# Arquitectura de Procesadores

David Alejandro González Márquez

Programación de softcores en FPGAs
Programa de Profesoras/es Visitantes
Departamento de computación
Universidad de Buenos Aires

Clase disponible en: `https://github.com/fokerman/fpgaSoftcoreProgrammingCourse`

# Instruction Set Architecture (ISA)

## Instruction Set Architecture (ISA)

Es la interfaz entre los comandos de *software* y el *hardware* que lo lleva a cabo.

El ISA especifica:

- **La organización de la Memoria**
  - Espacio de direcciones ($2^{32}$, $2^{48}$, $2^{64}$, ...).
  - Unidad direccionable (8 bits, 16 bits, 32 bits, ...).
  - Orden de los datos (*big-endian* o *little-endian*).
- **El conjunto de Registros**
  - Cantidad e indentificación de registros (R0, R1, R2, ... o EAX, EBX, ECX, ...).
  - Tipos de uso de registros (dedicados o de propósito general).
- **El conjunto de Instrucciones**
  - Codificación de instrucciones (tamaño fijo o variable, opcodes, predicados, ...).
  - Tipos de datos (int8, int16, int32, float, double, ...).
  - Modos de direccionamiento (directo, indirecto, indexado, ...).
  - Descripción de operaciones (add, sub, or, and, not, ...).

# Instruction Set Architecture (ISA)

**Instruction Set Architecture (ISA)**

Es la interfaz entre los comandos de *software* y el *hardware* que lo lleva a cabo.

El ISA especifica:

- **Mecanismos de entrada/salida**
  - Espacios de memoria independientes o compartidos (in, out).
  - Mecanismos de interrupciones.
- **Soporte de sistemas**
  - Administración de memoria.
  - Administración de procesos.
  - Administración de energía.
  - Administración de modos de operación.
  - Administración de hardware.

# Modelo de Von Neumann: Componentes básicos

**John von Neumann** propuso el modelo fundamental de una computadora para el procesamiento de programas en 1946.

Este modelo está basado en dos **principios**:

- **Programa almacenado**
  - Las instrucciones se guardan en una **arreglo lineal de memoria**.
  - La **memoria es unificada** entre datos e instrucciones.
  - La **interpretación** de los datos almacenados depende de señales de control.
- **Programa secuencial**
  - Una instrucción es procesada **por unidad de tiempo** (fetch, decode, execute).
  - El PC define la **próxima instrucción** a ejecutar.
  - El PC aumenta **secuencialmente** (excepto para las instrucciones de control).

y está compuesto de 5 **componentes básicos**:

- Memoria
- Procesador
- Unidad de control
- Interfaz de entrada
- Interfaz de salida

# Arquitecturas de procesadores

1974 **8080** - 8 bits - Intel

1974 **6800** - 8 bits - Motorola

1974 **Z80** - 8 bits - Zilog

1975 **MOS 6502** - 8 bits - MOS Technology.

1978 **x86, x86-64, AMD64** - 16, 32, 64 bits - Intel/AMD.

1979 **68000** - 16, 32 bits - Motorola

1983 **ARM (Advanced RISC Machine)** - 32, 64 bits - Licenciado por ARM Holdings.

1984 **MIPS** - 32, 64 bits - MIPS Technologies (comprado por Hewlett Packard)

1986 **PA-RISC** - 32 bits - Hewlett Packard

1987 **SPARC** - 32 bits - Sun Microsystems (comprado por Oracle)

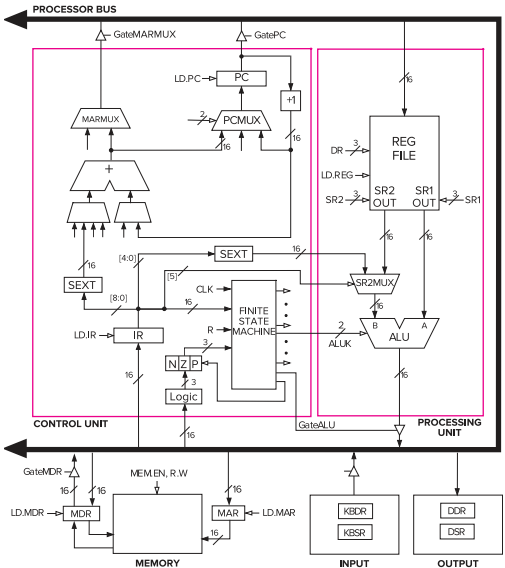
1992 **DEC Alpha** - 64 bits - Digital (comprado por Hewlett Packard)

1992 **PowerPC** - 32, 64 bits - IBM

2001 **Itanium** - 64 bits - Hewlett Packard y Intel

2007 **Cell** - 32, 64 bits - IBM

# LC-3: Ejemplo de una máquina von Neumann



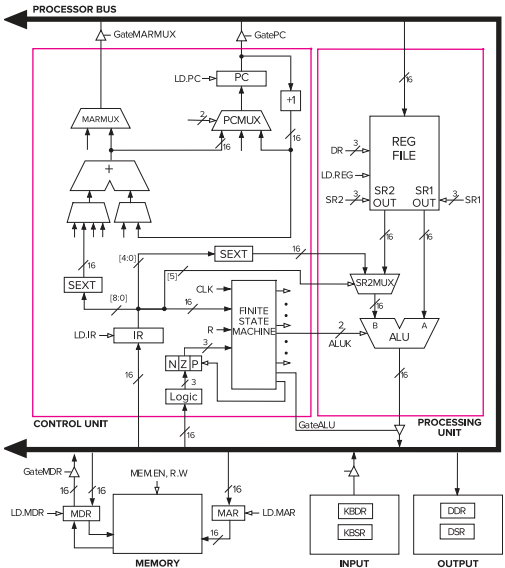

La LC-3 es una computadora simple que presentaremos para estudiar su funcionamiento.

En el datapath se pueden identificar todos los componentes básicos de una máquina von Neumann.

Notar que:

- Las flechas de punta negra identifican el **camino de los datos**.
- Las flechas de punta blanca identifican las **señales de control**.

# LC-3: Ejemplo de una máquina von Neumann



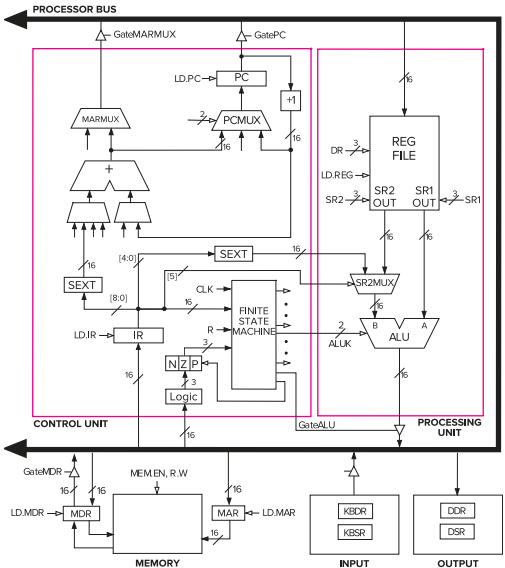

## MEMORY

Consiste en un arreglo de posiciones de memoria accesible por dos registros.

