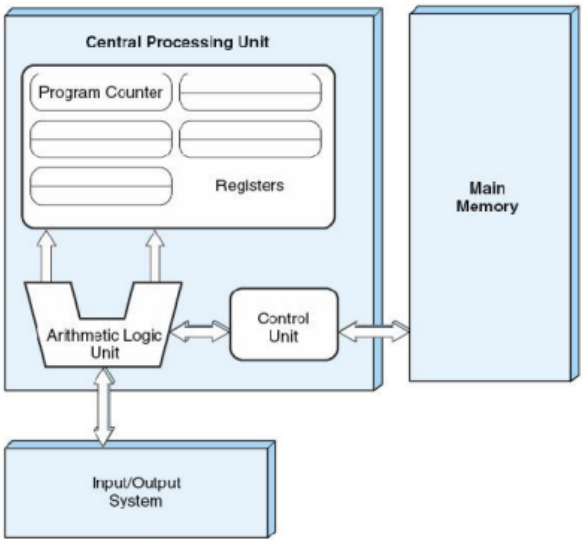
  Accumulator: se cuenta con un registro distinguido y las operaciones tienen como operando implícito dicho registro.

  General purpose registers: se cuenta con un banco de registros y las operaciones se realizan entre ellos así que, todos los operandos y el destino del resultado son explícitos.

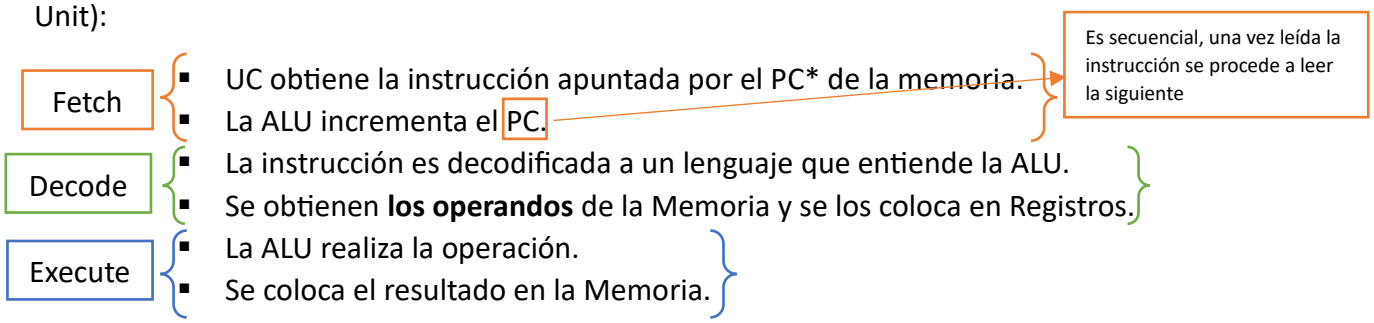
- **En esta materia utilizamos La arquitectura de Von Newmann:**
  - Los programas y los datos se almacenan en la misma memoria sobre la que se puede leer y escribir.
  - La operación de la maquina depende del estado de la memoria.
  - La memoria es redireccionarle.
  - La ejecución es secuencial.

Dicha arquitectura tiene 3 componentes principales:

- CPU: la Unidad Central de Procesamiento (**C**entral **P**rocessing **U**nit) contiene la Unidad de Control, la ALU y los Registros.
- Memoria: Almacenamiento de programas y datos.
- Sistema de entrada y salida.



La arquitectura de Von Newman sigue un determinado ciclo de instrucción (UC: Control Unit):



*PC= Point Counter: si registró una instrucción, el PC es quien apunta esta información.

- **Instruction Set Architecture:**

La ISA es el límite entre el software y el hardware. Está íntimamente relacionada con la organización del sistema pues se apoya sobre su implementación en términos de componentes electrónicos.

Define la forma en la cual un programador observa la arquitectura del sistema.

Define cómo se representan los datos, como se almacenan, como se acceden, qué operaciones se pueden realizar y cómo se codifican estas operaciones.

  - **Propiedades:**
    - **Complejidad del conjunto de instrucciones:** La ISA poseen instrucciones que indican operaciones posibles entre datos. Las operaciones usuales son de movimiento de datos (Mov, Load, Store, …), Aritméticas (suma: ADD, resta: sub, …), Lógicas (And, Or, Xor, …), etc. Las instrucciones tienen una estructura dada por cada ISA en específico.
    - **Longitud de las instrucciones:** puede ser fija o variable.
    - **Cantidad de memoria utilizado por un programa.**
    - **Datos: tipos de datos disponibles y representación de cada tipo de datos.**

- **Acceso a los datos:**

  Los datos se almacenan en registros, memorias, en el stack, en espacios de entrada y salida. Se puede acceder a estos datos con modos de direccionamiento. Cada ISA posee su propio modo.

  Para hablar de direcciones hay que hablar primero de los que solemos llamar **memoria principal**, que se puede entender como:

    - El dispositivo que almacena temporariamente y que permite acceder a la información con la que opera el procesador (capacidad lecto-escritura).

- El espacio contiguo y lineal de información sobre el cual opera nuestro modelo de cómputo.

