Ignacio Méndez Pérez
2023

Resumen Arquitectura de Computadores

INDICE

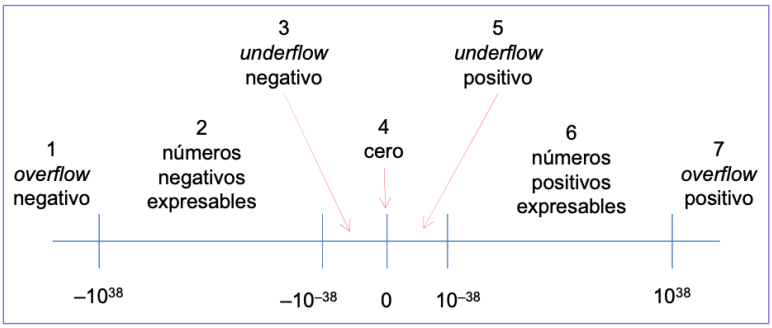
Ignacio Méndez Pérez
2023

Representación de números punto flotante

| 1 | 10000001 | 01000000000000000000000 |

$$(-1)^{signo} \times (1+fracción) \times 2^{exponente-127}$$

signo	*exponente*	*fracción*
un bit	8 bits	23 bits



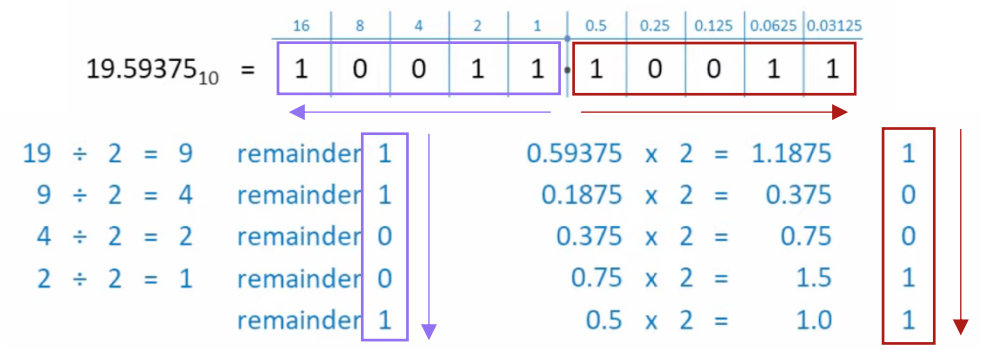
El **exponente** se almacena desfasado en 127 es decir en vez de almacenar un bit de signo aparte, o representar el número en complemento a 2, se desfasa el número de manera de tener sólo valores positivos. La ventaja de esto está en hacer más simple la aritmética.

La fracción corresponderá al número que viene después de la primera coma. Es decir si el número es 1.23049 entonces los bits de "fracción" representarán el número 0.23049.

Ejemplo: Representemos el número: $\mathbf{19.59375}$.

1. Determinamos el bit del signo: 0 (es positivo).

2. Convertimos el número a binario.



3. Normalizamos para que solo nos quede 1 número a la izquierda del punto:

4. Nuestro exponente será entonces $4 \Longrightarrow 4 + 127 = 131_{10} = 10000011_2$

5. Nuestros bits de fracción serán los que calculamos en el paso 3 sin el primer dígito:

$$001110011$$

Finalmente, tendremos que la representación queda:

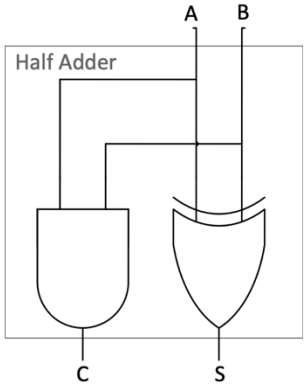
$$\boxed{19.59375_{10} = 0 \mid 10000011 \mid 001110011_2}$$

Ignacio Méndez Pérez
2023

Suma en circuitos

- Sumador de 1 bit (*half adder*):

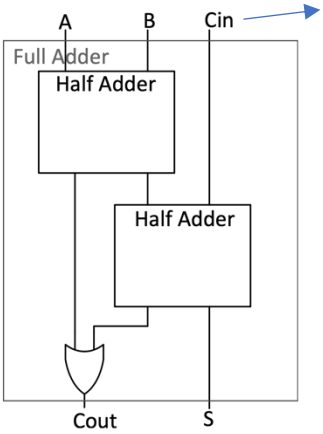
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



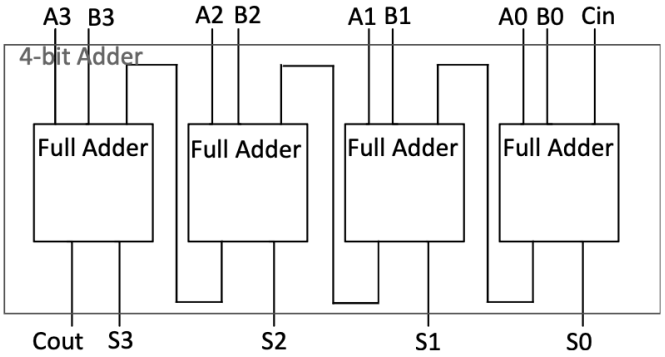
- ¿Cómo lo hacemos para sumar 4 bits?

El siguiente circuito es un sumador de 1 bit, pero con acarreo. Es decir,

Me indica si hay algún "acarreo" de alguna suma previa.



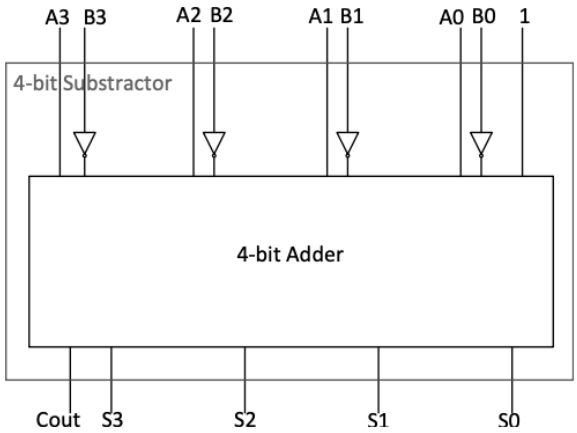
Sumador de 1 bit (*full adder*)



Para sumar 4 bits simplemente concatenamos 4 *full adders*.

Restador de 4 bits

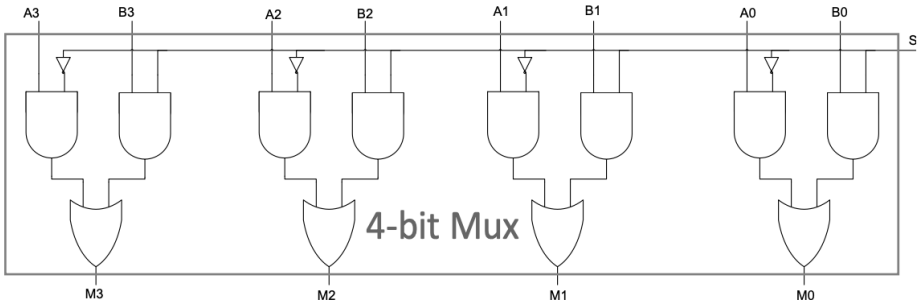
Si invertimos los bits, y sumamos 1, podemos reutilizar el *4-bit adder* para realizar la resta:



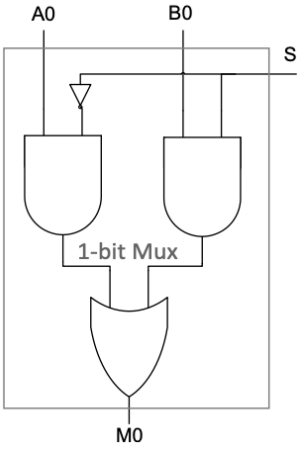
Ignacio Méndez Pérez
2023

Multiplexor

Permite seleccionar una de las múltiples entradas y transmitirla a la salida. En el ejemplo, S me indica se quiero quedarme con $A0$ o con $B0$.



Multiplexor de 8 entradas de 4 bit de datos

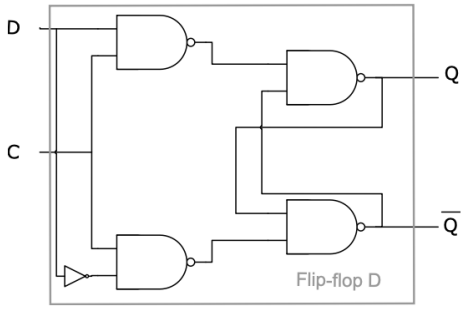


Multiplexor de 2 entradas de 1 bit de dato

Con esto, puedo combinar los circuitos de operaciones (suma, resta, etc.) y así solo con los bits S decidir qué operación realizar con los datos.

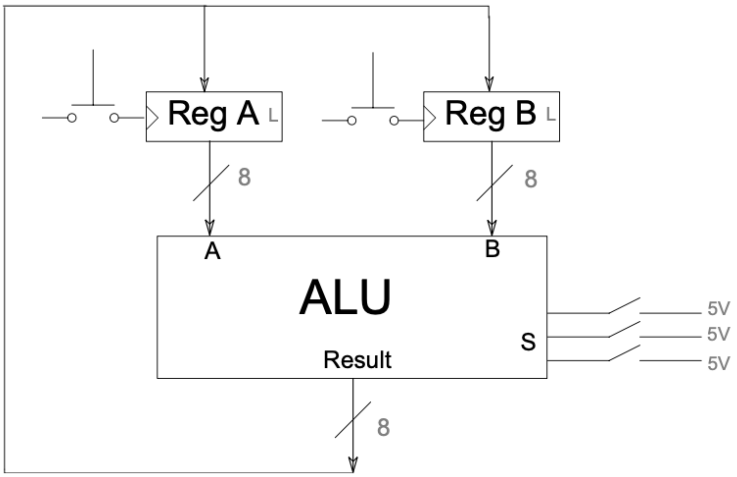
Flip-flops

Si en la entrada de datos D llega un determinado valor y además la señal de control C está activa, el valor de salida Q de *flip-flop* se actualiza con el valor de D. Si en cambio, la señal del control C está desactiva, el valor de Q es el mismo que tenía previamente, es decir, el *flip-flop* puede **guardar información**!

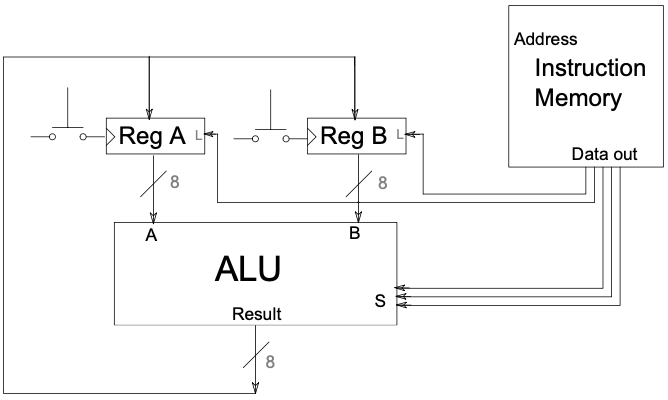


C	D	Q
0	x	Q
1	0	0
1	1	1

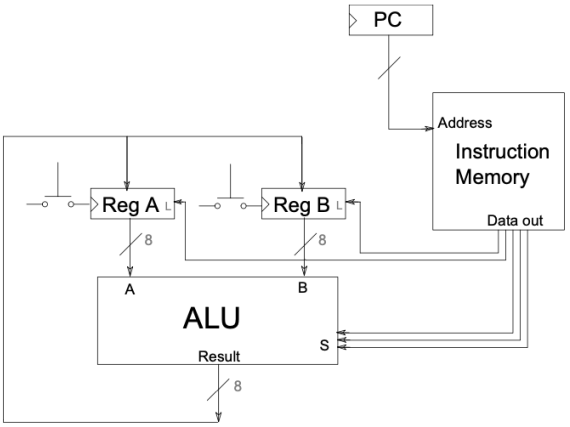
ALU



la	lb	s2	s1	s0	operación
1	0	0	0	0	A=A+B
0	1	0	0	0	B=A+B
1	0	0	0	1	A=A-B
0	1	0	0	1	B=A-B
1	0	0	1	0	A=A and B
0	1	0	1	0	B=A and B
1	0	0	1	1	A=A or B
0	1	0	1	1	B=A or B
1	0	1	0	0	A=not A
0	1	1	0	0	B=not A
1	0	1	0	1	A=A xor B
0	1	1	0	1	B=A xor B
1	0	1	1	0	A=shift left A
0	1	1	1	0	B=shift left A
1	0	1	1	1	A=shift right A
0	1	1	1	1	B=shift right A



Almacenamiento de instrucciones en *instruction memory*

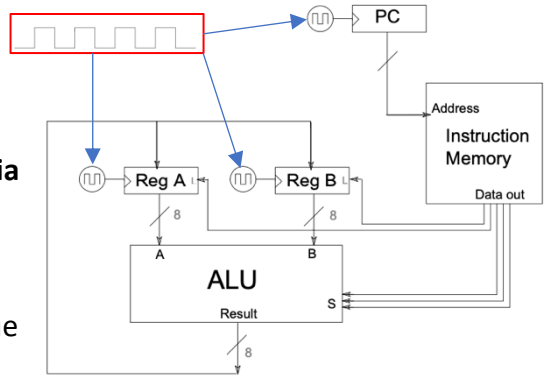


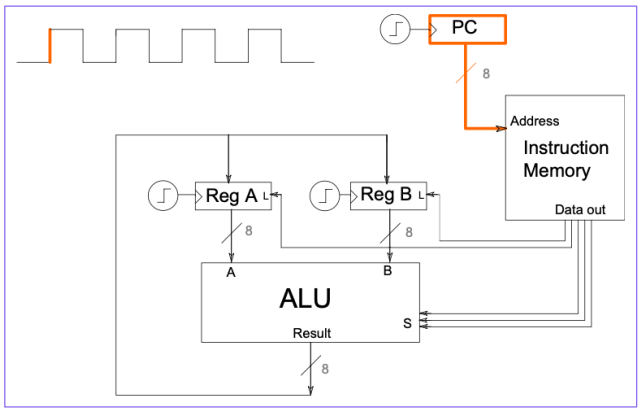
El valor del *program counter* (PC) define la dirección de la memoria de instrucciones, y por lo tanto la operación a realizar.