1. **Memory Address Register (MAR)**
   Contiene la dirección de memoria que se busca acceder.
2. **Memory Data Register (MDR)**
   Contiene el contenido de la posición de memoria, tanto para lectura como escritura.

Espacio direccionable dado por MAR de 16 bits, es decir $2^{16}$ posiciones de memoria.
Unidad direccionable dado por MDR de 16 bits.

# LC-3: Ejemplo de una máquina von Neumann




## INPUT/OUTPUT

Consiste en un teclado y un monitor.

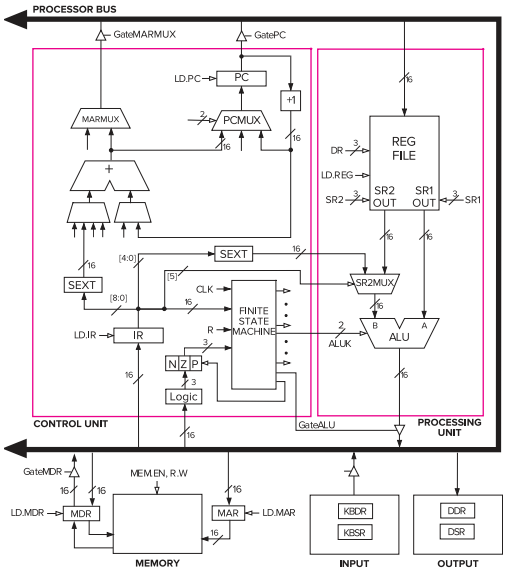
### Keyboard

- **Keyboard data register (KBDR)**
  Mantiene el código ASCII de la tecla presionada.
- **keyboard status register (KBSR)**
  Mantiene información del estado de la tecla presionada.

### Monitor

- **Display data register (DDR)**
  Mantiene el código ASCII del caracter enviado a pantalla.
- **Display status register (DSR)**
  Mantiene información del estado del caracter enviado.

# LC-3: Ejemplo de una máquina von Neumann

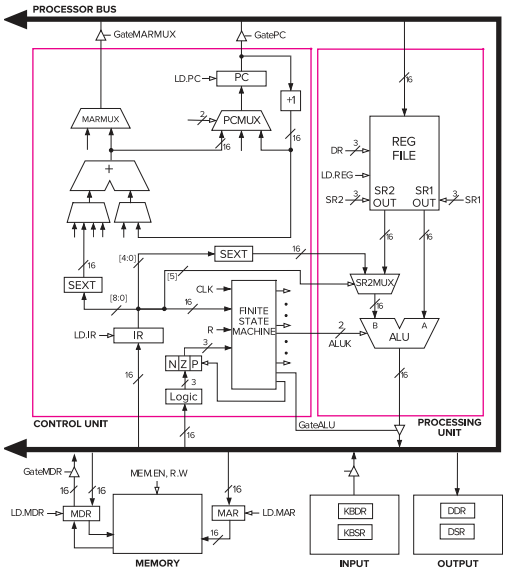



## THE PROCESSING UNIT

Consiste de dos partes:

- **ALU** encargada de realizar las operaciones aritméticas y lógicas. En la LC-3, la ALU está limitada para realizar solo la operación aritmética suma y las operaciones lógicas AND y NOT.
- **Banco de registros** encargado de almacenar datos temporariamente. Contiene 8 registros de propósi to general, un puerto de escritura y dos puertos de lectura.

# LC-3: Ejemplo de una máquina von Neumann




## THE CONTROL UNIT

Contiene en todas las estructuras y módulos necesarios para **administrar el procesamiento** de la computadora.

La parte más importante es la **máquina de estados** que controla el funcionamiento de todo el sistema.

Se ocupa de llevar paso a paso **el ciclo de instrucción** para todas las instrucciones soportadas por el procesador.

Notar que todas las entradas en la máquina de estados son datos, además del reloj, y las salidas son señales de control.

# Procesamiento de Instrucciones

La unidad más básica de procesamiento es la **instrucción**.
Está compuesta principalmente de dos partes:

- **opcode**: Indica la instrucción a realizar.
- **operands**: Indica sobre que valores se va a ejecutar.

En general existen cuatro tipos de instrucciones:

- Instrucciones de **operación** o calculo.
- Instrucciones de **movimiento** de datos.
- Instrucciones de cambio en el **control** de flujo.
- Instrucciones **especiales**.

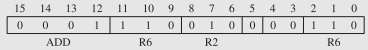
En la LC-3 las instrucciones son de **tamaño fijo de 16 bits**.

- **Bits [15:12]**: Contienen el opcode.
- **Bits [11:0]**: Se utilizan para identificar los operandos.

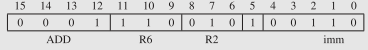
## Ejemplo instrucción ADD

Dos tipos posibles de codificación:

R6 ← R2 + R6




R6 ← R2 + 6




**Bits [15:12]**: `0001`, corresponde al ADD.
**Bits [11:9]**: Registro destino.
**Bits [8:6]**: Registro operando 1.
**Bits [5]**: Indica tipo de operando 2.
**Bits [2:0]**: Registro operando 2, si `Bits[5]=0`.
**Bits [4:0]**: Inmediato con signo extendido como operando 2, si `Bits[5]=1`.

# Ciclo de Instrucción

Las instrucciones son procesadas de forma secuencial una después de la otra.
Si bien internamente podemos romper esté contrato,
a nivel de la arquitectura del sistema lo debemos respetar.

De forma general el ciclo de instrucción consiste en las siguientes etapas:

1. FETCH
2. DECODE
3. EVALUATE ADDRESS
4. FETCH OPERANDS
5. EXECUTE
6. STORE RESULT

Cada etapa puede requerir más de un ciclo de reloj
o incluso puede no ser necesaria para resolver la instrucción.

# Ciclo de Instrucción: FETCH

**FETCH**: Obtiene la instrucción de memoria y la carga en el registro IR.

Pasos:

1. Carga el registro `MAR` con el contenido de `PC` y simultáneamente incrementa el `PC`.
2. Lee la memoria y guarda el contenido en el registro `MDR`.
3. Carga en el registro `IR` el contenido de `MDR`.

Notar que el PC siempre queda apuntando a la **próxima instrucción** que debe ser ejecutada.

Dependiendo de la unidad direccionable de la memoria y el tamaño de la instrucción puede ser necesario **leer más de un valor de memoria**. Incluso puede ser necesario decodificar parcialmente la instrucción para decidir si se debe leer o no los próximos datos de memoria.

# Ciclo de Instrucción: DECODE

**DECODE**: Evaluá la instrucción para determinar que acción realizar a nivel de microarquitectura.

En LC-3 un decodificador de 4 entradas identifica los **16 opcodes posibles** (`IR [15:12]`).
Dependiendo el opcode identificado, se interpretan los 12 bits restantes.

La decodificación puede incluir **generar señales de control o de datos** que dependiendo de la instrucción pueden ser utilizados o no.