Las dos cosas están vinculadas porque el espacio lineal modela a la información disponible en el dispositivo físico.
¿Qué significa contiguo y lineal? -> que interpretamos a la memoria como una secuencia de números ordenados, uno junto al otro.

El espacio de memoria es la unión de la información disponible para el procesador junto con la forma de acceder a ella. Al considerar a la memoria contigua y lineal podemos indicar la ubicación de un dato dentro de la misma por la posición que ocupa en una secuencia ordenada de tamaño fijo (dirección). Por ejemplo: si interpretamos a la memoria como una secuencia de datos de 1 byte, podemos indicar la ubicación del tercer byte solo por su posición 0x03 (0x es el sufijo para la notación hexadecimal).

La interpretación del tipo de dato, dentro de nuestra arquitectura, viene dado por el uso. Si queremos interpretar nuestros datos como enteros de dos bytes, el espacio de nuestra información será de la forma:

DATO	0x31C0	0x44A5	…	0xE36E

Pero en la memoria se encuentra de esta forma:

Dirección	0x00	0x01	…	0xFE	0xFF
DATO	0xC0	0x31	…	0x6E	0xE3

¿Cómo accedo a un dato que se encuentra en la posición 127?

Hay una aritmética implícita en el acceso a un dato de memoria.

Por ejemplo, si queremos acceder a un dato que es de dos bytes en la posición 2, entonces dependemos del direccionamiento. El direccionamiento a “x” bytes indica cuanta información hay entre dos posiciones contiguas de memoria.

Direccionando de a 1 byte:

- 0x00 apunta al primer byte
- 0x01 apunta al segundo byte
- 0x02 apunta al tercer byte y así hasta llegar al límite.

Direccionando de a 2 bytes:

- 0x00 apunta al primer byte
- 0x01 apunta al tercer byte
- 0x01 apunta al quinto byte y así hasta llegar al límite.

En general, si quiero acceder a un dato de tamaño “T” en la posición “i” de una memoria M con direcciones de tamaño “t”, la aritmética del acceso sería:

$$M_T [i] = M [i * (T/t)]$$

Básicamente hay que escalar el índice del dato que queremos recuperar por la relación entre el tamaño del mismo y el tamaño del direccionamiento T/t.

Entonces si quiero acceder al dato en la posición 127: (suponiendo que la memoria se direcciona de a 1 byte (t) y el dato es de 2 bytes (T))

$$M [126 * (2/1)] = M [254] = M[0xFE]$$

(recordamos que el primer índice de memoria es el 0x00 entonces la posición 127 se representa en la 126.)

Se puede pensar a la memoria como un vector de cajas:

la dirección: es el índice de cada caja

el tamaño de la dirección: se refiere a cuantos bits tiene el índice (o sea ahí te está diciendo cuantos índices puede haber)

direccionar a x: es el tamaño de cada caja, o sea cuanta información entra en cada uno de esos índices.

¿Utilizando direcciones de tamaño A que apuntan a datos de tamaño t, a cuanta información puedo apuntar?

La fórmula sería 2^A*t, por eso el tamaño de la dirección en sí y del dato apuntado determina la cantidad máxima de información a la que puedo hacer referencia.

¿Qué significa lo siguiente?: **Esta arquitectura tiene direcciones de 16 bits, direccionables a 4 bytes y opera con palabras de 64 bits.**

Que cada dirección de la memoria apunta a una porción distinta de cuatro bytes.

Que cada operación de transferencia entre registros va a copiar 64 bits (8 bytes), que para conseguir un dato de 64 bits voy a calcular M[i*(64/32)] y que la cantidad de memoria direccionable es 2^{16}*32 = 256Kb

- **Formato de instrucción:**

  Como dijimos previamente, una ISA posee varias propiedades que definen su comportamiento y sus características elementales.

  En el diseño de una ISA se consideran:

  - Tipos de operaciones
  - Números de bits por instrucción
  - Uso de stack, registros, etc.
  - Como se almacenan los datos
  - Numero de operandos por instrucción

- Operandos implícitos o explícitos
- Ubicación de los campos
- Tamaño y tipo de los operandos

Cuando nos enfrentamos por primera vez a una ISA es importante conocer esta información. Debemos saber el tamaño de las instrucciones, cuantos bits ocupan el op-code ya que limitan la cantidad de instrucciones, cuantos registros posee ya que cuantos más registros es más cómodo para el programador, pero más costosos para la instrucción, como serán los direccionamientos a memoria, etc.

- **Tamaño de memoria:**
  Si deseamos conocer el tamaño de la memoria direccionable debemos calcular:
  Tamaño unidad direccionable x cantidad de direcciones

  Si deseamos conocer la cantidad de direcciones posibles:
  Tamaño de la Memoria / Tamaño Unidad Direccionable

Es decir, es importante tener en cuenta como se direcciona la memoria -> Esto viene definido por la organización de la computadora -> ¿Es direccionable a Word? ¿Es direccionable a byte? ¿Es direccionable a 16 bits siendo la Word de 32 bits?