Clocks

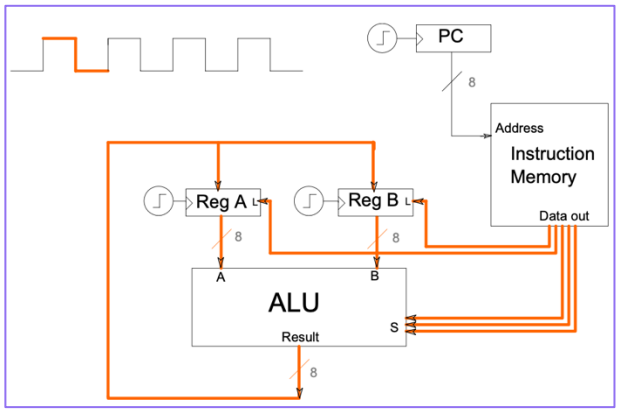
Osciladores que generan alternadamente pulso de voltaje alto (equivalentes a un 1 lógico) y pulsos de voltaje bajo (equivalentes a un 0 lógico) a una **frecuencia constante**.

Para tener sincronizadas las operaciones se utilizará un **único *clock*** que se conecta a todos los componentes que lo requieren.





Program Counter activado por flanco de subida

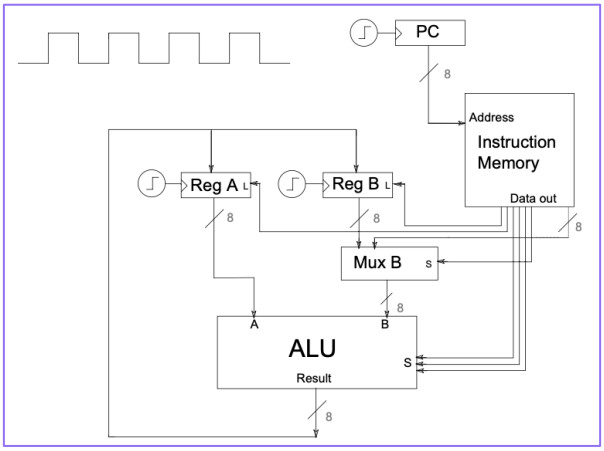


Cuando el *clock* subió y luego cuando baja, el *program counter* y los registros no se actualizan, dejando tiempo para procesar la instrucción y ejecutarla

Operaciones con literales

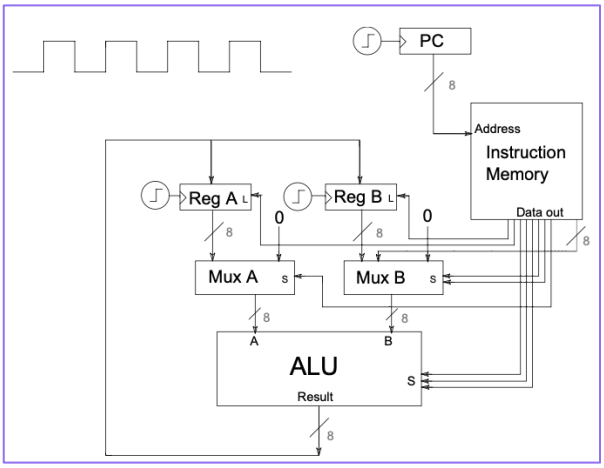
- El valor del literal es entregado de forma explícita, y se incluye como parte de la **instrucción**.

- Se agrega un multiplexor que permite elegir entre el registro B y el literal entregado.

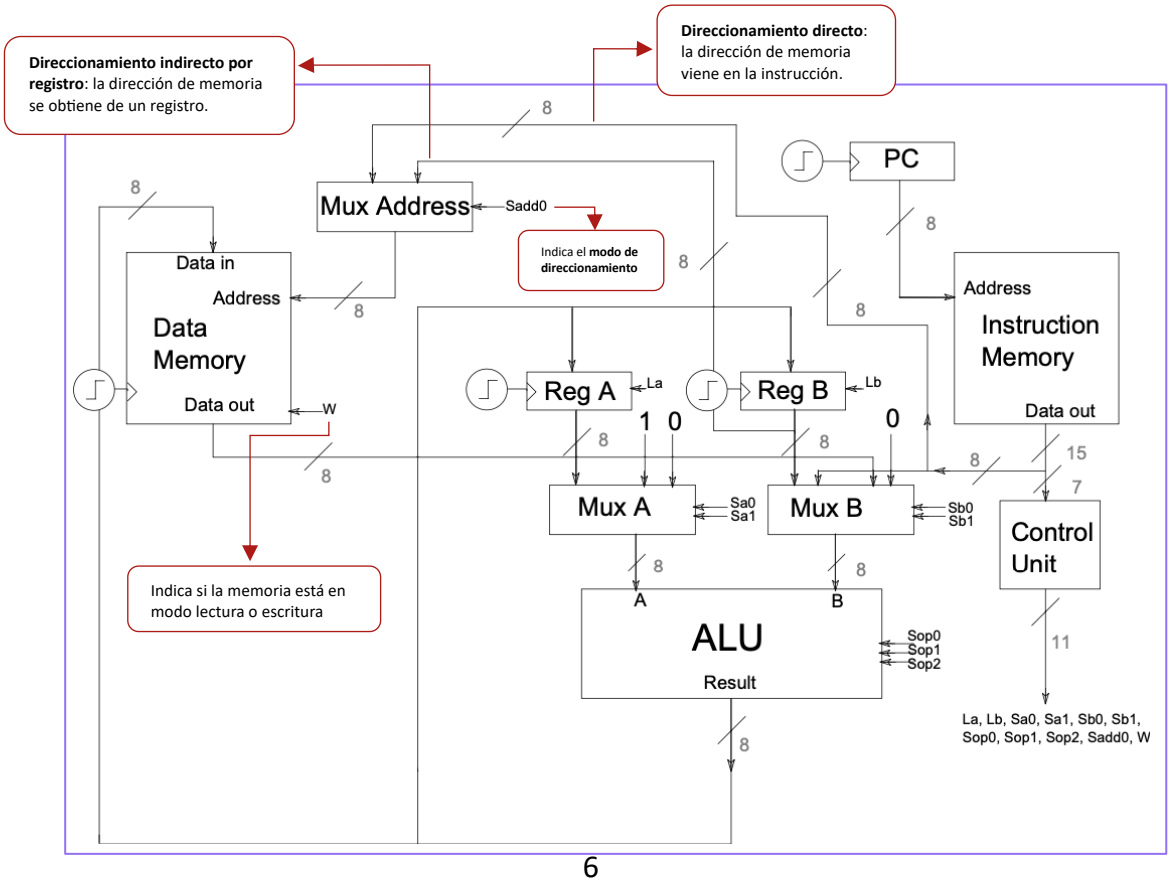


- Si quiero reemplazar algún registro por el literal, podemos hacer la operación de $A = 0 + literal$.

- Para permitir realizar estas sumas con 0 debemos agregar un nuevo multiplexor, esta vez al registro A, que permita elegir entre el resultado del registro y el valor 0. Adicionalmente, al multiplexor del registro B le agregamos una entrada que permite también elegir el valor 0.



- Una memoria de datos permite almacenar y agregar datos que pueden ser variables durante el transcurso del programa.



Salto incondicional

Se debe tener la capacidad de modificar la instrucción que se va a ejecutar. Para esto, es necesario agregar una nueva señal del control Lpc que indicará si el *program counter* está en modo carga $Lpc = 1$ o en modo incremento $Lpc = 0$.

Salto condicional

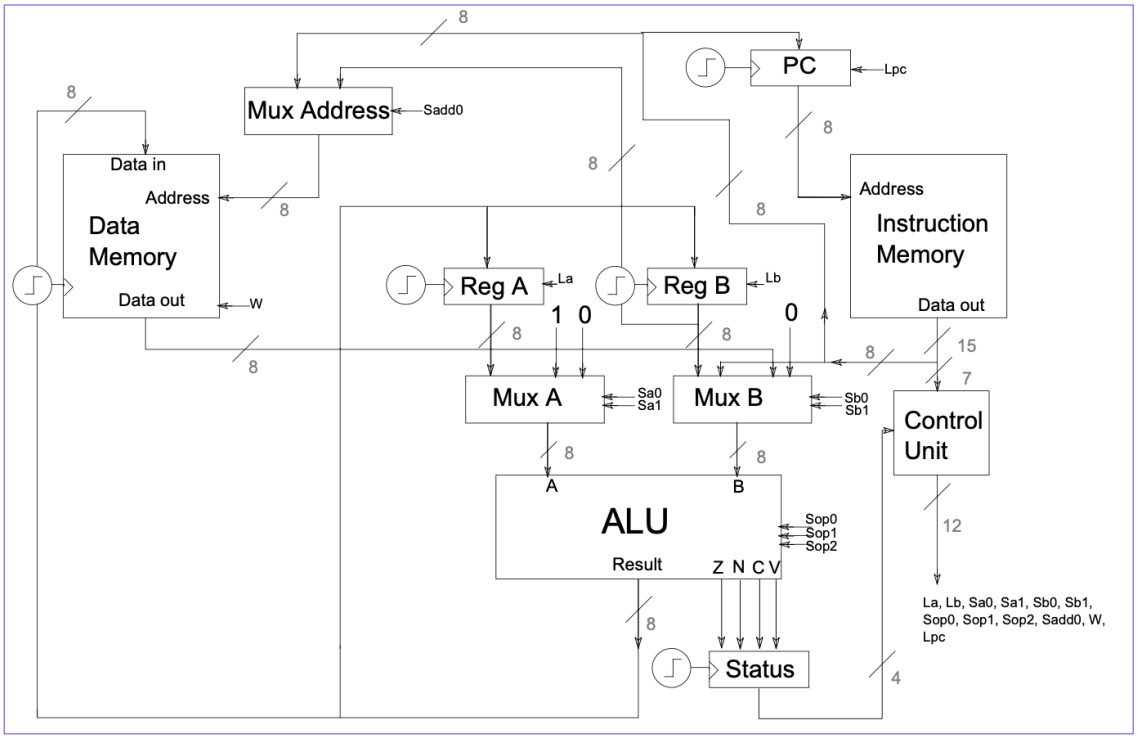
Podemos agregar la instrucción **`CMP A,B`** que ejecuta la resta entre los registros A y B y **no** almacena el resultado. El siguiente paso luego de ejecutar la comparación (CMP) es identificar si el resultado fue 0 o no.

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

- Zero (***Z***): El código de condición cero (*Zero*) se obtiene a partir del resultado de la ALU, haciendo un **`or`** entre todos los bits y luego un **`not`** de la salida del **`or`**. De esta forma, sólo se obtendrá una salida final de 1 cuando todos los bits del resultado de la ALU sean 0.
- Negative (***N***): El código de condición negativo se puede obtener de manera simple, tomando el bit más significativo del resultado. Dado que las operaciones se realizan en complemento a 2, si el bit más significativo es 1, será un número negativo, si es 0, positivo.
- *Carry* (***C***): El bit de *carry* es simplemente la salida «*carry out*» del sumador restador, ya que solo con estas operaciones puede ocurrir un *carry* (Se suma más de lo que se puede sumar).
- *Overflow* (**V**): Cuando a un número positivo se le suma algo y queda negativo. El bit de *overflow* es más complejo que el *carry*, ya que ocurre en distintas circunstancias dependiendo del signo de las entradas, la operación y de la salida. A continuación se presentan los casos posibles para que ocurra un *overflow*:

Operación	A	B	Resultado	Ejemplo (1 byte)
$A+B$	≥ 0	≥ 0	< 0	$127+4=-125$
$A+B$	< 0	< 0	≥ 0	$-127+-4=125$
$A-B$	≥ 0	< 0	< 0	$127--4=-125$
$A-B$	< 0	≥ 0	≥ 0	$-127-4=125$

Para implementar esto, es necesario agregar un circuito combinacional a la ALU que dependiendo de las entradas, operación y salidas entregue un 1 o un 0 para indicar el overflow.



Subrutinas (“funciones”)

Existen 3 elementos necesarios para poder implementar una subrutina: poder entregarle **parámetros**, poder recibir un valor de **retorno** y poder hacer la **llamada** a la subrutina.

- **Parámetros**

  Para poder entregarle parámetros a la subrutina, es necesario almacenarlos en algún lugar al cual está pueda acceder. En el computador básico existen dos lugares donde se pueden almacenar datos: registros y variables en memoria.

  1. **Registros**: una primera opción para pasar parámetros es almacenarlos en los registros y que luego la subrutina se encargue de obtenerlos. Para lograr esto, se debe saber a priori que registros se ocuparán para que parámetros, de manera de cargar los que efectivamente ocupará la subrutina. Aunque está implementación es simple, la principal desventaja está en el número limitado de registros, que impide entregar más parámetros que los registros disponibles.