Un ejemplo para nuestra máquina puede ser el bit 5 de IR que solo se utiliza sí la instrucción debe utilizar un segundo parámetro.

# Ciclo de Instrucción: EVALUATE ADDRESS

**EVALUATE ADDRESS**: Calcula la dirección de memoria necesaria para procesar la instrucción.

En LC-3, para las instrucciones que requieren acceder a memoria, en esta etapa se calcula la dirección de memoria relativa al PC actual.

*Por ejemplo:*
La instrucción LD, lee un valor de memoria y almacena su contenido en un registro.
La etapa de EVALUATE ADDRESS, toma de la instrucción un valor inmediato de 9 bits que se utiliza como un **desplazamiento** sobre el valor actual del PC, calculando el valor absoluto de la dirección de memoria.

Utilizar un registro como base de los desplazamientos en memoria es una prática común en las arquitecturas de procesadores, ya que permite reducir la cantidad de bits necesarios para codificar el valor absoluto de una dirección de memoria dentro de una instrucción.

# Ciclo de Instrucción: FETCH OPERANDS

**FETCH OPERANDS**: Trae de memoria los datos requeridos para procesar la instrucción.

En está etapa se leen los operandos de memoria.
En arquitecturas con modos de direccionamiento más complejos, es la encargada de leer los datos.
Por ejemplo, en indirecto a registro, lee los datos para luego usarlos como operandos.

En LC-3, para la instrucción LD está etapa carga el registro MAR con la dirección absoluta y luego carga el contenido de MDR en el registro destino.

Para la instrucción ADD en cambio, se ocupa de obtener los registros del banco de registros.

En microarquitecturas más complejas, la etapa de obtener los operandos se realizar fuera de orden. Luego, a medida que se calculan los operandos, las instrucciones son ejecutadas.

# Ciclo de Instrucción: EXECUTE

**EXECUTE**: Consiste en la ejecución de la operación codificada por la instrucción.

En LC-3, para la instrucción ADD correspondería a la etapa en que los operandos son cargados en la ALU y el resultado de la operación es generado.

Dependiendo de la instrucción, está etapa **puede no ser necesaría**, ya que no todas las operaciones requieren la ALU o acceder a registros para hacer su operación.

Incluso, dependiendo del *datapath*, es posible que está etapa sea necesaria pero sin hace nada útil.

# Ciclo de Instrucción: STORE RESULT

**STORE RESULT**: Almacena los resultados de la operación en el lugar que fueron designados.

Esta última etapa suele no ser necesaria en microarquitecturas muy simples.

En LC-3 como en otras computadoras, la instrucción ADD puede cargar los operandos, realizar la operación y guardar el resultado en un solo ciclo.

Es importante reconocer está etapa en un ciclo de instrucción general, ya que se puede no disponer del recurso donde escribir el resultado de la operación.

# Cambios en la ejecución secuencial de instrucciones

Hasta ahora nuestra máquina ejecuta instrucciones de forma **secuencial**, una detrás de la otra.

El cálculo de la dirección de la nueva instrucción se realiza en la etapa de `Fetch`.

Las instrucciones de control se ocupan de **pisar el valor del PC** con un nuevo dato obtenido en la etapa de `Execute`.

Existen tres tipos de instrucciones de control:

- **Salto incondicional**: Cambia el PC remplazandolo por un valor calculado o estático.
  En LC-3, la instrucción: JMP (*jump*)
- **Salto condicional**: Cambia el PC si se cumple una determinada condición.
  En LC-3, la instrucción: BR (*conditional branch*)
- **Llamado o retorno de subrutina**: Cambia el PC y resguarda el PC remplazado. Luego puede retornar el control al PC que fue resguardado.
  En LC-3, las instrucciones: JSR(R), (*JumpSubRutine*), RET (*return*)

# Control del ciclo de instrucción

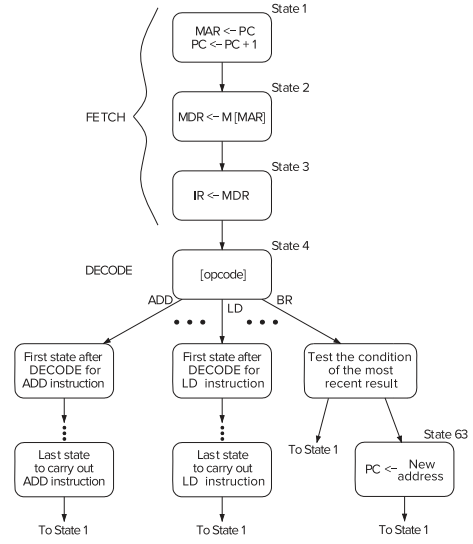
Es controlado por una **máquina de estados**.
Cada estado representa un conjunto de señales que se envían a los diferentes componentes del *datapath*.

Las flechas entre los diferentes estados representan las **transiciones** de un estado a otro. Estas pueden ser secuenciales o depender de alguna condición.

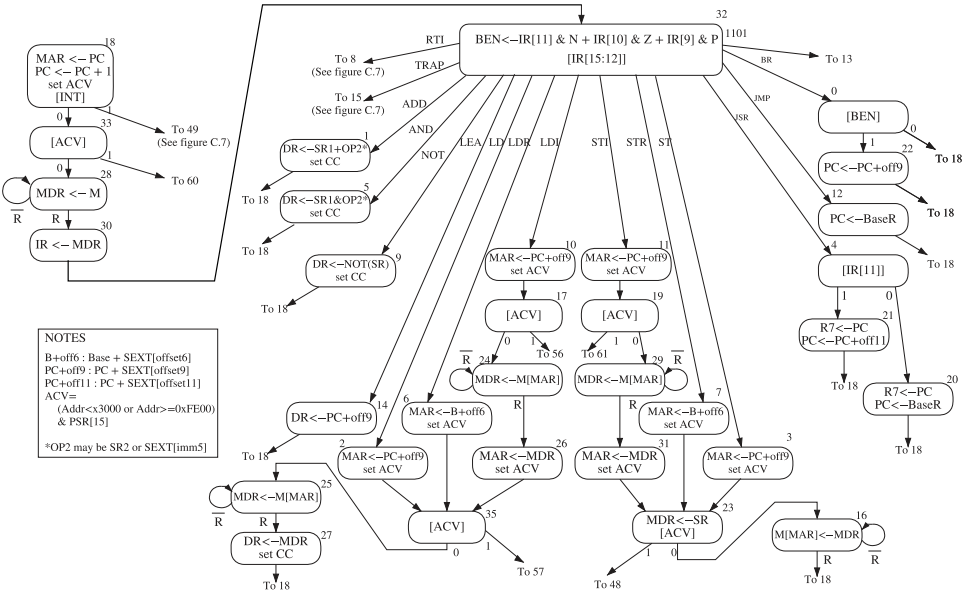
**La etapa de `Fetch`** corresponden a los primeros tres estados.

**La etapa de Decode** demora un ciclo representado por el estado 4, saltando al estado correspondiente dependiendo de la instrucción decodificada.