- **Operandos:**

  Los operandos indican la estructura que conllevaran las instrucciones, por ejemplo:

Risc – V y Mainframes son arquitecturas que utilizan 3 operandos -> A = B + C

Intel o Motorola son arquitecturas que utilizan 2 operandos donde al menos uno debe ser un registro -> A = A+B

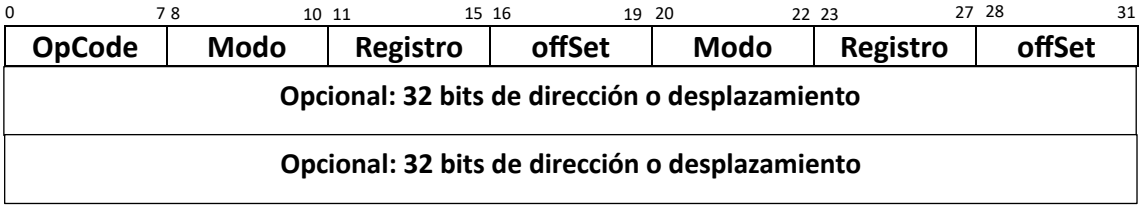
En arquitecturas de tipo accumulator utilizan 1 operando -> AC = AC + A (+operando implícito. Registro acumulable).

En arquitecturas de pila no se utilizan operandos -> push (pop () + pop ()) (+operando implícito: pila).

- **Ortogonalidad:**

Máxima ortogonalidad: cualquier instrucción puede ser usada con cualquier modo de direccionamiento. Es una característica “elegante” pero muy costosa. Implica tener muchas instrucciones.

- **Ejemplo final:**



¿Qué podemos decir de este formato de instrucción?

- Las instrucciones son de longitud variable: de 32 bits, 64 bits o 96 bits
- Hay 8 bits para opCode -> 2^8 instrucciones = 256 como máximo
- Hay 5 bits para identificar registros -> 2^5 registros = 32 como máximo
- Hay 3 bits para identificar despalzamiento o escala (offset)
- Hay 4 bits para direcciones / datos/desplazamiento
- Es necesario leer una segunda palabra para valores inmediatos, direcciones, etc.

General Porpose registers (ORGA 1)

La máquina de Orga 1 es una máquina de registros de propósito general: todos los registros en principio sirven para cualquier cosa (No hay registros privilegiados).

- **Modos de direccionamiento:**
    - **Inmediato:** El operando es un valor y por lo tanto no requiere acceso a memoria
    - **Directo:** El valor se encuentra en la dirección de memoria que figura como operando en la instrucción. Es el tipo de acceso que se utiliza cuando se usan variables.
    - **Indirecto:** El valor se encuentra en la dirección de memoria que se encuentra en la dirección de memoria que figura como operando en la instrucción. Es el tipo de acceso que se utiliza cuando se usan punteros. Existe acceso múltiple a la memoria para encontrar el operando.
    - **Registro:** El valor se encuentra en el registro. Es el tipo de acceso que se utiliza cuando se usan variables.
    - **Indirecto con Registro:** El valor se encuentra en la dirección de memoria que está en un registro. Es el tipo de acceso que se utiliza cuando se usan arreglos.
    - **Indexado:** El valor se encuentra en la dirección de memoria que está en un registro más la constante que acompaña a la

instrucción como operando. Es el tipo de acceso que se utiliza cuando se accede a campos de una estructura.

Para más info. leer la “Cartilla Maquina Orga 1”, en el cual se explican todos los detalles.

- **Etiquetas:** Las etiquetas son renombres de direcciones de memoria. Al momento de programar en Assembler, es imposible saber en qué dirección de memoria se va a guardar la información del programa.
  ¿En qué posición de memoria va a quedar cada etiqueta? Se debe calcular la cantidad de palabras que ocupa cada instrucción y luego se podrá conocer la posición en la que se encuentra dicha etiqueta.
- **Calculo de saltos condicionales:** Los saltos condicionales son relativos. Esto significa que las etiquetas de los saltos son reemplazadas por el desplazamiento necesario para “llegar” desde la dirección en la que estamos parados, hasta la de la etiqueta de destino.

  Destino = Dirección Etiqueta – Dirección instrucción (post fetch)

  Lo cual implica:

  contenido del Pc = Dirección instrucción + incremento

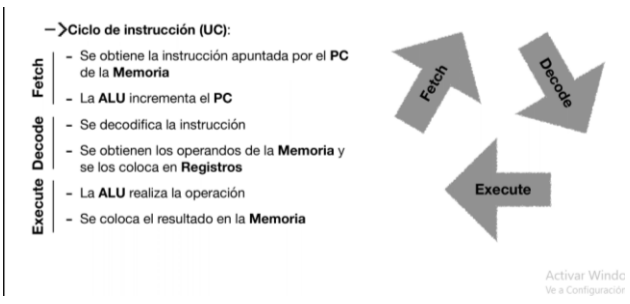
Unidad de Control – Microprogramación

Nos paramos en el nivel 1 de la jerarquía de la computadora.

La unidad de Control es la encargada de orquestar los procesos, marca el tiempo de los eventos que ocurren.