  2. **Variables**: otra opción para el paso de parámetros es almacenarlos en memoria y que la subrutina los obtenga de ahí. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina. La ventaja respecto a los registros están en la mayor disponibilidad de espacio.

- **Valor de retorno**

  Al igual que con los parámetros, el valor de retorno debe ser almacenado en algún lugar que pueda ser accedido tanto por la subrutina.

  1. **Registros**: Al igual que con los parámetros, para ocupar registros como valor de retorno hay que saber a priori que registro ocupará la subrutina. La diferencia con los parámetros es que como el retorno es sólo un valor, es factible realizar esto, a pesar de tener pocos registros.

  2. **Variables**: Otra opción para el retorno es ocupar una variable en memoria. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina.

- **Llamada y dirección de retorno**

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  MOV A, (vector1)
                  MOV (var1), A
                  MOV A, (vector2)
                  MOV (var2), A
                  //Llamar a subrutina y retornar
                  MOV A, (prodPunto)
                  ADD A, (res)
                  MOV (prodPunto), A

                  MOV B, vector1
                  INC B
                  MOV A, (B)
                  MOV (var1), A
                  MOV B, vector2
                  INC B
                  MOV A, (B)
                  MOV (var2), A
                  //Llamar a subrutina y retornar
                  MOV A, (prodPunto)
                  ADD A, (res)
                  MOV (prodPunto), A

                  JMP end

      mult:
      start:      MOV  A,(res)
                  ADD  A,(var2)
                  MOV  (res),A
                  MOV  A,(i)
                  ADD  A,1
                  MOV  (i),A
                  MOV  B,(var1)
                  CMP  A,B
                  JLT  start

      end:
```

  Debemos de alguna forma pasar de la ejecución del código principal a la ejecución del código de esta.

  El primer paso para poder acceder al código de la subrutina es que se encuentre en la memoria de instrucciones. En el ejemplo anterior se observa que el código de la subrutina se incluyó al final, y se agregó un *label* `mult` para indicar el inicio de la subrutina. Llamar a la subrutina, entonces, se puede hacer simplemente saltando al *label* correspondiente con una instrucción JMP `mult`. El problema está en el retorno: al finalizar la ejecución de la subrutina, ¿cómo se sabe a dónde volver? No es posible agregar otro `JMP`, ya que no se puede saber a qué dirección se va a retornar (depende de donde se llamó a la subrutina). Para poder realizar correctamente el retorno es necesario almacenar el valor del *program counter* antes de realizar la llamada, para luego volver a cargarlo luego de retornar de la subrutina. En realidad, lo que nos interesa es guardar el valor del program counter incrementado en 1, dado que al volver de la subrutina queremos ejecutar la siguiente instrucción, no la actual.

  Para lograr la llamada a la subrutina, entonces, debemos agregar el soporte de *hardware* necesario que permita almacenar el valor del *program counter* incrementado en 1 en memoria (al hacer la llamada) y para poder cargarlo desde memoria (al retornar).

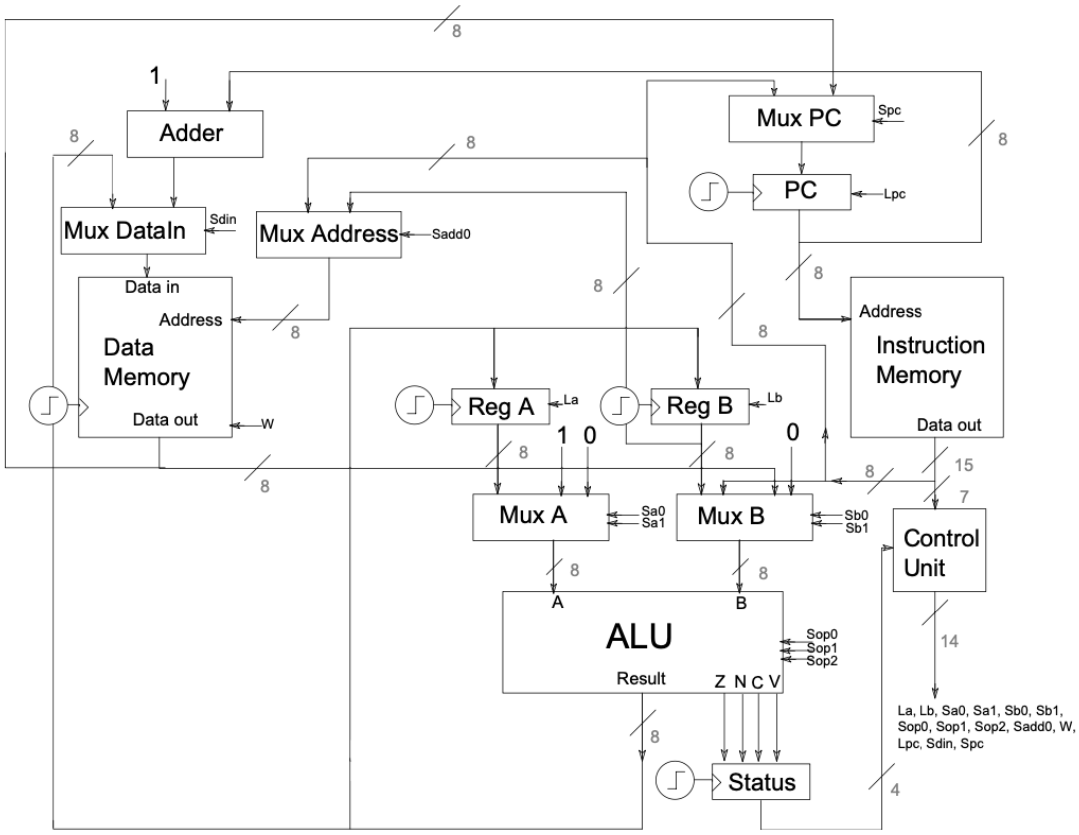


Figura 3: Almacenamiento del program counter

Con esta modificación de hardware podemos agregar dos instrucciones para realizar el llamado a subrutina y retorno de ellas:

1. La instrucción `CALL dir` la cual almacena el PC en memoria y salta al *label* `dir` donde estará almacenada la subrutina
2. La instrucción RET la cual se debe agregar al final de la subrutina y salta de vuelta a la dirección almacenada previamente del PC.

Con estas dos instrucciones se puede reescribir el código completo del programa que calcula el producto punto de la siguiente forma:

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  JMP init
      mult:
                  MOV A, 0
                  MOV (res), A
                  MOV A, 0
```

```
start:    MOV (i), A
          MOV  A,(res)
          ADD  A,(var2)
          MOV  (res),A
          MOV  A,(i)
          ADD  A,1
          MOV  (i),A
          MOV  B,(var1)
          CMP  A,B
          JLT  start
          RET

init:     MOV A, (vector1)
          MOV (var1), A
          MOV A, (vector2)
          MOV (var2), A
          CALL mult
          MOV A, (prodPunto)
          ADD A, (res)
          MOV (prodPunto), A

          MOV B, vector1
          INC B
          MOV A, (B)
          MOV (var1), A
          MOV B, vector2
          INC B
          MOV A, (B)
          MOV (var2), A
          CALL mult
          MOV A, (prodPunto)
          ADD A, (res)
          MOV (prodPunto), A
```

Para completar el manejo de subrutinas falta aún un elemento: en que parte de la memoria se guarda el valor del PC al hacer el llamado a la subrutina. Una opción es que el programador se encargara de indicar una dirección (por ejemplo almacenándola en el registro B de direccionamiento indirecto) previo a la llamada y previo al retorno. El problema de esto es que agrega complejidad a la programación.

Para solucionar este problema, se agrega un nuevo registro al computador denominado ***stack pointer*** o **SP** el cual comienza cargado con el valor de la última dirección de la memoria. La idea será entonces ocupar esa dirección para realizar la carga del program counter al ejecutar la instrucción CALL. Luego, al llamar a la instrucción RET se leerá el valor de memoria apuntado por el *stack pointer* y se cargará en el *program counter* para volver al flujo normal del programa.

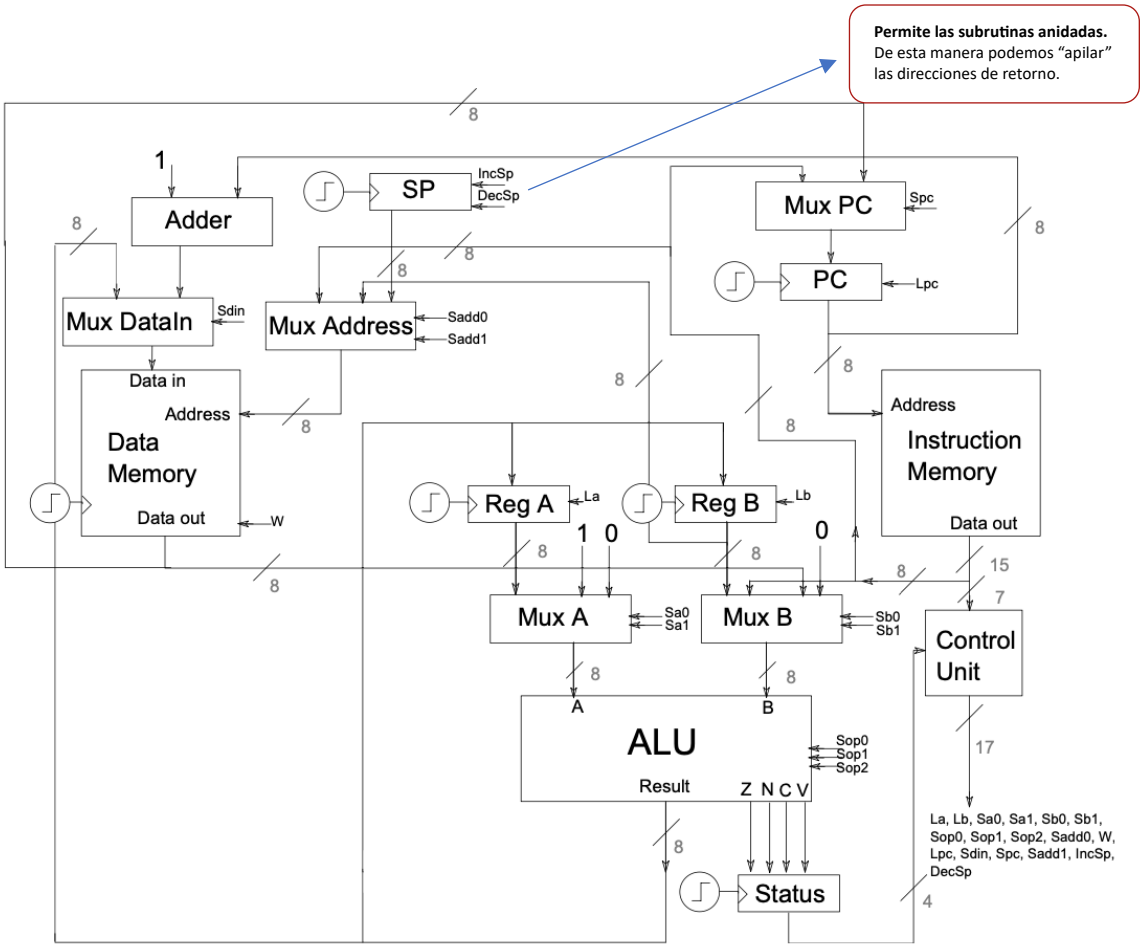


Figura 5: Agregadas señales de incremento y decremento al stack pointer

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
MOV	A,B	A=B		-
	B,A	B=A		-
	A,Lit	A=Lit		MOV A,15
	B,Lit	B=Lit		MOV B,15
	A,(Dir)	A=Mem[Dir]		MOV A,(var1)
	B,(Dir)	B=Mem[Dir]		MOV B,(var2)
	(Dir),A	Mem[Dir]=A		MOV (var1),A
	(Dir),B	Mem[Dir]=B		MOV (var2),B
	A,(B)	A=Mem[B]		-
	B,(B)	B=Mem[B]		-
	(B),A	Mem[B]=A		-
ADD	A,B	A=A+B		-
	B,A	B=A+B		-
	A,Lit	A=A+Lit		ADD A,5
	A,(Dir)	A=A+Mem[Dir]		ADD A,(var1)
	A,(B)	A=A+Mem[B]		-
	(Dir)	Mem[Dir]=A+B		ADD (var1)
SUB	A,B	A=A-B		-
	B,A	B=A-B		-
	A,Lit	A=A-Lit		SUB A, 2
	A,(Dir)	A=A-Mem[Dir]		SUB A,(var1)
	A,(B)	A=A-Mem[B]		-
	(Dir)	Mem[Dir]=A-B		SUB (var1)
AND	A,B	A=A and B		-
	B,A	B=A and B		-
	A,Lit	A=A and Lit		AND A,15
	A,(Dir)	A=A and Mem[Dir]		AND A,(var1)
	A,(B)	A=A and Mem[B]		-
	(Dir)	Mem[Dir]=A and B		AND (var1)
OR	A,B	A=A or B		-
	B,A	B=A or B		-
	A,Lit	A=A or Lit		OR A,5
	A,(Dir)	A=A or Mem[Dir]		OR A,(var1)
	A,(B)	A=A or Mem[B]		-
	(Dir)	Mem[Dir]=A or B		OR (var1)
NOT	A,A	A=not A		-
	B,A	B=not A		-
	(Dir)	Mem[Dir]=not A		NOT (var1)

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
XOR	A,B	A=A xor B		-
	B,A	B=A xor B		-
	A,Lit	A=A xor Lit		XOR A,15
	A,(Dir)	A=A xor Mem[Dir]		XOR A,(var1)
	A,(B)	A=A xor Mem[B]		-
	(Dir)	Mem[Dir]=A xor B		XOR (var1)
SHL	A,A	A=shift left A		-
	B,A	B=shift left A		-
	(Dir)	Mem[Dir]=shift left A		SHL (var1)
SHR	A,A	A=shift right A		-
	B,A	B=shift right A		-
	(Dir)	Mem[Dir]=shift right A		SHR (var1)
INC	B	B=B+1		-
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label
CALL	Dir	Mem[SP] = PC + 1, SP–, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP–		-
PUSH	B	Mem[SP] = B, SP–		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
		B = Mem[SP]		-

Arquitectura de computadores

- **Microarquitecturas:** La microarquitectura de un computador se refiere a los distintos componentes de *hardware* que estarán presente en un sistema computacional.