Esta máquina de estados puede ser implementada de diferentes formas. Sin embargo, la forma más simple, es codificar los estados en una memoria y accederlos secuencialmente. Adicionando lógica para saltar a estados según datos de control como el opcode.



# Control del ciclo de instrucción: Máquina de estados completa (Sin interrupciones)

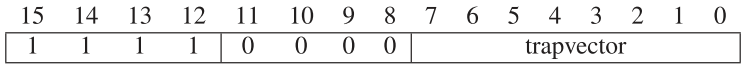


# Interrupciones (Instrucción TRAP)

Los procesadores poseen mecanismos para **detener el procesamiento** y tomar el control.

El soporte se puede prestar mediante una instrucción o independiente de las instrucciones.
Puede contar con soporte para un vector de interrupciones o una única interrupción.

En LC-3, se cuenta con la instrucción TRAP (opcode=1111) que realiza está tarea.




Los bits [7:0] (`trapvector`), codifican el vector de interrupciones.
Este valor se usa como índice en la **tabla de vectores de interrupción** para identificar la rutina.

# Conjunto de Instrucciones (*Instruction Set*)

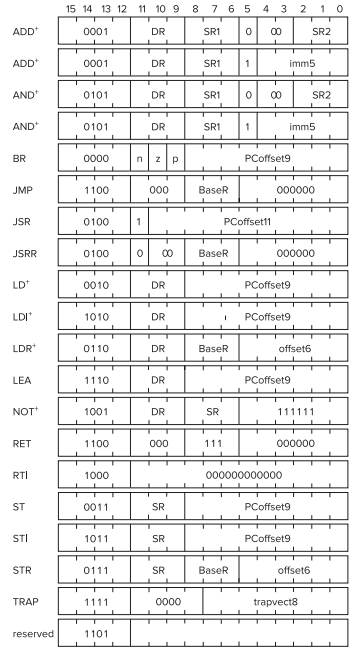
Los opcode permiten identificar las diferentes instrucciones.
Pueden estar determinados por un conjunto de bits fijo o variable.

Se busca que la codificación de instrucciones sea **regular**.
Los opcode y parámetros siempre en los mismos lugares e interpretados de la misma forma.
Esto simplifica la máquinaria de decodificación.

Puede ser que en algunos casos sea necesario pasar por varias **etapas de decodificación** para identificar las instrucciones.

En LC-3, su ISA cuenta con 15 instrucciones.
Cada una identificada por un **opcode único**.

Se identifican **diferentes tipos** de instrucciones, que dependen de como los parámetros fueron ordenados.



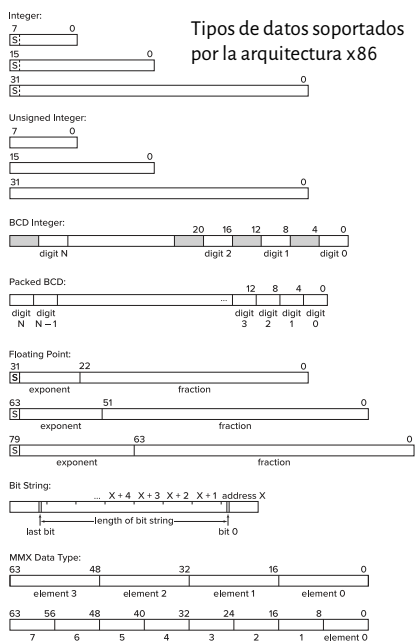
# Tipos de datos

Las diferentes instrucciones pueden soportar parámetros en distintos o en un único tipo de datos.

Internamente los distintos tipos de datos se resuelven con máquinaria diferente del procesador, lo que implica diferencias en la performance de las operaciones.

Los tipos de datos pueden ser tanto iterpretaciones de los bits de registros, como de memoria o incluso registros internos no accesibles por el programador.

En LC-3, solo se soportan datos enteros de tamaño fijo de 16 bits.



# Modos de direccionamiento

Es el mecanismo por el cual se especifica donde se encuentra un operando.

Generalmente los operandos pueden estar en tres lugares:

- En un registro.
- En una posición de memoria.
- En la instrucción como un literal.

En LC-3 se soportan 5 modos de direccionamiento:

- Inmediato o Literal
- Registro
- Relativo-PC
- Indirecto
- Base-Desplazamiento

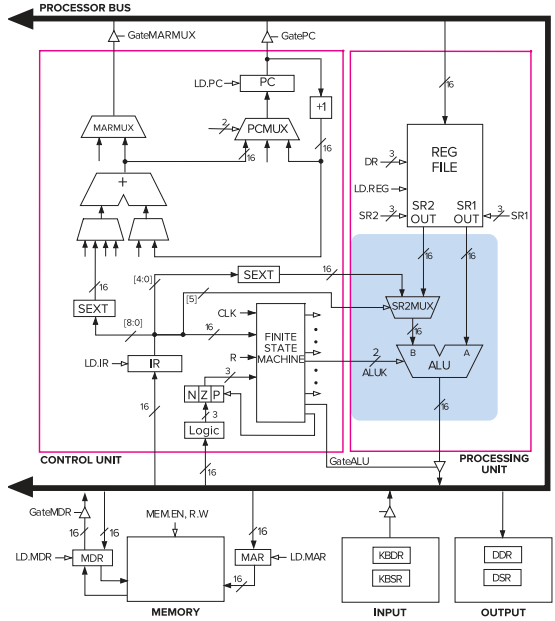
# Instrucciones de operación

Las operaciones aritméticas como ADD, SUB, NEG, MUL, DIV, o lógicas como AND, OR, NOT, XOR son ejemplos básicos de instrucciones.
Pueden operar con un solo operando como NEG o NOT, o con dos o más parámetros.

Resolver los parámetros se reliza a traves de diferentes modos de direccionamiento.

- **Inmediatos**: Decodificarlos de la instrucciones o usar constantes.
- **Registros**: Se debe poder identificarlos.
- **Memoria**: Generar u obtener la dirección, acceder al dato para escritura o lectura.

En LC-3, solo se cuenta con tres instrucciones de este tipo: ADD, AND, y NOT.



# Instrucciones de operación

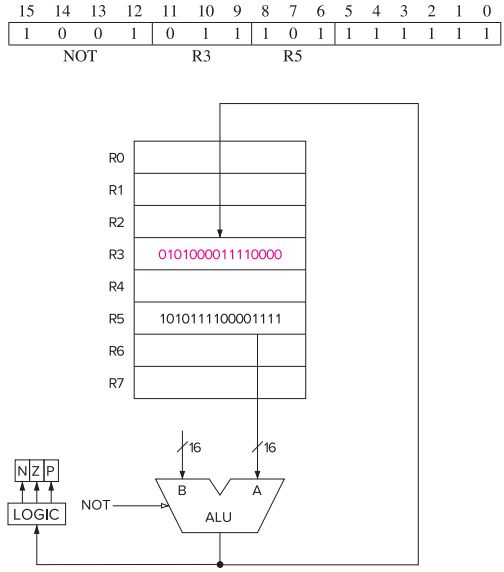
La instrucción NOT (opcode = 1001) opera con dos parámetros, un registro fuente y otro destino.