Recordamos que estamos dentro del modelo de Von Newman. Poseemos un procesador que es el que se encarga de hacer las operaciones y también tenemos una memoria que se encarga de guardar los datos con los que vamos a operar.

En el ciclo de instrucción de la arquitectura de von Newman tenemos los 3 pasos conocidos como Fetch, Decode y Execute. Sabemos que en términos generales debemos obtener la información, entender qué instrucción es y volver a ejecutar. Ahora nos comenzamos a preguntar, entre otras cosas, a qué nos referimos por instrucciones, qué tamaño tienen y a qué nos referimos con ejecutar dicha instrucción.

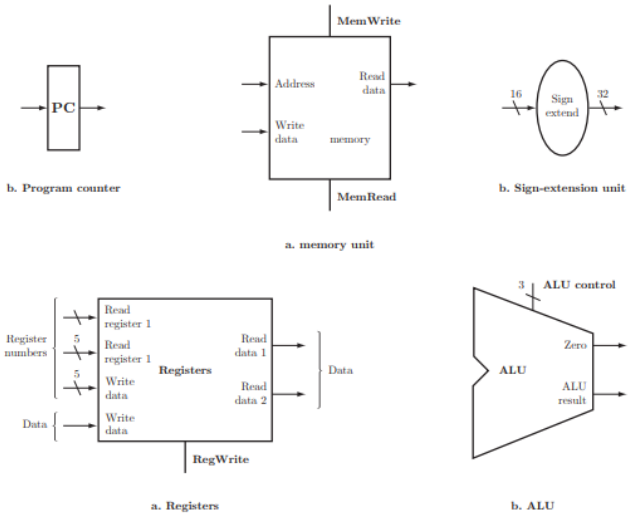


Supongamos que nuestra instrucción posee un formato variable (por ejemplo, en Orga1). Entonces nuestro Fetch se vuelve complicado porque debemos acceder a memoria, y basándose en la información obtenida, debemos identificar qué tipo de instrucción es, para luego, decidir si hay que buscar el resto de los datos si hace falta. En cambio, si el formato es fijo, entonces el Fetch se vuelve muy simple, pues simplemente hay que procesar la información obtenida.

Entonces frente a estas preguntas debemos recordar que recursos poseemos:

- Nuestro diseño de la computadora posee:
    - Única ALU
    - Única Memoria
    - Único Banco de Registros

Recursos disponibles



Por lo tanto, algunos problemas a resolver son:

- Una instrucción puede utilizar un mismo recurso más de una vez lo que hace que se pierdan los valores anteriores.
- Una instrucción puede utilizar el mismo recurso durante la misma etapa para más de una cosa diferente.

Esta es información primordial para el programador pues, por ejemplo, sabiendo que una instrucción reutiliza recursos y corre el riesgo de perder información, debe garantizarse de guardar la información que no desee perder al final de cada instrucción.

Entonces nuestro objetivo es fijar una ISA elegante y ver como podemos llevar esa implementación de las instrucciones a circuitería.

Cada uno de los circuitos que armemos para que se puedan ejecutar las instrucciones de la ISA van a estar mediadas por **señales de control.** Es decir, dado un componente, vamos a tener un PIN tal que si ponemos un 1 entonces el componente realizará una

acción. Por lo tanto, las instrucciones se desglosarán en instrucciones más pequeñas que irán indicando qué componentes accionar dentro de la circuitería. Estas instrucciones más pequeñas son las que se conocen como **microinstrucciones**.

Etapas de ejecución de una instrucción

1. Fetch de la instrucción: **IF** (Fetch)
2. Decodificación (y lectura de registros: **ID** (Decode)
3. Ejecución ó cálculo de dirección de memoria: **EX** (Execution)
4. Acceso a datos en memoria: **MEM**
5. Escritura en registros: **WB** (Write Back)

- **MIPS: Microprocessor without Interlocked Pipeline Stages**

MIPS es una arquitectura sencilla y antigua, aunque actualmente siguen existiendo como procesadores. Se le dice que es un procesador RISC porque posee pocas instrucciones.

MIPS - Microprocessor without Interlocked Pipeline Stages

—>Procesador RISC desarrollado por MIPS Computer Systems

—>32 registros de propósito general (salvo excepciones)

—>3 tipos de instrucciones de 32 bits con el siguiente formato:

Type	-31-	format (bits)				-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

—>El PC es de 32 bits: los 4 más significativos (31..28) dividen la memoria en (Reserved, Text, Data y Stack), los 26 siguientes son la dirección relativa en dicho espacio y los 2 menos significativos son 00 para alinear a palabra.

—>Las instrucciones se dividen en: **1- CPU instructions** (Loads and Stores, ALU, Shifts, Multiplication and division, Jump and Branch y Exception) y **2- Floating-Point Unit instructions** (Arithmetic, Data transfer y Branch)

La arquitectura MIPS posee un coprocesador dedicado a operaciones sobre números de punto flotante.

De esta arquitectura se puede observar que el formato de instrucción es de ancho fijo. Esto logra que el paso del Fetch sea muy sencillo, aunque en ciertas instrucciones se pierde memoria.