    - **Registros**: Distintos computadores tendrán distinta cantidad y tamaño de registros. En el caso del computador básico visto hasta ahora se tienen dos registros de 8 bits.
        - Más registros → Menos traspasos de datos a memoria.
        - Mayor tamaño → Permite realizar operaciones con números de mayor rango y precisión.

    - **Unidades de ejecución**: La unidad básica de ejecución de todo computador es la ALU y como tal estará presente en toda microarquitectura.

    - **Unidad de control**: Existen 2 modelos de unidades de control principales:
        - Sistemas ***hard-wired***: Cada instrucción tiene asociado directamente las señales de control que ejecutan una función en el computador.
        - Sistemas **micro-programadas**: Cada instrucción del programa se traduce en un conjunto de microinstrucciones, que ejecutan sub-funciones muy específicas.

    - ***Condition codes***: Sistemas minimalistas trabajarán sólo con los *condition codes* Z, N, C y V por ejemplo.

    - ***Stack***: La presencia de un *stack* para subrutinas también es un elemento que variará según el sistema computacional.

    - **Memorias**: La cantidad de palabras de las memorias y el tamaño de cada una de estas también será un elemento que variará entre distintos computadores.

- **Arquitectura del Set de Instrucciones (ISA)**: Las instrucciones de un computador definirán como se deben escribir programas, y se diferenciarán por los siguientes factores.

    - **Tipos de instrucciones**: Dependiendo del ISA, existirán distintos tipos de instrucciones disponibles.

    - **Tipos de datos**: Distintos ISA pueden definir distintos tipos de datos que son soportados.

    - **Modos de direccionamiento**: Algunos de los posibles modos de direccionamiento son los siguientes:

        - Direccionamiento inmediato o literal.
        - Direccionamiento por registro.
        - Direccionamiento directo.
        - Direccionamiento indirecto por registros.
        - Direccionamiento indirecto por registro base + *offset*.
        - Direccionamiento indirecto por registro base + registro índice.
        - Direccionamiento indirecto por registro con post incremento.
        - Direccionamiento indirecto por registro con post decremento.
        - Direccionamiento indirecto.

    - **Manejo del stack**: El manejo del stack variará según cada computador, lo que se ve reflejado en las instrucciones disponibles en el ISA.

    - **Formato de la instrucción**: Dependiendo del ISA, este formato será general para todas las instrucciones, o existirán distintos formatos específicos.

    - **Palabras por instrucción**: La cantidad de palabras que se requieran para almacenar una instrucción variará en distintos ISAs.

    - **Ciclos por instrucción**: La cantidad de ciclos que se demora en ejecutar cada instrucción también variará en distintos ISAs.

- ***Reduce Instruction Set Computer* (RISC) (pertenece a ISA)**:

  Los sets de instrucciones RISC fueron desarrollados con el objetivo de minimizar la complejidad del hardware del computador, simplificando la arquitectura y diseño de este. En general un computador con ISA RISC se califica como que tiene «énfasis en el *software*» ya que el *hardware* del computador sólo provee funcionalidades básicas, y las funcionalidades avanzadas son implementadas por *software*. Sus características principales son las siguientes:

    - Instrucciones de un sólo ciclo de *clock*.
    - Unidad de control *hard-wired*.
    - Formato de instrucción uniforme e idealmente almacenado en sólo una palabra.
    - Registros de propósito general idénticos, que permiten todos realizar las mismas funciones.
    - Modos de direccionamiento simples.
    - Códigos en *assembly* largos.
    - Pocos tipos de datos soportados directamente en *hardware*.

Arquitectura x86 (16 bits)

- **Variables**: El *assembly* de la arquitectura x86 de 16 bits soporta dos tipos de datos principales:
  - Tipo *byte*: de 8 bits representado por el símbolo db
  - Tipo *word*: de 16 bits representado por el símbolo dw. La declaración de variables en el *asssembly* tiene la siguiente sintaxis: `Identificador Tipo Valor`.

Ejemplo multiplicación: Código Assembly x86.

```
;Calculo de la multiplicacion res = a*b

MOV AX, 0
MOV CX, 0
MOV DX, 0

MOV CL, a          ;CL guarda el valor de a
MOV DL, b          ;DL guarda el valor de b

start:
CMP CL, 0           ;IF a <= 0 GOTO end
JLE endprog

ADD AX, DX          ;AX += b

DEC CL              ;a--
JMP start

endprog:

MOV res, AX        ;res = AX

RET

a       db 10
b       db 200
res     dw 0
```

En la arquitectura de 16 bits vienen implementadas las instrucciones `MUL op` y `DIV op` las cuales implementan las operaciones de multiplicación y división entera respectivamente, ocupando el registro AX como operando y resultado, de la siguiente forma:

- `MUL op => AL = AL*op`
- `DIV op => AL = AX/op`

```
;Calculo de la multiplicacion res = a*b

MOV AX, 0

MOV AL, a          ;AL = a
MUL b              ;AX = AL*b

MOV res, AX        ;res = AX

RET

a       db 10
b       db 200
res     dw 0
```

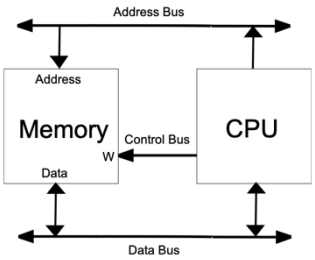
Ejemplo de potencia sin variables locales

```
;Calculo de la potencia pow = base*exp

MOV BL, exp
MOV CL, base
PUSH BX             ;Paso de parmetros (de derecha a izquierda)
PUSH CX
CALL potencia       ;potencia(base,exp)
MOV pow, AL         ;Retorno viene en AX
RET

potencia:           ;Subrutina para el calculo de la potencia
PUSH BP
MOV BP, SP          ;Actualizamos BP con valor del SP

MOV CL, [BP + 4]    ;Recuperamos los dos parametros
MOV BL, [BP + 6]

MOV AX, 1           ;AX = 1

start:
CMP BL,0            ;if exp <= 0 goto endpotencia
JLE endpotencia

MUL CL              ;AX = AL * base

DEC BL              ;exp--
JMP start
endpotencia:

POP BP
RET 4               ;Retornar, desplazando el SP en 4 bytes

base      db 2
exp       db 7
pow       db 0
```

Ejemplo de potencia con variables locales

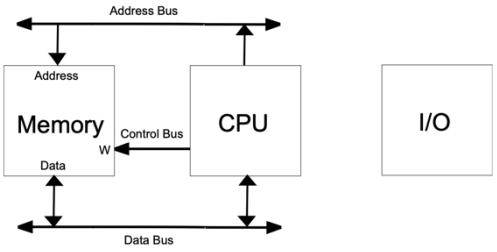
```
;Calculo de la potencia pow = base*exp

MOV BL, exp
MOV CL, base

PUSH BX             ;Paso de parmetros (de derecha a izquierda)
PUSH CX

CALL potencia       ;potencia(base,exp)

MOV pow, AL      ;Retorno viene en AX

RET

potencia:           ;Subrutina para el calculo de la potencia

PUSH BP
MOV BP, SP          ;Actualizamos BP con valor del SP
SUB SP, 2           ;Reservamos espacio para variable i

MOV CL, [BP + 4]    ;Recuperamos los dos parametros
MOV BL, [BP + 6]

MOV AX, 1           ;AX = 1
MOV [BP - 2],0

start:
CMP [BP - 2],BL     ;if i >= n goto endpotencia
JGE endpotencia

MUL CL              ;AX = AL * base

INC [BP - 2]        ;i++
JMP start

endpotencia:

ADD SP, 2
POP BP
RET 4               ;Retornar, desplazando el SP en 4 bytes

base      db 2
exp       db 7
pow       db 0
```

Canales de comunicación



Los canales de comunicación que habitualmente se ocupan para los distintos tipos de comunicación en un computador corresponden a **buses compartidos**. Un bus compartido consiste en un canal de cables eléctricos que puede ser accedido por más de un dispositivo a la vez.

Dispositivos de entrada y salida: I/O (*input/output*)



El modelo general de un computador incluye una gran variedad de dispositivos que pueden ser conectados éste, para ingresar y recibir datos. A este conjunto de dispositivos se les conoce como dispositivos *input/output* (I/O). Considerando los dispositivos de I/O, el modelo del computador *Von Neumann* se extiende de la siguiente forma.

- **Comportamiento**: El comportamiento de un dispositivo I/O indica que puede hacer el computador con el dispositivo. Las opciones son:

    - Entrada (*Input*): Entregan datos al computador, pero no permiten que el computador envíe datos.
    - Salida (*Output*): Reciben datos del computador, pero no entregan datos.
    - Entrada o salida (*Input or Output*): Permiten entregar datos al computador o recibir datos, pero no de manera simultánea.
    - Almacenamiento (*Input and Output*): Entregar y recibir datos almacenados en el dispositivo.

- **Comunicante**: La segunda característica indica con quien se está comunicando el dispositivo:

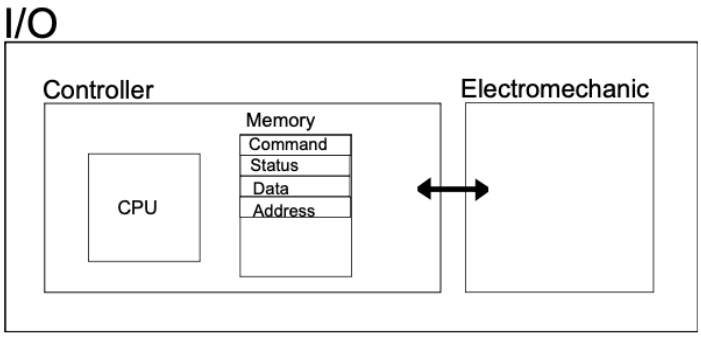
    - Humano: Dispositivos como el *mouse*, teclado y la pantalla, son dispositivos que se comunican con un ser humano.
    - Máquina: Dispositivos como la tarjeta de red, o los discos duros se comunican con máquinas, sin intervención de un ser humano.

- **Velocidad de transferencia**: La velocidad de transferencia de los dispositivos de I/O es muy variable entre dispositivos, y a veces en un mismo dispositivo. En general se utiliza la velocidad ***peak*** como referencia para cada dispositivo.

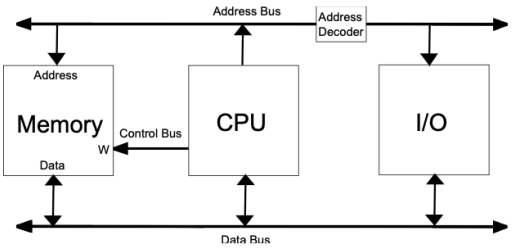
- **Componentes de I/O**

    - Controladores: Circuitos digitales encargados de controlar las partes electro/mecánicas del dispositivo según la comunicación realizada con el computador.
    - Circuitos de control: Encargados de regular el funcionamiento del dispositivo, coordinar la comunicación con el computador y ejecutar la detección y/o corrección de errores.
    - Conversores ADC o DAC: Pueden incluir conversores análogo-digital o digital-análogo para traducir señales eléctricas continuas en información para el computador o vice-versa.
    - Memoria: Será el mecanismo fundamental para comunicarse con el computador.
        - Buffer: Almacena los datos que el dispositivo este entregando o recibiendo del computador.
        - Registros de control: Direcciones específicas de la memoria que indican comandos que debe ejecutar el dispositivo.
        - Registros de status: Direcciones específicas de la memoria que indican información al computador.

- Registros de datos: direcciones específicas de la memoria para leer o escribir datos individuales o asociados a la memoria local (buffer) del dispositivo.
- Registros de dirección: direcciones específicas de la memoria para direccionar la memoria local (*buffer*) del dispositivo.



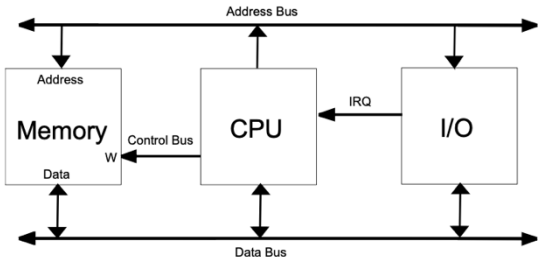
***Memory mapped* I/O**



Esta técnica permite que los dispositivos de I/O se comuniquen con la CPU y accedan a la memoria del sistema utilizando instrucciones de lectura y escritura normales.