Utilizamos los bits [8:6] para especificar el registro fuente y los bits [11:9] para el registro destino.

Por convención los bits [5:0] deben contener todos un 1.

En el ejemplo se realiza un NOT de R5 y se almacena el resultado en el registro R3. Además los flags son alterado según el resultado de la operación.

Notar que ambos parárametros pueden ser el mismo registro.



# Instrucciones de operación

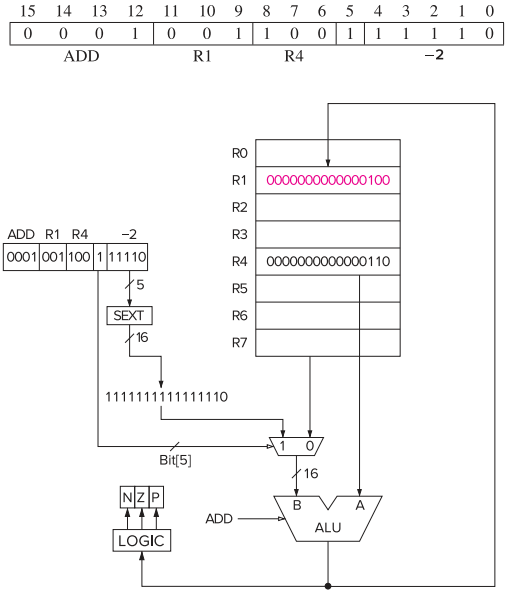
En LC-3, las instrucciones ADD (opcode = 0001) y AND (opcode = 0101) realizan operaciones binarias, que requieren como entrada dos datos de 16 bits.

Tiene dos sabores, donde uno de sus parámetros puede ser es un valor inmediato o un registro.

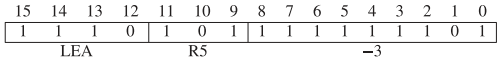
Los bits [11:9] especifican el registro destino y los bits [8:6] especifican uno de los parámetros.

El segundo parámetro depende del bit [5].

- `bit [5] = 0`: Los bits [2:0] indican el registro sobre el que operar.
- `bit [5] = 1`: Los bits [4:0] indican un inmediato que se toma en complemento a dos y se opera con el signo extendido.

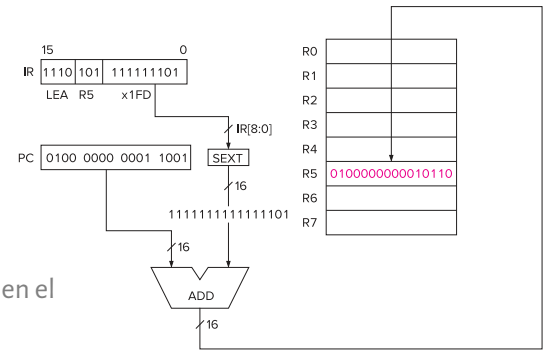


# Instrucción LEA (*Load Effective Address*)




En arquitecturas donde las direcciones a memoria son **relativas**, está instrucción permite obtener la dirección de memoria absoluta a utilizar.

En LC-3, se utiliza para calcular una dirección de memoria usando como **base el PC** y un valor **inmediato de desplazamiento**.




Esta instrucción utiliza la máquinaria de calculo de direcciones, pero su resultado no se almacena en el registro MAR, sino que se carga **directamente en el banco de registros**.

LEA (opcode = 1110) carga en el registro indicado por los bits [11:9]
el valor del PC sumado a los bits [8:0] tomados con el signo extendido.

# Instrucciones de movimiento de datos

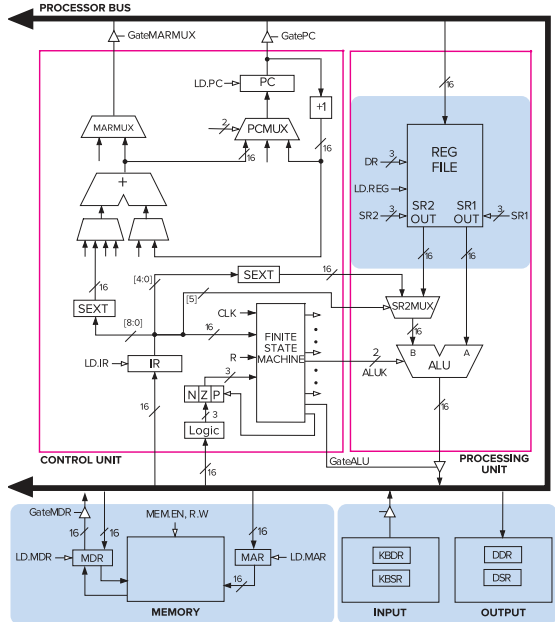
Estas instrucciones **mueven o copian información** entre registros de propósito general, memoria o registros de entrada/salida.

En general se identifican dos parámetros, **el fuente**, que se mantiene igual luego de la lecutra y **el destino** que se altera, pisando el contenido que previamente tenia.

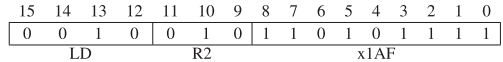
En LC-3 existen seis instrucciones de este tipo: LD, LDR, LDI, ST, STR, y STI.

- **LD*** (*Load*): Carga un dato desde memoria.
- **ST*** (*Store*): Guarda un dato a memoria.

Todas operan usando una dirección de memoria y un registro, ya sea fuente o destino según el caso.



# Instrucciones de movimiento de datos




Las instrucciones LD (opcode = 0010) y ST (opcode = 0011) usan el tipo de direccionamiento relativo al PC.

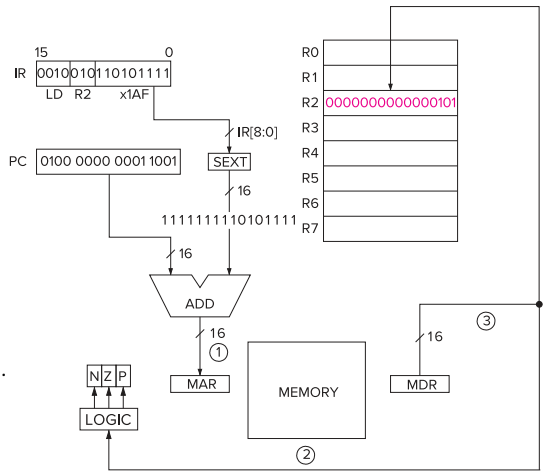
Los bits [8:0] indican el **desplazamiento** desde el PC. Mientras que los bits [11:9] especifican el **registro**.

Notar que los *flags* también son alterados cuando se lee o escribe un dato en memoria. Además el desplazamiento está limitado entre -255 y +256.

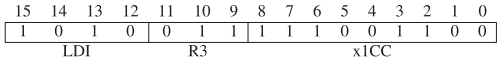
La operación requiere tres pasos:

1. Suma el PC con el IR [8:0] con signo extendido, y el resultado es cargado en MAR.
2. La memoria es leída y el cotendio cargado en MDR.
3. El valor luego es cargado en el registro y los *flags* alterados según el dato.