Etapas de la ejecución de una instrucción MIPS:

1) Fetch de instrucción
2) Decodificación de instrucción (y lectura de registros) (Decode)

3) Lectura a datos de la memoria previo cálculo de la dirección
4) Ejecución de la instrucción (Execute)
5) Escritura de resultados en la memoria o registros (Write back)

- **RTL – Register Transfer Languaje**

Como dijimos antes, las instrucciones escritas por el programador en Assembler, desde el punto de vista del procesador, serán vistas como una secuencia de instrucciones más pequeñas (las microinstrucciones), que, mejor dicho, es como un programa, al cual lo llamaremos microprograma.

RTL es un lenguaje creado para poder escribir microprogramas y que sean más entendibles por el programador. Se utiliza para describir la secuencia exacta de las micro operaciones.

Es un lenguaje muy intuitivo y se basa en:

- Cada instrucción es implementada como un microprograma.
- Los microprogramas son secuencias de micro operaciones
- Las micro operaciones permiten mover datos entre registros y memoria.
- La ejecución de los microprogramas presenta una precedencia temporal derivada de la utilización múltiple de recursos.

Resumen para cada etapa del ciclo de instrucción

Etapa	Tipo de instrucción	acciones
IF	todas	IR <- mem[PC] PC <- PC + 4
ID	todas	A <- R[Rs] B <- R[Rt] ALUout <- PC + (inm16 < <2)
EX	tipo R Load/Store Branch Jump	ALUout <- A op B ALUout <- A + signextend(inm16) if(A==B) then PC <- ALUout PC<31:2> <- PC<31:28>, IR<25:0> < < 2
MEM	Load Store	MBR <- mem[ALUout] mem[ALUout] <- B
WB	tipo R Load	R[Rd] <- ALUout R[Rt] <- MBR

Etapas según el tipo de instrucción:

- Tipo R: 4 etapas (IF, ID, EX y WB)
- Branch y Jump: 3 etapas (IF, ID y EX)
- Store: 4 etapas (IF, ID, EX y MEM)
- Load: 5 etapas (IF, ID, EX, MEM y WB)

Etapa 1: Fetch (IF) *RTL*

- IR <- mem [PC]
- PC <- PC + 4

Etapa 2: Decode (ID) *RTL*

- Opción lectura de registros:
  - A <- R [IR [25:21]]
  - B <- R [IR [20:16]]
- Opción cálculo de saltos:
  - ALUout <- PC + signextend (IR [15:0]) < < 2

Etapa 3: Execute (EX) *RTL*

- Opción aritmética/lógica:
  - ALUout <- A op B
  - ALUout <- A op signextend (IR [15:0])
- Opción dirección (Load o Store):
  - ALUout <- A + signextend (IR [15:0])
- Salto condicional:
  - Si A = B, PC <- ALUout
- Jump:
  - PC [31:28] <- PC || IR [25:0] < < 2

Etapa 3: Memoria (MEM) *RTL sólo para Load y Store*

- Ï Opción lectura:
  - MBR <- mem [ALUout]
- Opción escritura:
  - Mem [ALUout] <- B

Etapa 3: Escritura (WB) *RTL*

- Opción tipo R o aritmética inmediata:
  - R [IR [15:11]] <- ALUout
- Opción escritura:
  - R [IR [20:16]] <- MBR

Paso 1: requerimientos de la ISA

- Memoria
  - Para instrucciones y datos
- Registros 32x32
  - Leer RS, leer RT
  - Escribir RT ó RD
- PC, MDR
- A, B para datos intermedios, ALUout (retiene salida de la ALU)
- Extensor de signo de 16 a 32 bits
- Sumar y restar registros y/o valores inmediatos
- Operaciones lógicas (and/or) con registros y/o valores inmediatos
- Sumar 4 al PC ó 4+inmediato extendido * 4

Microarquitectura – Introducción

- **El programa como objeto de estudio:**

Podemos interpretar el programa como la composición de instrucciones que indican como ha de modificarse el estado del procesador en base a la semántica de ejecución asociado a éstas.

Podemos definir el estado del programa como el conjunto de valores asociado tanto a los registros (propósito general, PC, registros internos) como a las distintas posiciones de memoria.

La semántica asociada a las instrucciones puede indicar como se transforman los valores de los registros asociados a datos del programa (registros del propósito general y memoria) o al control de flujo del mismo (PC, SP).

La semántica de las instrucciones induce un conjunto posible de ejecuciones en base a su estado inicial (PC, registros, memoria) y las modificaciones introducidas por la lógica de control de flujo (saltos).

Las instrucciones se codifican y almacenan en memoria y la modificación de flujo de ejecución necesita conocer la propia estructura del programa.

- **¿Cómo reproducimos la idea de ejecución secuencial u ordenada del programa?**

Con un registro de propósito particular (PC) que indica de qué dirección de memoria tomar la próxima instrucción.

- **Arquitectura de OrgaSmall:**
  - Arquitectura Von Newmann, memoria de datos e instrucciones compartida.
  - 8 registros de propósito general, R0 – R7.
  - 1 registro de propósito específico PC.
  - Tamaño de palabra de 8 bits e instrucciones de 16 bits.
  - Memoria de 256 palabras de 8 bits.
  - Bus de 8 bits
  - Diseño micro programado

- **Datapath**

Cuando hablamos de datapath nos referimos al flujo de datos. Analizar el datapath nos permite reproducir el mecanismo deseado de Fetch – Decode – Execute.