El mapeo explícito a los distintos registros y buffers de los dispositivos los coordina el *address decoder*, una pieza de *hardware* especializada que estará «vigilando» el bus de dirección para determinar si la dirección colocada corresponde a la RAM o alguno de los dispositivos.

Interrupciones



En un sistema de I/O basado en interrupciones, el dispositivo de I/O será el encargado de avisar cuando ocurre un suceso relevante a la CPU, liberando a ésta de tener que estar preguntándole a cada dispositivo si ha habido algún evento relevante.

Para lograr esto, el dispositivo de I/O se debe conectar a la CPU con una señal de control dedica denominada ***interrupt request*** o **IRQ**. Mediante esta señal de 1 bit el dispositivo le notificará a la CPU si ocurrió algo relevante.

***Programmed* I/O**

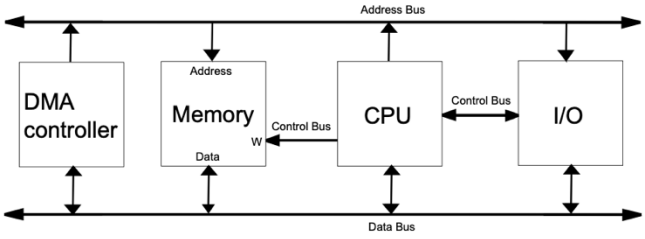
Aun considerando el uso de interrupciones, todavía existe una fuente de ineficiencia en la comunicación con los dispositivos de I/O. El problema ocurre cuando un dispositivo tiene que copiar datos a memoria (por ejemplo el disco duro). Supongamos que la CPU iniciara la comunicación y le solicita al dispositivo copiar ciertos datos a memoria, el algoritmo sería de la siguiente forma:

1. Configurar el dispositivo para ser leído, enviando señales de control correspondientes.
2. Leer un dato del dispositivo, ocupando *memory map* o *port* I/O. El dato queda guardado en un registro de la CPU.
3. Leer un dato del dispositivo, ocupando memory map o port I/O. El dato queda guardado en un registro de la CPU.
4. Leer un dato del dispositivo, ocupando memory map o port I/O. El dato queda guardado en un registro de la CPU.

Esta forma de transferencia de datos se conoce como Programmed I/O (PIO), ya que se realiza a través del programa.

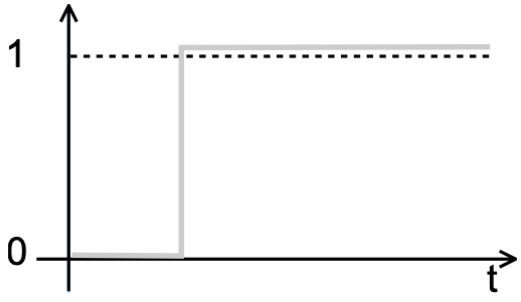
Direct Memory Access (DMA)

Para permitir que los dispositivos de I/O se comuniquen directamente con la memoria, se agrega un componente de *hardware* conocido como **controlador DMA**. El controlador DMA tendrá acceso al bus de datos y se encargará de traspasar la información que el dispositivo le envíe directamente a memoria sin pasar por la CPU.

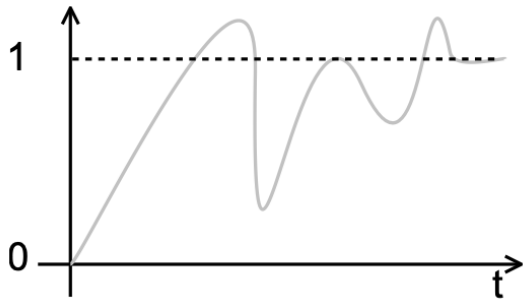


Interacción Humano-Computador

Si los interruptores y botones fueran ideales, cuando se conectan deberían inmediatamente representar un 1, y cuando se desconectan, inmediatamente un 0, como se observa:



Sin embargo, esto no ocurre en la práctica, debido a que los circuitos eléctricos no son ideales, y por tanto no tienen transiciones instantáneas. Lo que ocurre en la práctica es lo que se observa en la siguiente figura, al cerrar el interruptor, la señal eléctrica «**rebota**» varias veces en torno al valor de voltaje asociado al 1 binario, hasta que finalmente se regula en el estado correcto. Mientras la señal está rebotando se dice que esta está en **régimen transiente**, y cuando se estabiliza, entra en **régimen permanente**.



La solución a este problema consiste en agregar entre el interruptor y el componente un circuito de retraso, que sólo dejará pasar la señal luego de un cierto tiempo. De esta forma, si este circuito se calibra para dejar pasar la señal sólo después del transiente, se evita el problema del rebote.

Memoria caché

La memoria caché es un componente esencial en la jerarquía de memoria de un computador. Se trata de un lugar de almacenamiento intermedio que se comunica directamente con la CPU y aprovecha el principio de localidad, donde datos cercanos en el tiempo tienden a ser accedidos nuevamente. La caché se utiliza para acelerar el acceso a datos que son frecuentemente utilizados por la CPU, reduciendo así los tiempos de acceso a la memoria principal.

Mecanismo de acceso a los datos

Para poder almacenar bloques desde la memoria principal, la caché debe definir una **función de correspondencia** entre las direcciones de la memoria principal y su propia memoria. La función de correspondencia más simple se conoce como ***directly mapped***. Con este mecanismo, cada **bloque** de la memoria principal tiene asociado un solo **bloque** en la memoria caché. La caché identificará cada bloque interno con un **índice** (equivalente a una dirección, pero para bloques). La fórmula de asociación entre un bloque de memoria y un índice de caché depende de los siguientes factores:

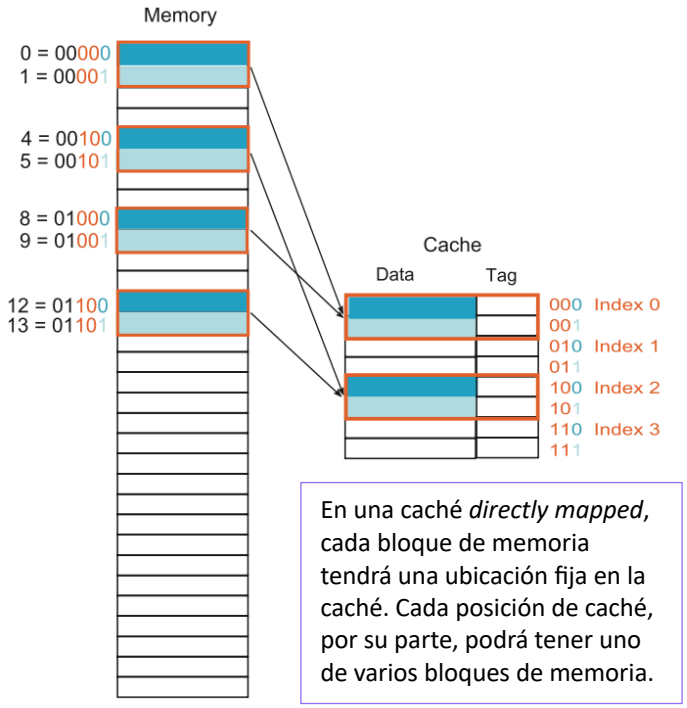
- Tamaño de cada bloque de caché. Por simplicidad siempre será una potencia de 2.
- Cantidad de bloques en caché. Por simplicidad siempre será una potencia de 2.
- Dirección del bloque de memoria. La dirección del bloque de memoria será la dirección de la primera palabra del bloque.

Para observar cómo se realiza la asociación entre memoria y caché, revisemos el siguiente ejemplo se tiene:

- Una memoria principal de 32 bytes (32 palabras).
- Una caché de 8 bytes, con 4 bloques cada uno de 2 bytes (2 palabras). El 5to bloque de memoria está asociado a la dirección $8 = 01000$. Para obtener su asociación en caché hay que descomponer la dirección de la siguiente forma:

    - Como la caché tiene bloques de 2 palabras $= 2^1$, se requiere **1** bit para determinar la posición de la palabra dentro del bloque. En este ejemplo entonces, el bit menos significativo de la dirección se usará para indicar la posición dentro del bloque, o sea 01000, por lo que en este caso, la palabra asociada a la dirección $8 = 01000$ estará almacenada en la palabra 0 del bloque.

    - Como la caché tiene 4 bloques $= 2^2$, se requieren **2** bits para determinar el índice del bloque dentro de la caché. Se usarán los siguientes dos bits de la dirección para determinar el índice, o sea 01000, por lo que en este caso, el bloque asociado a la dirección $8 = 01000$ se almacenará en el bloque 00 de caché.



En una caché *directly mapped*, cada bloque de memoria tendrá una ubicación fija en la caché. Cada posición de caché, por su parte, podrá tener uno de varios bloques de memoria.

Un elemento adicional que se debe almacenar para cada palabra es un bit de validez o ***valid bit***. Este bit indica si los valores almacenados en caché son válidos, lo que es importante al comenzar a llenar la caché, ya que cuando la caché está vacía, las palabras que tienen almacenadas no son válidas.

Políticas de escritura

En el caso que la CPU quiera escribir un dato en memoria además de preocuparse del acceso, debe encargarse de actualizar la memoria principal para que está mantenga consistencia en la información. Existen dos políticas usadas para realizar esta actualización:

- ***Write-through***: en esta política cada escritura en caché, se actualiza inmediatamente en memoria.

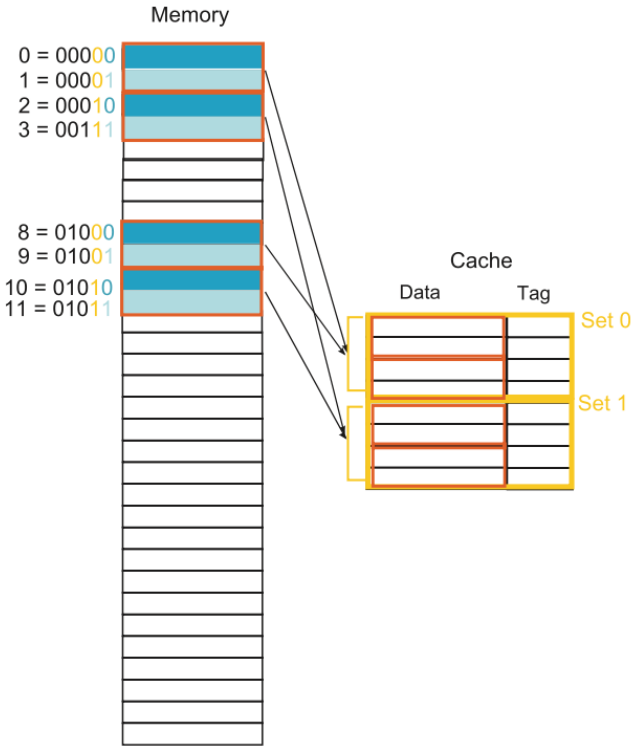
- ***Write-back***: en esta política las escrituras se realizan en principio sólo en caché. El valor se actualizará en memoria sólo cuando el bloque que estaba en caché vaya a ser reemplazado.

Funciones de correspondencia

Una caché con función de correspondencia ***directly mapped*** presenta la desventaja de que pueden haber muchas colisiones.

- ***Fully associative***: cada bloque de memoria puede asociarse a cualquier bloque de la caché. De esta manera se aprovecha al máximo la disponibilidad de espacio en la caché y se evita el problema de la contención. El problema de esta función de correspondencia, es que ante un requerimiento de memoria de la CPU, ahora se vuelve más difícil encontrar la palabra en caché. Para saber si una palabra está en caché es necesario buscar en todas las posiciones, y además es necesario guardar un tag más grande, que contenga toda la dirección menos los bits de ubicación de la palabra en el bloque.

- ***N-way associative***: la caché se divide lógicamente en **conjuntos** o sets, siendo cada uno un *group* de bloques. Cada bloque de memoria tendrá un mapeo directo a un sólo conjunto, pero dentro del conjunto podrá ubicarse en cualquier bloque disponible. De esta forma, al tener el mapeo directo al conjunto, se reducen las comparaciones al momento de buscar si está un dato o no en caché, y al tener libertad de elegir en que bloque del conjunto almacenar, se mantienen niveles de *hit rate* altos.
  En una caché *n-way associative*, cada bloque de memoria estará asociado a un conjunto de bloques de la caché, y dentro del conjunto se podrá ubicar en cualquier lugar. En este caso, la caché es 2-way associative, es decir, tiene conjuntos de 2 bloques cada uno.



Políticas de reemplazo:

- *First-in First-out (FIFO)*: Primero que sale, primero que entra.
- *Least frequently used (LFU)*: Reemplaza el bloque que ha sido accedido con menos frecuencia.
- *Last recently used (LRU)*: Reemplaza el bloque que se usó hace más tiempo.
- *Random*: Elegir al azar el bloque a reemplazar.

Multiprogramación

El concepto de multiprogramación se refiere a la idea general de poder cargar múltiples programas dentro de un mismo computador para que sean ejecutados en un determinado momento.

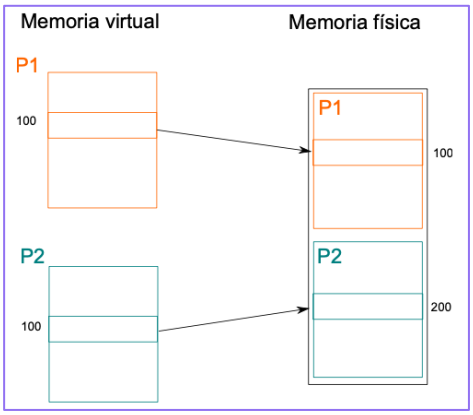
Un programa está compuesto por dos partes:

1. Su **representación en memoria** que incluye el código, datos y stack del programa.

2. Su **estado de ejecución** que incluye los valores almacenados en los registros de la CPU que indican el estado actual del programa en la máquina.