Para el caso de ST el comportamiento es simétrico.



# Instrucciones de movimiento de datos




Las instrucciones LDI (opcode = 1010) y STI (opcode = 1011) usan el tipo de direccionamiento indirecto a memoria.

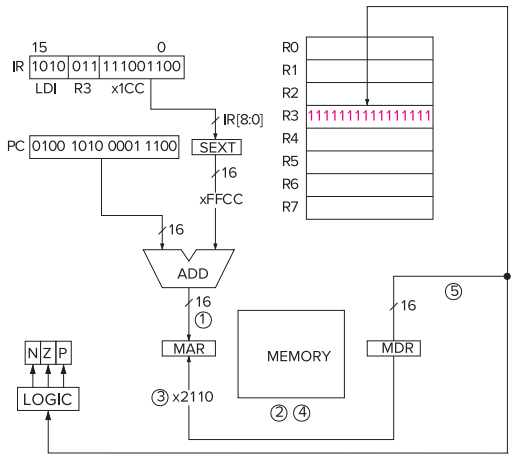
Los bits [8:0] indican el **desplazamiento** desde el PC, para llegar a la dirección que se utilizará como puntero. Mientras que los bits [11:9] especifican el **registro**.

- `LDI Rx, VALUE ;` Rx ← mem[mem[VALUE]]
- `STI Rx, VALUE ;` mem[mem[VALUE]] ← Rx

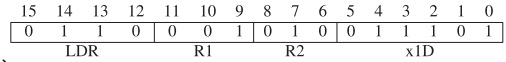
La operación requiere cinco pasos:

1. Suma el PC con el IR [8:0] con signo extendido, y el resultado es cargado en MAR.
2. Lee de memoria y carga el resultado en MDR.
3. Carga MAR con el valor de MDR.
4. Lee de memoria y carga el resultado en MDR.
5. Mueve el contenido de MDR al registro destino y actualiza los *flags*.

Para el caso de STI el comportamiento es simétrico.



# Instrucciones de movimiento de datos




Las instrucciones LDR (opcode = 1010) y STR (opcode = 1011) usan el tipo de direccionamiento base-desplazamiento.

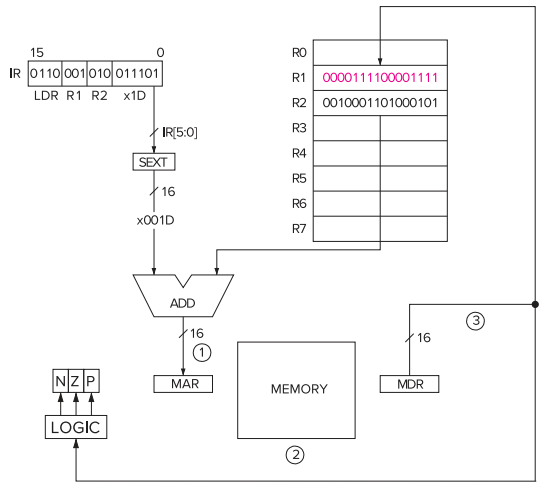
Los bits [8:6] indican el registro que será usado como puntero y los bits [5:0] el **desplazamiento**.
Los bits [11:9] especifican el **registro** destino.

- `LDR Rx, Ry, OFFSET;` Rx ← mem[Ry + OFFSET]
- `STR Rx, Ry, OFFSET;` mem[Ry + OFFSET] ← Rx

La operación requiere tres pasos:

1. Suma el registro indicado por IR [8:6] con el IR [5:0] con signo extendido, y el resultado es cargado en MAR.
2. Lee de memoria y carga el resultado en MDR.
3. Carga el registro destino con el valor de MDR.

Para el caso de STR el comportamiento es simétrico.



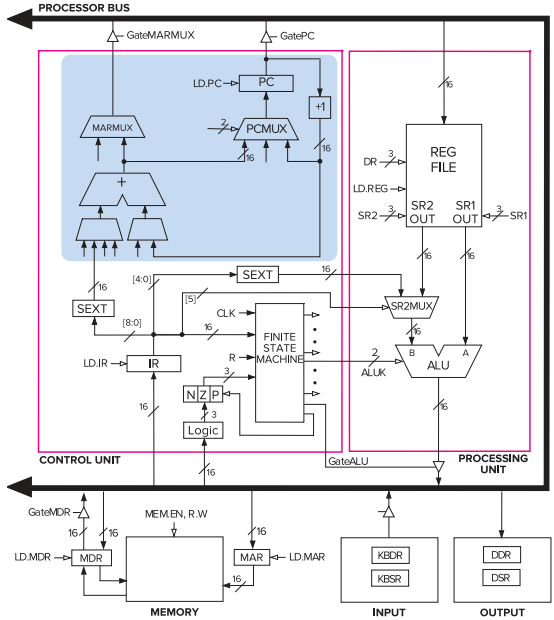
# Instrucciones de control

El procesador ejecuta instrucciones una a una, las instrucciones de control permiten alterar está operación secuencial.

Existen principalmente tres formas de alterar el control, de forma **incondicional**, de forma **condicional** o por medio de una **interrupción**.

En LC-3 tiene seis instrucciones de este tipo:

- JMP (*Unconditional Jump*)
- BR (*Conditional Branch*)
- JSR, JSRR (*Subroutine Call*)
- RET (*Return from Subroutine*)
- TRAP (*Trap or Interrupt*)
- RTI (*Return from Trap or Interrupt*)



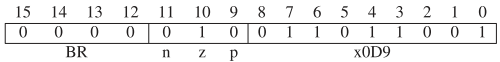
# Instrucciones de Saltos Condicionales

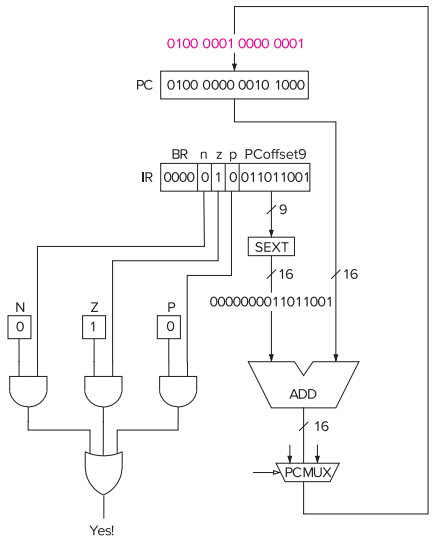
Estas instrucciones toman la decisión de cambiar o no el PC en tiempo de ejecución.

Esta decisión puede ser en base a un valor de un *flag* específico, a una condición de un conjunto de *flags*, o incluso a un valor específico de un registro.

En LC-3, la instrucción BR (opcode = 0000) contiene en los bits [11], [10] y [9] los códigos de condición de forma explicita.
Si alguno de los bits simultáneamente con los *flag* están en 1, entonces se realiza el salto.