  - **Ciclo de Fetch – Decode**

    Participan el PC, la memoria y la unidad de Decode. El PC no solo encapsula un registro, sino que expone también una señal de control que permite incrementarlo (en una palabra).

Veamos que secuencia de microinstrucciones (RTL) debería suceder para conseguir que la instrucción almacenada en memoria (de 2 palabras de 8 bits) se almacena en los registros internos del módulo de decodificación:

MM_{addr} := PC

DE_H := MM_{DATA}

PC_{INC}

MM_{ADDR} := PC

DE_L := MM_{DATA}

PC_{INC}

Aquí las asignaciones (:=) indican la activación de un par de señales write/ enable Out y las declaraciones aisladas (PC_{INC}) indican la activación durante in ciclo de la señal indicada.

PC_enOut MM_enAddr

MM_enOut DE_loadH PC_inc

PC_enOut MM_enAddr

MM_enOut DE_LoasL PC_inc

Pero al escribir el microprograma de su implementación se indican las señalas de esta manera.

Cada línea se corresponde con un ciclo de reloj y los nombres declarados en cada una, son las señales de control que se activan a la vez en cada uno de estos ciclos.

- **Ciclo de EXECUTE:**
  Participan la ALU, que resuelve la aritmética, los registros y el decode que indica qué registros participan.
  Veamos que secuencia de microinstrucciones (RTL) debería suceder para conseguir que se transfieran los valores de los registros indicados en la instrucción a la ALU, se realice la operación y se copie el resultado en el registro de destino.

ALU_A := R_A

ALU_B := R_B

ALU_{Add}

R_A := ALU_{OUT}

Aquí las asignaciones a RA, RB indican no solo la activación de un par de señales write / enable Out sino que indican el índice del registro de interés a partir de los bits correspondientes al operando de la instrucción (Decode y multiplexor correspondiente).

RB_enOut ALU_enA RB_selectIndexOut = 0

RB_enOut ALU_enB RB_selectIndexOut = 1

ALU_OP= ADD ALU_opW

RB_enIn ALU_enOut RB_selectIndexIn = 0

El microprograma asociado

- **Datapath del STR**
  En la ejecución (EXECUTE) del STR participan el controlador de memoria, los Registros y el Decode que indica el registro fuente y la dirección destino.
  Veamos que secuencia de microinstrucciones (RTL) debería suceder para conseguir que se transfiera la dirección de memoria al controlador de memoria, y el valor del registro fuente a la dirección indicada.

$M_{Addr} := DE_{imm}$

$M_{Data} := R_A$

Notemos que son simplemente dos asignaciones pero que el controlador tiene entradas de dirección y de datos.

DE_enOutImm MM_enAddr

RB_enOut MM_Load RB_selectIndexOut = 0

El microprograma asociado

Observemos que la asignación desde el Decode al controlador de memoria se reguarda con un tri-estado por realizarse a través de recurso compartido(bus).

- **Saltos Condicionales, ¿Cómo implementarlos?**

  Su expresión en RTL:

  If Z=1

  $PC := DE_{imm}$

  ¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición (Z=1)? -> Tenemos que definir primero como implementar el mecanismo que ejecuta microinstrucciones. Todo explicado en detalle en el apartado de MICRO PC.

- **Datapath del JZ**
  En la ejecución (EXECUTE) del JZ participan el controlador de memoria, el PC que puede o no ser sobrescrito, el Decode que indica valor con el cual podría actualizarse el PC, y la ALU cuyos flags determinan si el salto se realiza.

JZ_microOp load_microOp

Reset_microOp

DE_enOutImm PC_Load

Reset_microOp

El microprograma asociado

¿Qué función cumple la señal **reset_microOp** aquí? -> Esto tiene que ver con cómo ubicamos a los microprogramas en la memoria. Vuelve el micro PC a cero, se cierra el ciclo regresando al FETCH luego del EXECUTE de cada instrucción.

- **MICRO PC**

  Así como existe un registro de propósito específico que indica de qué dirección de memoria tomar la próxima instrucción (PC), existe otro registro interno que indica cuál es la micro instrucción que va a ser ejecutada en el siguiente ciclo de reloj, el MICRO PC.