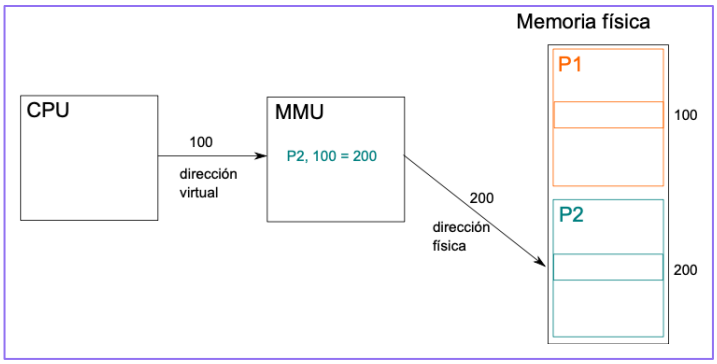
Para lograr trabajar con múltiples programas, entonces, es necesario permitir el manejo de múltiples representaciones en memoria y de múltiples estados de ejecución.

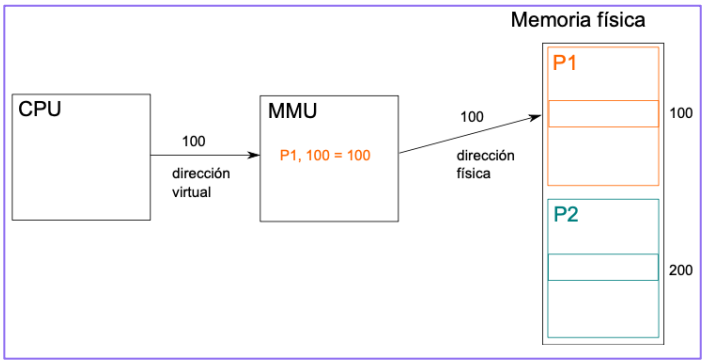
Manejo de memoria



Memoria virtual: Separa direcciones virtuales (internas al programa) de direcciones físicas. Cada programa tiene su espacio virtual, creando la ilusión de ser el único en la máquina. Estas direcciones se mapean en ubicaciones físicas diferentes, pero al programa esto le es transparente; solo le interesa su espacio virtual.

Para implementar un sistema de memoria virtual, es necesario agregar a la CPU un componente de hardware que realice la traducción de una dirección virtual a una dirección física. Este componente se le conoce como **Memory Management Unit (MMU)** y es parte de la CPU. La MMU se encargará de traducir las solicitudes de memoria que vengan desde la CPU, mapeando para el programa actual la dirección física correspondiente.





Para realizar la traducción la MMU debe almacenar una tabla que tenga la asociación virtual-física. La opción más simple para esto es almacenar todos los pares de direcciones virtual-física, lo que en el caso de los programas 1 y 2 serían los siguientes:

Dirección virtual	Dirección física
100	100
101	101
102	102

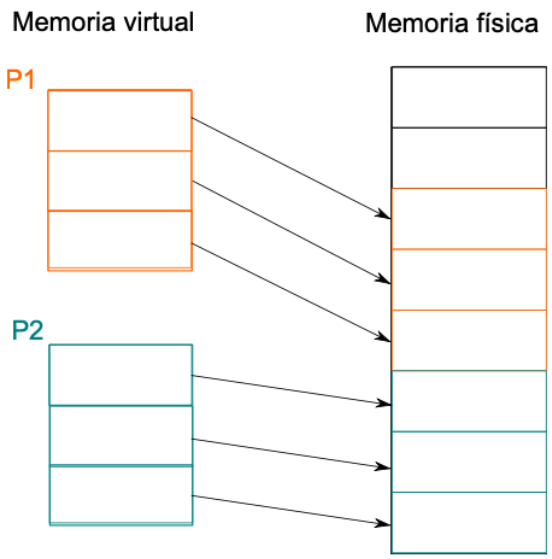
Tabla 1: Tabla de mapeo virtual-físico del programa 1.

Dirección virtual	Dirección física
100	200
101	201
102	202

Tabla 2: Tabla de mapeo virtual-físico del programa 2.

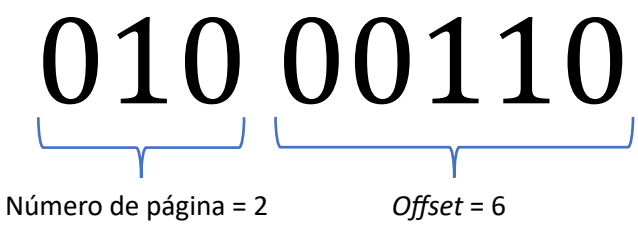
El problema de almacenar todos los mapeos posible está en que se necesitan tantas tablas como programas hay, y por tanto se requeriría para cada programa un espacio de almacenamiento igual al tamaño dela memoria física, lo que claramente no es implementable en la práctica.

Para solucionar el problema del tamaño de la tabla de mapeo virtual-físico, se agrega el concepto de **paginación**. La paginación corresponde a dividir la memoria en bloques de palabras contiguos conocidos como páginas en el espacio virtual o marcos en el espacio físico. De esta manera, cada programa tendrá asociada una cierta cantidad de páginas de memoria virtual, las cuales estarán mapeadas a marcos físicos. Con este esquema, las tablas de mapeo, denominadas **tablas de páginas**, pueden tener un tamaño razonable lo que permite implementar el sistema en la práctica.



En un sistema de memoria virtual con paginación, la dirección de memoria será interpretada como dos partes: una que indica el número de página, y otra que indica el offset dentro de la página.

Si tomamos un espacio de direccionamiento de 8 bits:



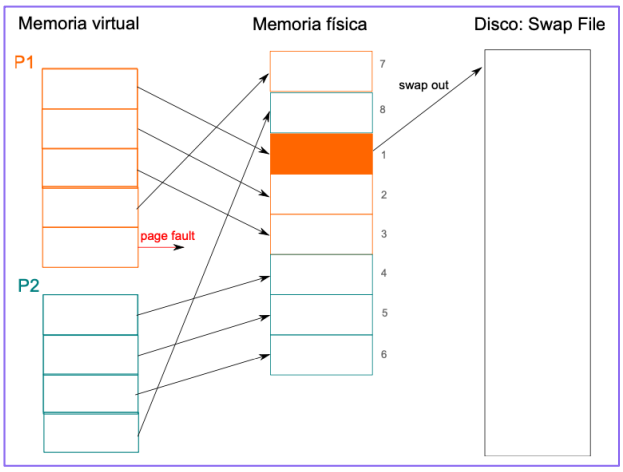
La página virtual debe ser traducida a un marco físico ocupando la tabla de página correspondiente. Supongamos la página 2 esta mapeada al marco 7 = 111. Ocupando el número del marco físico (111) más el offset original (00110) se obtiene la dirección física real: 11100110 = 230.

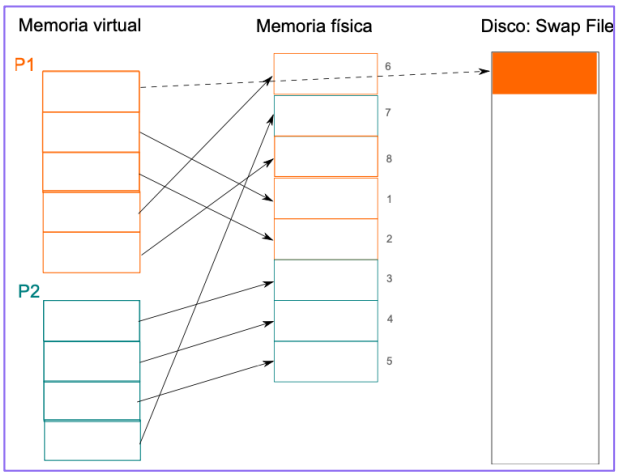
De tener las tablas de página en memoria, es que para cada acceso a memoria de un programa dado, se requieren dos accesos en la práctica: uno para ir a buscar el mapeo virtual-físico en la tabla de página y otro para realizar el acceso real.

Para mejorar el rendimiento en los accesos a memoria, y evitar que siempre ocurra un doble acceso, se agrega una caché especialmente dedicada a almacenar entradas de tabla de página, la cual se conoce como ***Translation Lookaside Buffer*** **(TLB)**. La TLB almacenará algunas de las entradas de la tabla de página del programa que actualmente está en ejecución.

¿Qué pasa si se llena la memoria física?

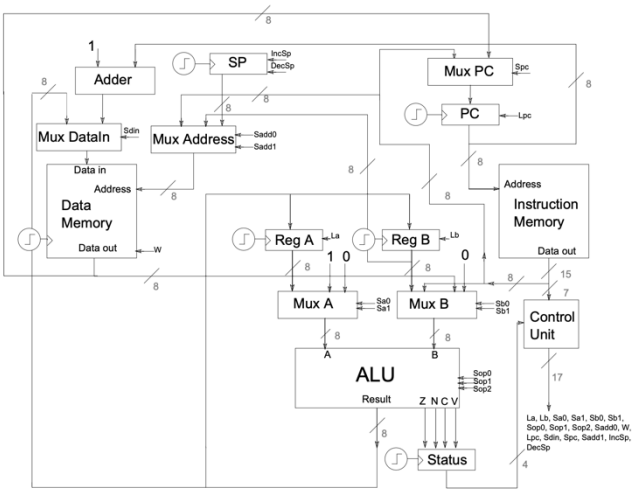
Se utiliza el disco duro como almacenamiento de respaldo para los marcos de memoria. Para lograr esto se reserva un espacio especial en el disco, denominado **swap file**, el cual será utilizado para respaldar marcos. De esta forma cuando un programa requiere un marco, pero la memoria está completamente ocupada, se copiará un marco de memoria al disco para dejar espacio para la nueva solicitud. Este proceso de respaldo de memoria a disco se conoce como **swap out**.

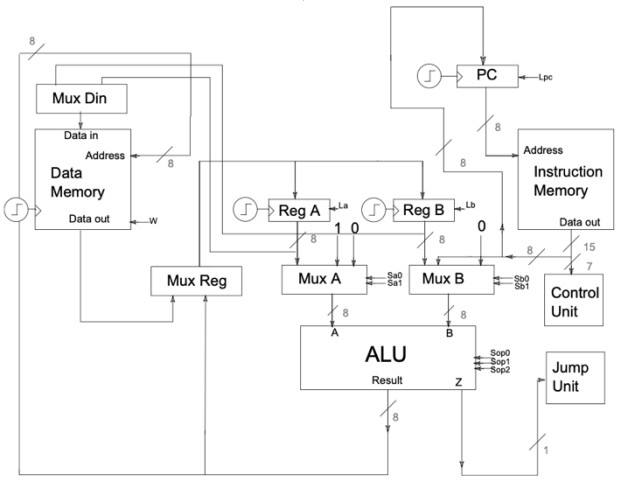




Ahora si el programa 1 requiere el acceso a la página 0:

- Ocurrirá otro *page fault*.
- Se realiza un ***swap out*** respaldando el bloque a reemplazar.
- El sistema operativo revisa si la página solicitada está en disco (sí está).
- Se realiza un ***swap in*** del disco a memoria, restaurando el mapeo original.



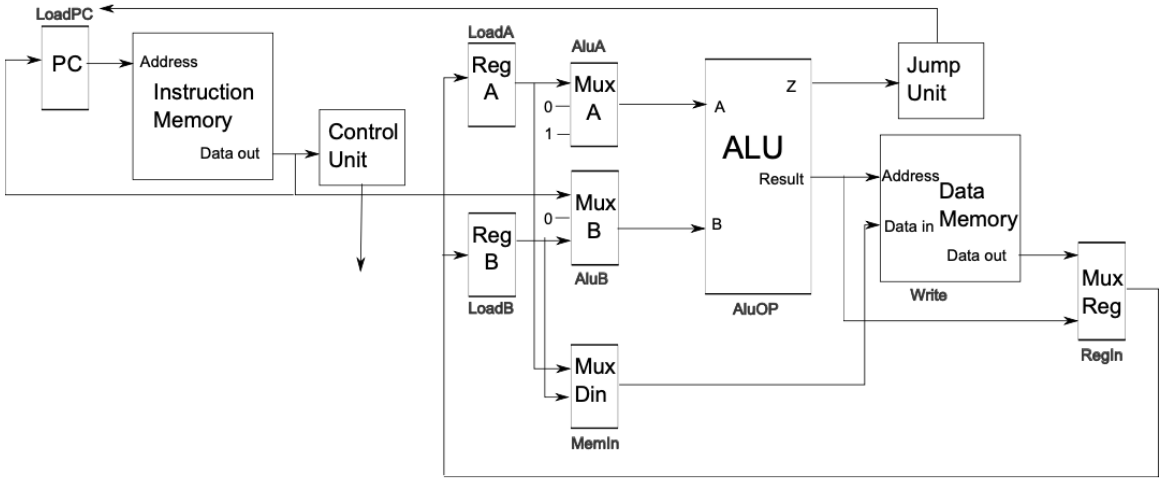


Paralelismo

El ciclo actual de una instrucción en el computador básico *Harvard* que se ha estudiado, aunque es simple, presenta algunos elementos que harán innecesariamente difícil el análisis que realizaremos es este capítulo. Para evitar estos problemas y trabajar con un ciclo más simple, se usará una versión simplificada del computador básico.