Se utiliza como destino la dirección calculada como relativa al PC, con el desplazamiento dado por los bits [8:0] extendiendo el signo.



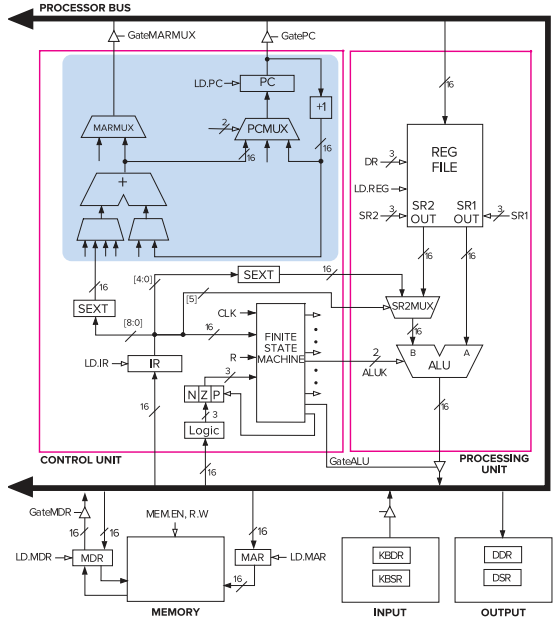
# Instrucciones de Saltos Incodicionales

Las instrucciones de saltos incondicionales se diferencian en la forma en que se calcula la dirección destino.

En LC-3 se tiene instrucción JMP (opcode = 1100).

Utilza los bits [8:6] para indicar el **registro** que contiene la direccion de la próxima instrucción, moviendo su contenido al PC.

Está instrucción permite hacer un salto sin la limitación de desplazamientos en los saltos incondicionales.

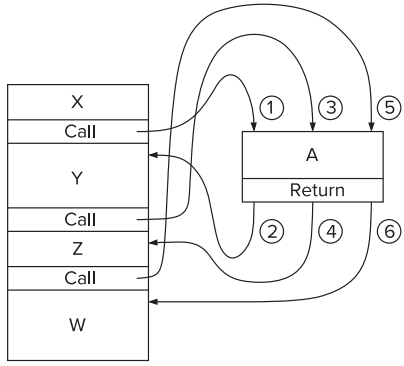


# Instrucciones de *Call* y *Return*

Las instruciones para llamar y retornar de una subrutina utilizan un lugar donde **almacenar la dirección previa del PC**.

Este lugar puede ser la pila (*stack*) o puede ser un registro específico.

La implementación, puede ser soportada por completo por una instrucción o requerir de más instrucciones.




En LC-3, el *call* es implementado de forma similar a un JMP, con la diferencia que deja en un registro específico el PC anterior.

El *return* es implementado por la instruccción JMP, usando un registro específico como fuente.

# Instrucciones de *Call* y *Return*

En LC-3 se proveen dos formatos para llamar a subrutinas.

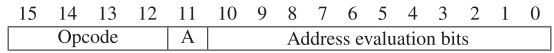
- **JSR**: Direccionamiento relativo al PC.
- **JSRR**: Direccionamiento por registro.

En ambos formatos, una vez calculada la dirección destino, el valor previo de PC o **dirección de retorno se almacena en el registro R7**.

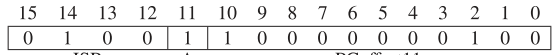
La instrucción **RET** es la misma instrucción que JMP. Se codifica seteando de forma fija el registro R7 como destino del salto.

Registro donde queda la dirección de retorno.

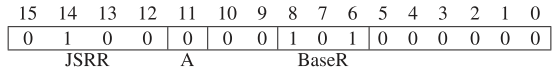
Formato instrucción *Call*



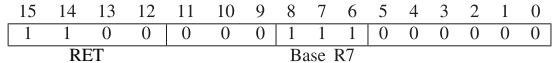

Ejemplo Instrucción JSR




Ejemplo Instrucción JSRR



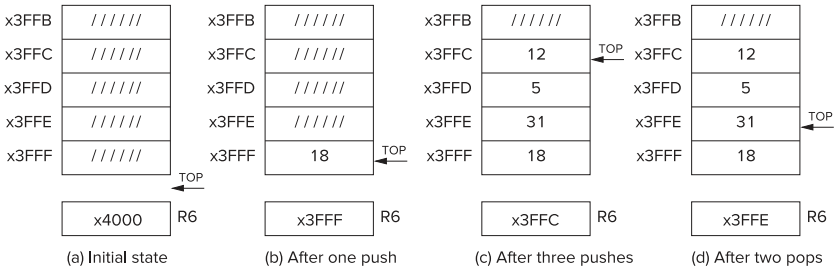

Instrucción RET



# Pila (*stack*)

La forma más común de implementar una pila es en memoria. El *stack* corresponde a una secuencia de posiciones de memoria sobre las que se lleva registro de la primer posición.

Se utilizan dos mecanismos, uno para **cargar datos** (*push*) y otro para **retirar datos** (*pop*)




En LC-3, por convención se utiliza el registro R6 como puntero al tope de la pila.
Este apunta a la primera posición ocupada de la pila.

# Mecanismo de *Push* y *Pop*

Para realizar un **PUSH** se deben ejecutar las siguientes instrucciones:

```
ADD R6, R6, #-1
STR Rx, R6, #0
```

Las dos instruccciones realizan:

1. Decrementa el puntero al tope de la pila, apuntando a la nueva posición de memoria que debe ser escrita.
2. Escribe en memoria el registro Rx, en la posición de memoria apuntada por el tope de la pila.

Para realizar un **POP** se deben ejecutar las siguientes instrucciones:

```
LDR Rx, R6, #0
ADD R6, R6, #1
```

Las dos instruccciones realizan:

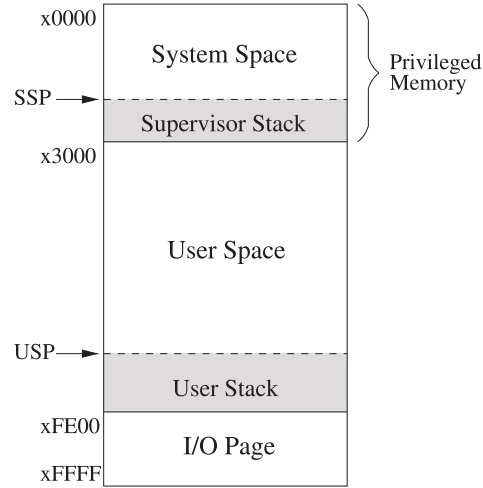
1. Lee de memoria el dato apuntado por el tope de la pila y lo escribe en el registro Rx.
2. Incrementa el puntero al tope de la pila, apuntando a la última posición de memoria ocupada en la pila.

# Organización de la memoria

En LC-3 tiene un espacio de direccionamiento de 16 bits (`0x0000` a `0xFFFF`).