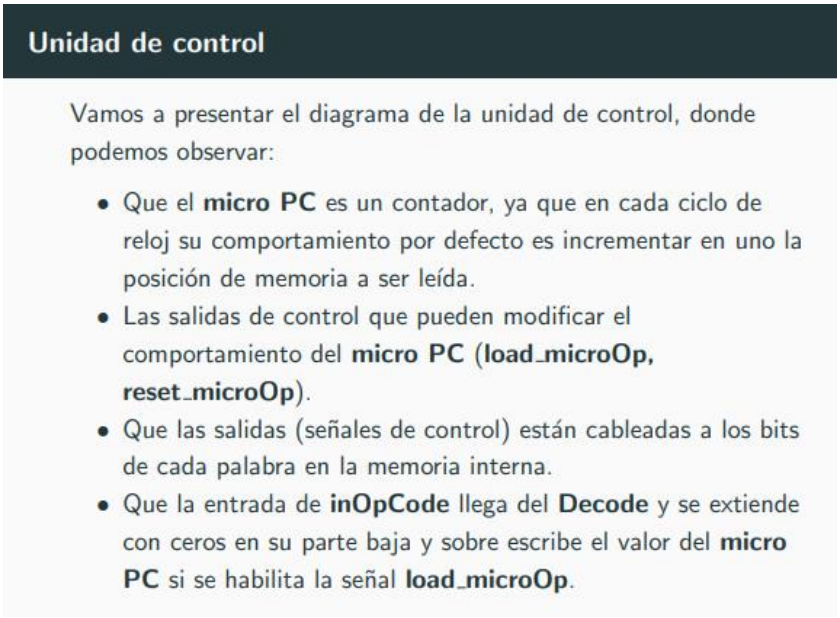
¿A qué dirección de memoria hace referencia? -> Unidad de control.

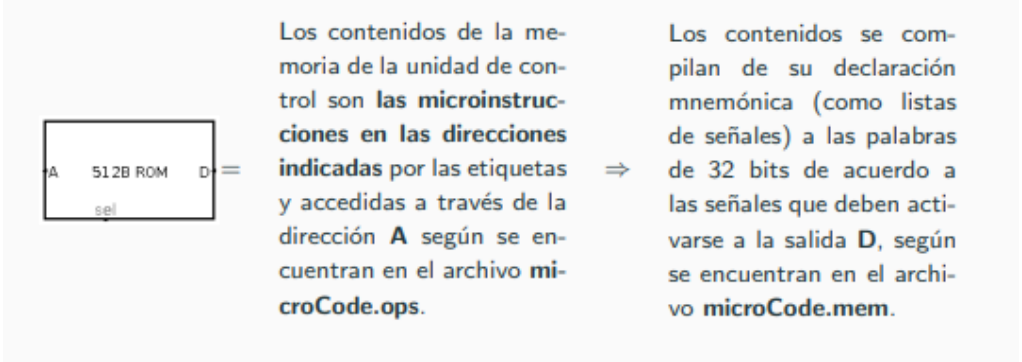
Los microprogramas que permiten ejecutar las acciones asociadas con cada instrucción de nuestro lenguaje (ASM) se ejecutan dentro de un componente llamado unidad de control que cuenta con una memoria interna donde almacena la codificación de los microprogramas.

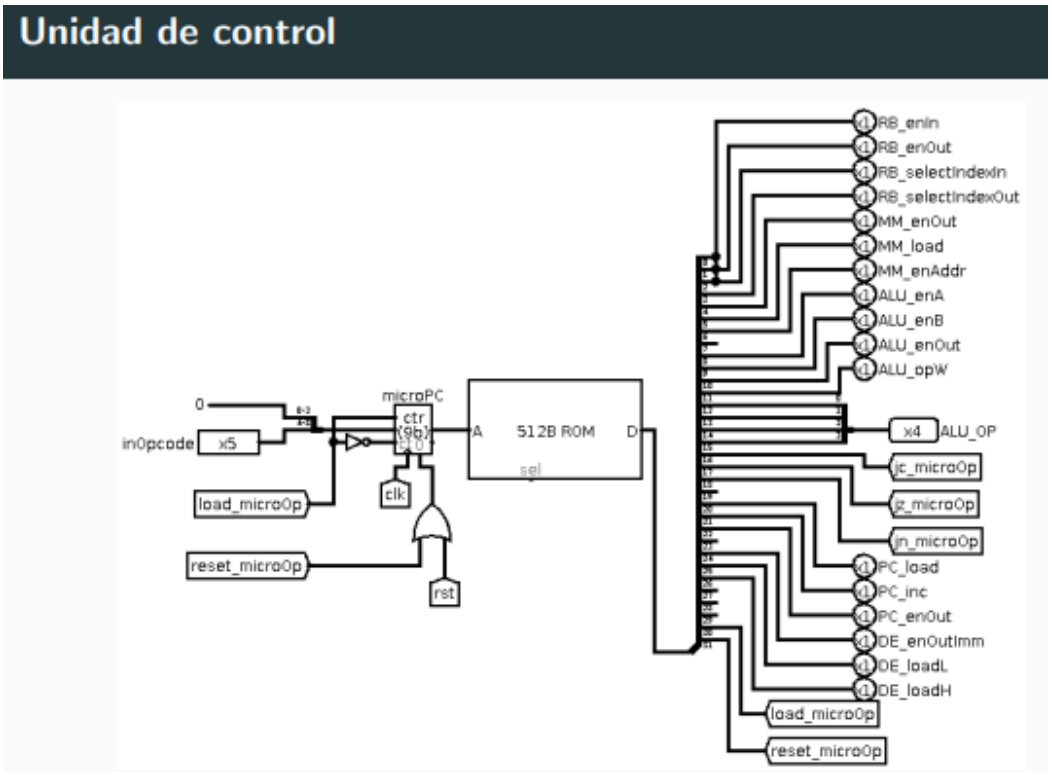
Esta memoria está compuesta por palabras que en nuestro caso son de 32 bits, se acceden a través de direcciones de 9 bits y cada bit dentro de una palabra determina el valor de una señal de control dentro de nuestra organización.