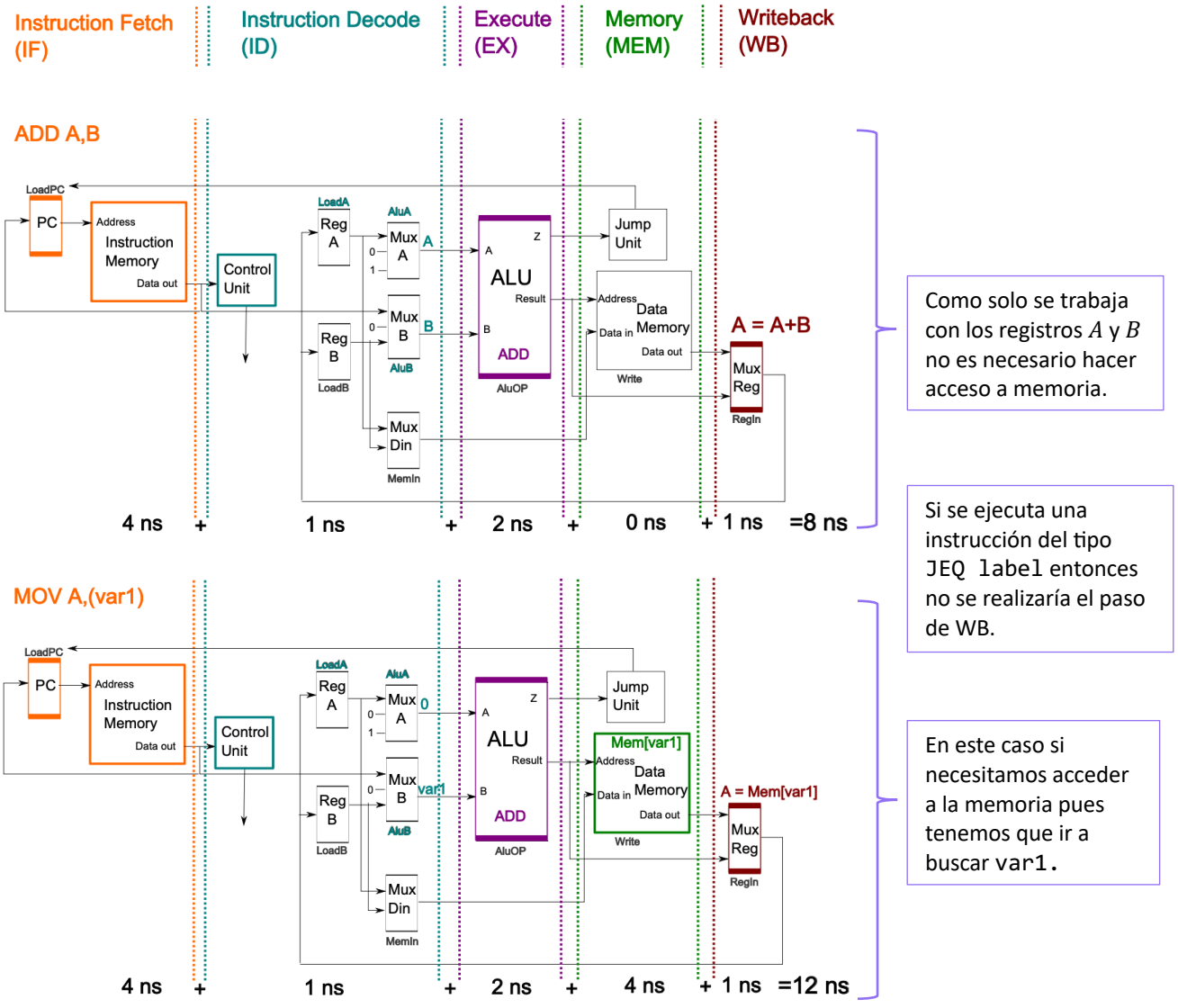
- Se eliminó el soporte del *stack*, eliminando el *stack pointer* y la conexión del PC con memoria.
- Se simplificaron los saltos condicionales (*Z*).
- Se eliminó la conexión entre la salida de memoria y el MUX B.
- La entrada de datos de la memoria ahora solo puede provenir de los registros A y B, y no de la ALU.
- La dirección de la memoria ahora proviene de la ALU.
- La unidad de salto, se separó de la unidad de control.

La idea detrás de estas modificaciones es, por un lado, simplificar el computador, pero también lograr que el ciclo de las instrucciones sea más uniforme que en la versión original. Con estas modificaciones es posible reorganizar las partes del computador, para mostrar con más claridad las distintas partes del ciclo, lo que se observa en la figura 3.



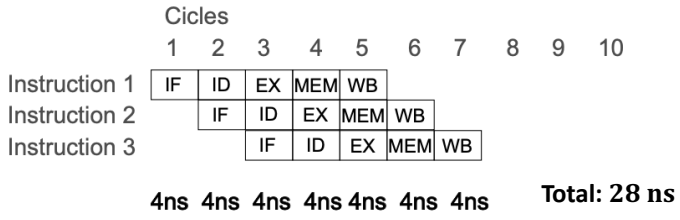
El ciclo de la instrucción para este computador contará de 5 etapas:

1. **Instruction fetch (IF)**: Se va a buscar a la memoria de instrucciones la siguiente instrucción, apuntada por el valor actual del *Program Counter*. La salida de esta etapa es la instrucción obtenida desde memoria, la cual se separará en un *opcode* y en el parámetro.
2. **Instruction decode (ID)**: La unidad de control se encargará de transformar el opcode de la instrucción obtenida en las señales de control específicas que le indicarán al procesador que tarea ejecutar.
3. **Execute (EX)**: Realizada por la ALU. Esta generará el *condition code* Z.
4. **Memory (MEM)**: Lectura o escritura en memoria de datos. Esta etapa solo estará presente si a la instrucción ejecutada le correspondía transferencia hacia o desde memoria (por ejemplo la instrucción `MOV (var_1, A)`).
5. **Writeback (WB)**: Escribir en los registros, ya sea un resultado de la ALU o un dato obtenido desde memoria (por ejemplo las instrucciones `ADD A,B` y `MOV A, (var1)`). Solo estará presente para algunas instrucciones.

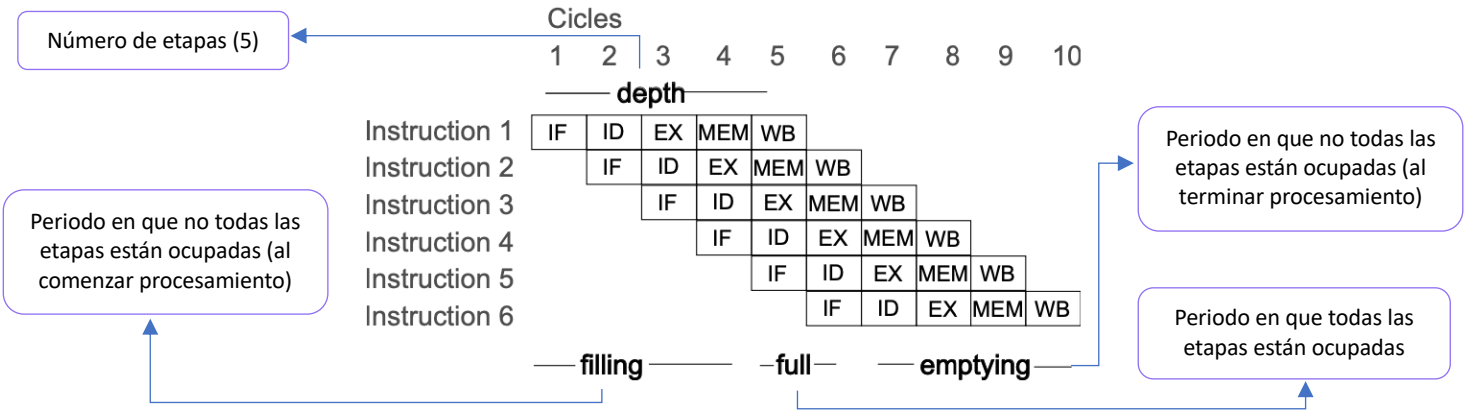


Instruction pipeline

Supongamos que tenemos 3 instrucciones seguidas. Podemos aprovechar que una vez que termina una etapa no es necesario esperar a que termine la instrucción completa para seguir ejecutando otras (tendremos que tomar un intervalo de 4 ns ya que esa es la etapa que más se demora).



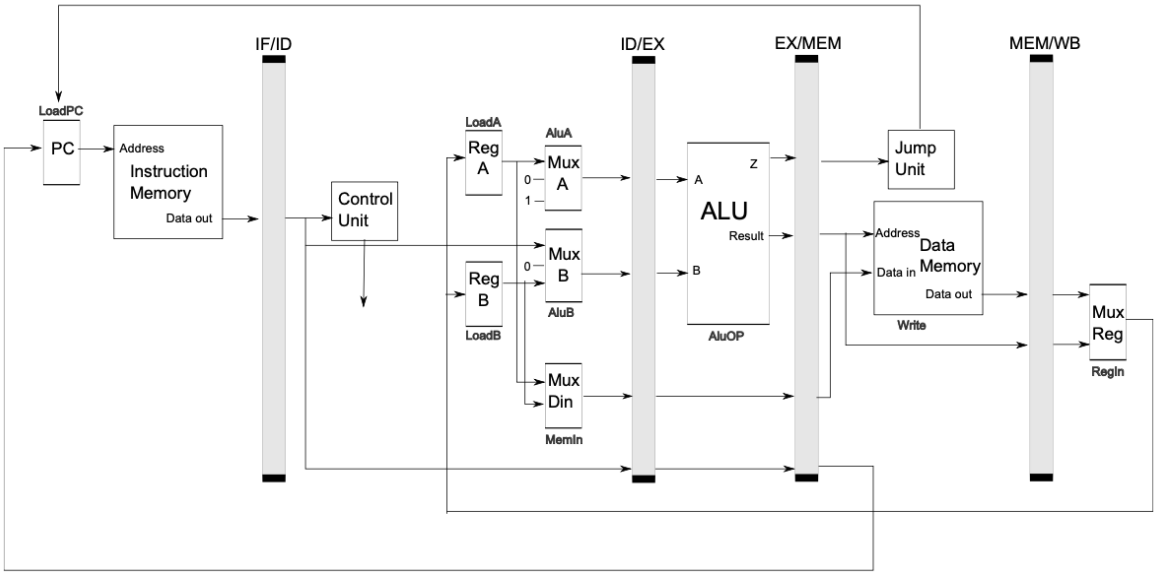
La gran paradoja del *instruction pipeline* es que, el tiempo para procesar cada instrucción no disminuye, sino que **aumenta**. Esto se debe a que la etapa más lenta indicará la velocidad del *clock* y por tanto ese tiempo se multiplicará por todas las etapas.



Para implementar un procesador con *pipeline* es necesario ir almacenando los resultados de las distintas etapas y propagándolos, de manera que en el siguiente ciclo se puedan utilizar para la siguiente etapa. Para lograr esto, se utilizan una serie de registros que se ubican entre las distintas etapas:

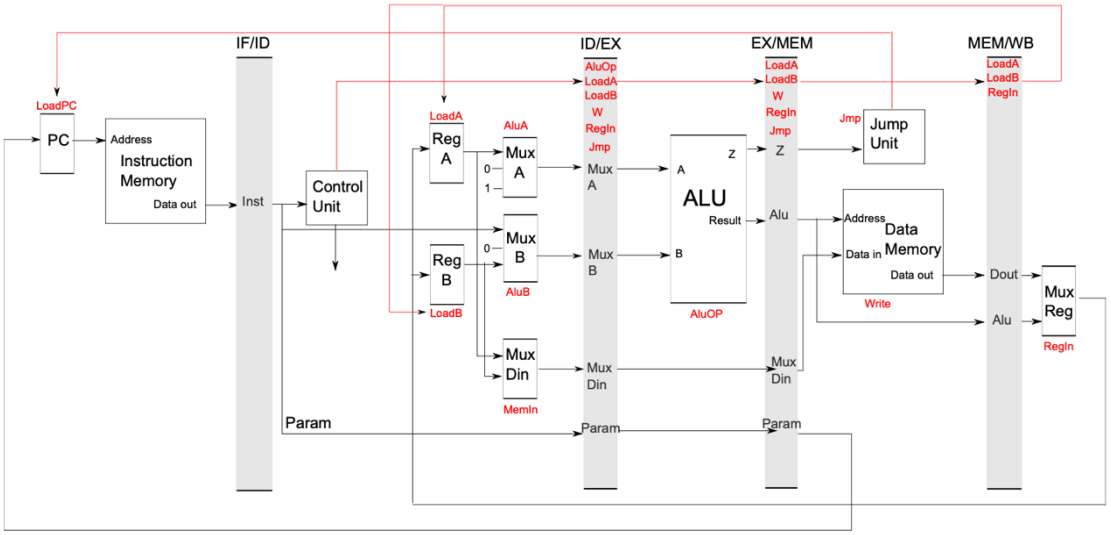
- **IF/ID**: Almacena la instrucción obtenida desde la memoria de instrucciones, incluyendo el *opcode* y el parámetro
- **ID/EX**: Almacena los dos parámetros de la ALU, el valor del registro seleccionado para ser copiado a memoria (si corresponde) y el parámetro de la instrucción.
- **EX/MEM**: Resultado de la ALU, el valor del registro seleccionado para escribir en memoria y el parámetro de la instrucción ambos que se propagaron de la etapa anterior.
- **MEM/WB**: Resultado de la ALU y el valor leído de memoria, los cuales serán utilizados en WB para escribir (si corresponde) en los registros.