Este espacio está dividido de la siguiente forma:

- `0x0000` a `0x2FFF` - *System space*
  Memoria privilegiada. Espacio para las estructuras y código del sistema operativo.
- `0x3000` a `0xFDFF` - *User space*
  Memoria sin privilegios. Espacio para todos los programas de usuario.
- `0xFE00` a `0xFFFF` - *I/O space*
  Memoria pivilegiada. Espacio donde se mapean los registros de entrada salida.

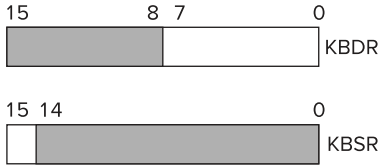



Notar que existen dos espacios de pila independientes, para supervisor y usuario.

# Entrada/Salida - Dispositivo de entrada (*Keyboard*)

Para administrar los caracteres que llega desde el teclado se utilizan **dos registros**.

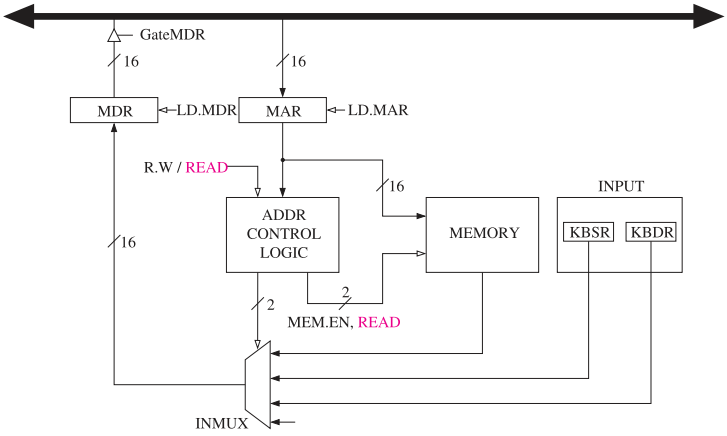
- `0xFE02` - **KBDR** (*keyboard Data Register*)
  Mantiene en los 8 bits menos significativos el caracter que fue presionado.
- `0xFE00` - **KBSR** (*keyboard Status Register*)
  Utiliza solo el bit más significativo como señal de sincronización.




Si KBSR[15] = 1, el código ASCII en KBDR[7:0] es válido y aún no fue leído.
En este estado el teclado no puede enviar una nueva tecla.

Si KBSR[15] = 0, el código ASCII en KBDR[7:0] es invalido.
En este estado el teclado puede escribir la nueva tecla y cambiar el estado.

# Entrada/Salida - Entrada mapeada a memoria



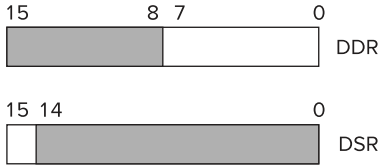

En este caso, cuando se carga el registro MAR se identifica si la dirección es de un registro de entrada/salida.

En vez de leer de memoria, la lógica de control se encarga de **seleccionar el registro que se busca leer**, guardando su valor en el registro MDR.

# Entrada/Salida - Dispositivo de salida (*Monitor*)

Para administrar los caracteres a imprimir en pantalla se utilizan **dos registros**.

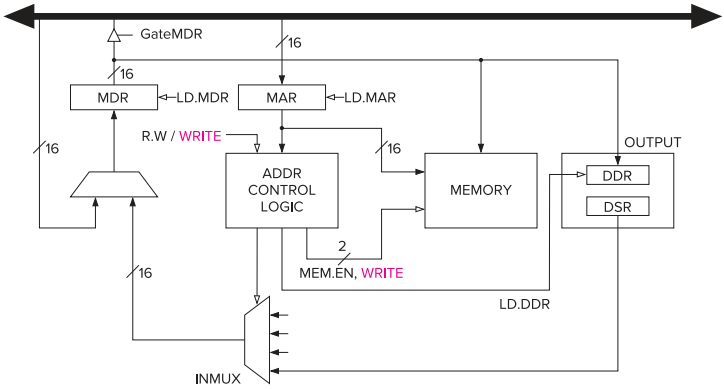
- `0xFE06` - **DDR** (*Display Data Register*)
  Mantiene en los 8 bits menos significativos el caracter que quiere ser escrito en pantalla.
- `0xFE04` - **DSR** (*Display Status Register*)
  Utiliza solo el bit más significativo como señal de sincronización.




Si KBSR[15] = 1, el código ASCII en KBDR[7:0] fue leído por el monitor.
Ahora se puede escribir un nuevo caracter y setear a cero la señal de sincronización.

Si KBSR[15] = 0, el código ASCII en KBDR[7:0] está a la espera de ser leído por el monitor.
En este estado, se debe esperar a que el monitor lea para escribir el nuevo caracter.

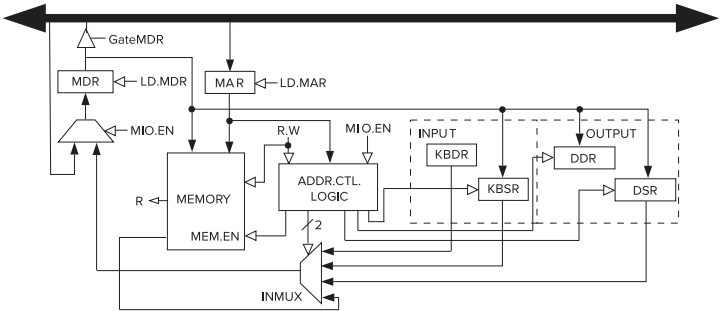
# Entrada/Salida - Salida mapeada a memoria




En este caso, cuando se carga el registro MAR se identifica si la dirección es de un registro de entrada/salida.

En vez de escribir en memoria, la lógica de control se encarga de **activar la señal para escribir el registro**, tomando el valor del registro MDR.

# Entrada/Salida - Mapeada a memoria




Los mecanismos de mapeo a memoria son similiares para lectura y escritura.

Mapear a memoria implica lógica que decide que direcciones corresponden a que almacenamiento.

Este mecanismo se complementa con **interrupciones**, que permite despertar rutinas para resolver una acción de entrada/salida específica.

La solución para administrar un espacio de entrada/salida independiente, es similiar a tener otra memoria accedida por instrucciones diferentes.

# Resumen

Definir un ISA implica minimamete,

- Definir la **interfaz** entre el software y el hardware del procesador.
- Definir sus **instrucciones**, su comportamiento y codificación.
- Definir los tamaños y propiedades de la **memoria**.
- Definir como se organizará la memoria y administrará la **pila**.
- Definir los modos de **direccionamiento** y su alcance dependiendo de cada instrucción.
- Definir los mecanismos de **protección** con los que contará el procesador.
- Definir el soporte para **entrada/salida**, administración de dispoositivos e **interrupciones**.

# Bibliografía


- Yale N. Patt, Sanjay J. Patel
  **“Introduction to Computing Systems”** - Third Edition - McGraw-Hill
  - Chapter 4 - The von Neumann Model → Pag. 121-137
  - Chapter 5 - The LC-3 → Pag. 145-177
  - Chapter 8 - Data Structures → Pag. 263-268
  - Chapter 9 - I/O → Pag. 313-327

¡Gracias!