Por esos, sus microprogramas escritos como conjunción de señales se traducen a una serie de palabras de 32 bits.

- **Diagrama de la unidad de control:**







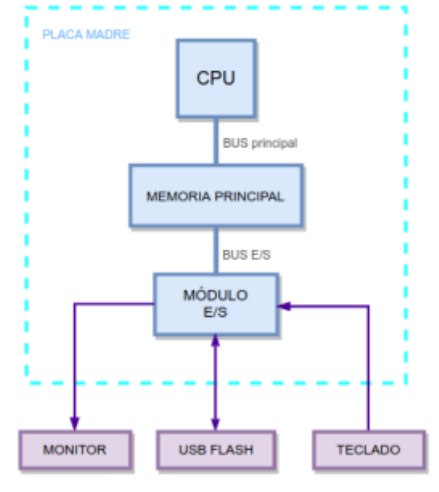
ENTRADA / SALIDA

Interactuamos con el exterior con dispositivos de Entrada/Salida.

- Entrada: teclado, mousepad, lector de huella, webcam, etc.
- Salida: monitor, placa de sonido, impresora, etc.
- Entrada/Salida: disco rígido, router, USB flash, etc.

Cada dispositivo de E/S tiene sus propios registros en los que la CPU puede leer o escribir datos.

- **Tipos de registro:**
  - Lectura (para dispositivos de Entrada)
  - Escritura (para dispositivos de Salida)
  - Lectura / Escritura (para E/S)



¿Qué instrucciones nos permiten acceder a estos registros? -> Dos formas principales:

- E/S independiente (IN y OUT)
- E/S mapeado a direcciones de memoria (direcciones de memoria principal reservados)

¿Cómo controlamos los dispositivos? -> Métodos de control de E/S:

- Polling: continuamente se debe consultar la llegada de nueva información. El problema es que puede ocurrir que la información llegó, pero hasta que no se preguntó, no se lee.
- Interrupciones: se deja la actividad actual para hace otra cosa, pero al volver debo recordar en donde me había quedado.

- **Polling:**

El sistema posee al menos un registro exclusivo para cada dispositivo, la CPU los debe monitorear continuamente.

Cuando la CPU detecta que en algún registro hay un dato de relevancia, entonces actúa según corresponda.

Es el método más lento, pero que tiene una complejidad baja en cuanto al hardware. El modulo de E/S se conecta solo con la memoria, no tiene conexión con la CPU.

¿en qué casos usamos polling?

- Se tiene un sensor que mide un valor de la realidad (temperatura, humedad, etc.). Si me pierdo algún dato no hay problema.
- Un sensor no tiene capacidad para interrumpir.
- La CPU no tiene otra cosa por hacer o disponemos de múltiples unidades de procesamiento.
- Se conoce que el sensor muestra valores con cierta frecuencia.

- **Interrupciones:**

La CPU no está constantemente monitoreando los registros.

Cada dispositivo interrumpe a la CPU al enviar una señal de interrupción.

Es por eso que hay una conexión entre el Modulo E/S y la CPU.

La CPU debe interrumpir su tarea para atender al dispositivo. Para ello guarda la información del contexto actual, carga la dirección de la rutina de atención de interrupción, la ejecuta y al terminar restaura el estado anterior.

El procesador ORGA1i es una extensión de ORGA1 Con la capacidad de atender una única interrupción (enmascarable) de un dispositivo E/S.

Para esto tenemos:

- Nuevas señales:
    - Entrada: INT (pedido de interrupción)
    - Salida: INTA (reconocimiento de interrupción) la cual es enviada por el procesador.
- Nuevo flag:
    - **I** -> indica si el procesador puede ser interrumpido.
- Nuevo registro:
    - PSW (Program Status Word), donde se almacenan los flags
- Nuevas instrucciones.
- Nueva dirección reservada:
    - 0x0000 donde se indica la dirección de la rutina de atención de las interrupciones

¿Qué pasa cuando interrumpen a la CPU?

- En el caso del procesador ORGA1i, si el dispositivo de E/S activa la señal de interrupción y el flag **I** vale 1, realiza los siguientes pasos atómicamente.
    - Asigna [SP] = PSW y decrementa SP
    - Asigna [SP] = PC y decrementa SP
    - Realiza I <- 0 Pues es para que el procesador vuelva a ser interrumpido.
    - Asigna PC = [0x0000]

¿Cuándo se lee la instrucción? -> Luego del execute.

¿En que dirección se guarda la posición de inicio de la instrucción de la interrupción? -> en 000

Resumen

La **Velocidad** para cada método de E/S depende del *Hardware* dedicado.
Lo que no hace el hardware, lo tendrá que hacer el *Software* ejecutando instrucciones:

Método de E/S	*Hardware*	*Software*	*Velocidad*
Polling	*	***	*
Interrupciones	**	**	**
DMA	***	*	***