- El resultado de la etapa WB se almacena en los registros A o B, por lo que no se requieren registros especiales.



Además de almacenar datos, los registros intermedios deben almacenar las señales de control, generadas en la etapa ID, que son necesarias en etapas siguiente.

- **ID/EX**: Almacena todas las señales de control necesarias para todas las siguientes etapas: operación de la ALU (`AluOP`), señal de escritura en Memoria (`W`), señales de carga en los registros (`LoadA` y `LoadB`), señal de selección de que se va a escribir en los registros (`RegIn`) y señal que indica si la instrucción actual es de salto (`Jmp`).
- **EX/MEM**: En la etapa EX se utilizó la señal `AluOp`, por lo que no es necesario propagarla, se propagan todas las demás.
- **MEM/WB**: En la etapa `MEM` se utilizan las señales `Jmp` y `W`, solo se requiere propagar la señal `RegIn` y las señales `LoadA` y `LoadB`.



Hazards

Al incorporar un *pipeline* en el computador, se generan una serie de problemas que son conocidos como *hazards*. Estos problemas ocurren por distintas razones, pero en general se deben al hecho de empezar a procesar una instrucción antes de que se haya procesado la anterior.

- **Hazards de datos**

Para entender los data hazards, revisemos el siguiente ejemplo de un programa en el que se ejecutan la instrucción ADD A,B y luego la instrucción AND B,A. Se puede observar que la segunda instrucción ocupa como parámetro el registro A. La primera instrucción va a modificar el valor de A, por lo que es necesario que haya terminado de ejecutarse para que las siguientes instrucciones tengan el valor correcto.

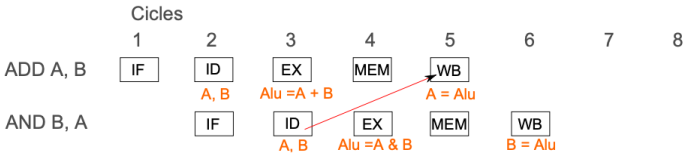


Figura 15: Ejemplo de data hazard, las flechas rojas indican las dependencias de datos

Como se observa en el diagrama, la primera instrucción ADD A,B va a actualizar el valor del registro A en su etapa de **WB**, es decir en el ciclo 5. El problema es que la siguiente instrucción AND B,A, necesita obtener el valor de A en su etapa **ID**, la que ocurre en el ciclo 3. La causa de este problema es que la secuencia de instrucciones del ejemplo presenta **dependencia entre los datos** que utiliza. Si la instrucción 2 no ocuparán el registro A, no habría problema, pero como depende de tener el valor que se genera en la primera instrucción, ocurre un error.

Si observamos el diagrama de la figura 16 observamos que aunque efectivamente es en la etapa WB de la instrucción 1 cuando el registro A se actualiza, el nuevo valor de A se generó antes, en la etapa EX, la cual ocurre en el ciclo 3 de ejecución. Para la instrucción 2 además, tenemos que aunque el valor de A se capta en la etapa ID, este se necesitará en la etapa EX, es decir en el ciclo 4. De esta forma, cuando el nuevo valor de A se necesita (ciclo 4), este ya se generó (ciclo 3), y está almacenado en los registros de la etapa ID/EX. Lo que necesitamos entonces es una forma de propagar el valor que se generó en la ALU al procesar la primera instrucción luego de la ejecución para que pueda ser ocupado como parámetro en la ejecución de las siguientes etapas EX, de la segunda instrucción (figura 16). Este mecanismo de propagación que se agrega a un *pipeline* se conoce como ***forwarding***.

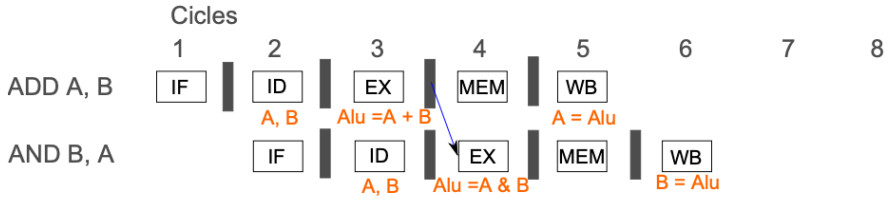
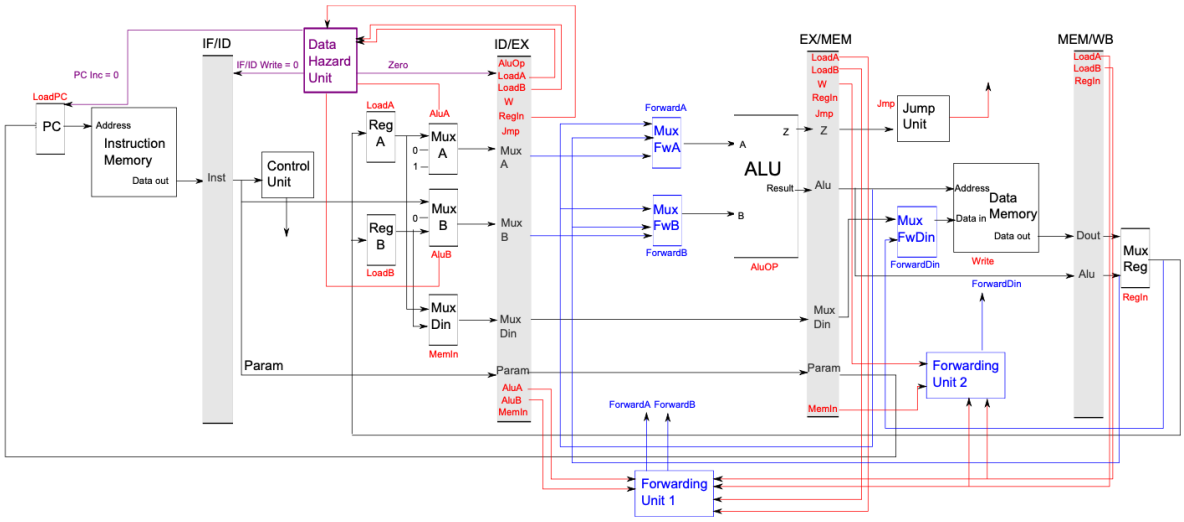


Figura 16: Ejemplo de data hazard, las flechas azules indican cuando hay que hacer forwarding

La idea de *forwarding* es, al estar ejecutando una instrucción, detectar si va a ocurrir un data *hazard*, y en ese caso realizar la propagación correspondiente. Esta detección debe realizarse en la etapa EX de la instrucción actual, antes de que se ejecute la operación de la ALU, que es lo primero que va a necesitar algún parámetro.



- **Hazards de control**

Un segundo tipo de *hazard* que ocurre en un procesador con *pipeline* corresponde a los *hazards* de control. Este tipo de hazard ocurre cuando hay una instrucción de salto: el problema con estas instrucciones es que dependiendo de si hay o no salto cambiará cual es la siguiente instrucción, pero con el diseño actual del procesador esto se sabe recién en la etapa MEM de la instrucción de salto, y por tanto a esas alturas ya entraron las siguientes instrucciones al *pipeline* (figura 26).

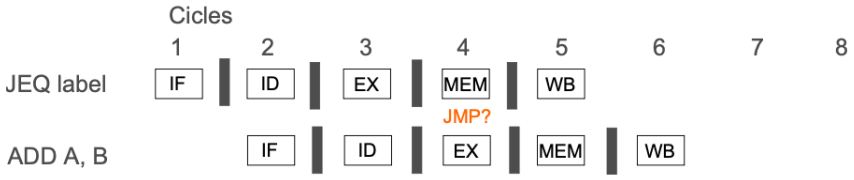


Figura 26: Hazard de control producido por una instrucción de salto: si el salto se efectúa la segunda instrucción no se debe ejecutar.

Para solucionar este tipo de hazards existen distintos mecanismos. Lo más simple es realizar ***stalling*** del pipeline los ciclos que sean necesarios para que la siguiente instrucción no entre hasta que se sepa si hay o no salto, y por tanto se sepa que instrucción corresponde. En el caso de este pipeline, esto involucra detener la CPU por 3 ciclos, es decir perder 3 ciclos por cada salto (figura 27).

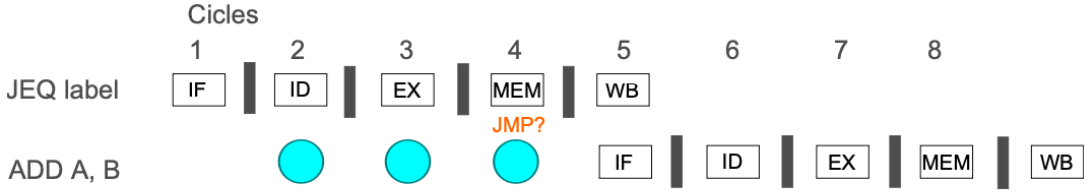


Figura 27: Stalling de 3 ciclos para prevenir control hazard

Es posible mejorar el manejo de los saltos, ocupando un mecanismo de **predicción de salto**. La idea de este mecanismo es que uno puede predecir si el salto va a ocurrir o no, y en caso de que la predicción sea correcta, no se van a perder ciclos de la CPU.

En caso de que la predicción sea incorrecta y efectivamente ocurra el salto, las siguientes tres instrucciones que ya entraron al pipeline no debe ser ejecutada. Para lograr esto se agrega una unidad de hazard de control en la etapa **`IF`** (figura 29), la cual cuando detecta que hay salto envía una señal de ***flush*** que avisa a las siguientes etapas del pipeline que no deben ejecutar las señales de control que se habían enviado (figura 30).

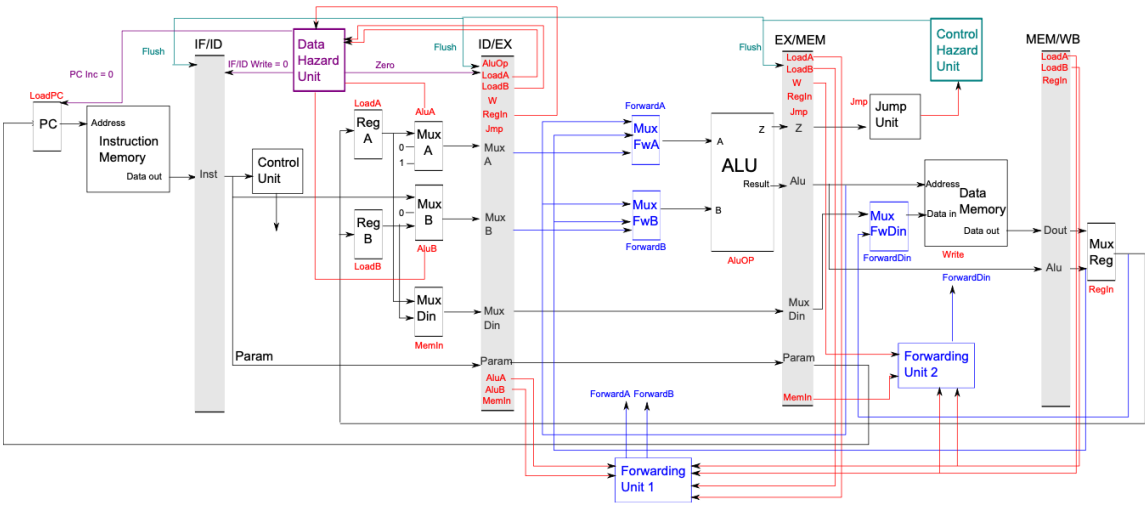


Figura 29: Soporte de hardware para control hazards.

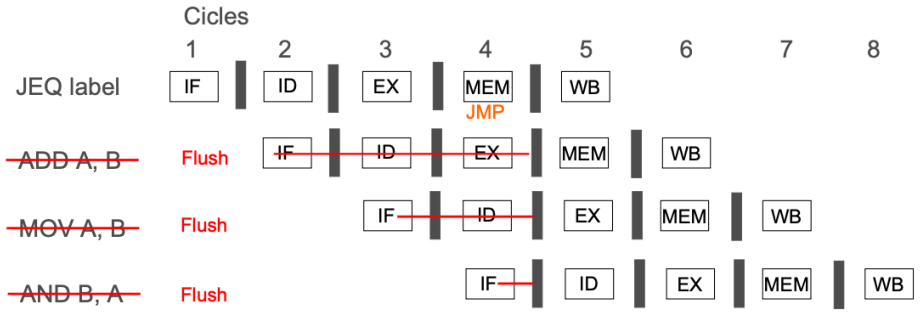


Figura 30: La unidad de control de hazard detecta que ocurrió un salto y envía una señal de flush para no ejecutar las siguientes etapas.

- **Hazards estructurales**

Los hazards estructurales ocurren cuando dos etapas del *pipeline* necesitan acceder a la misma unidad funcional. Un ejemplo ocurre en un computador *Von Neumann*, donde la memoria única de instrucciones y datos sería accedida tanto en la etapa `IF` como en la etapa MEM. Existen distintas formas de solucionar este tipo de hazards:

- Agregar unidades funcionales extras que permitan atender las dos etapas del ciclo sin contención. En el caso de las memorias, tener una caché *split* de primer nivel transforma la arquitectura del computador en Harvard y se soluciona el problema.

- Agregar «burbujas» mediante *stalling* que hagan que una de las instrucciones que quería ocupar la unidad funcional en un cierto ciclo espere hasta el siguiente.

Ejemplo

El siguiente ejemplo de código muestra la secuencia efectiva del *pipeline* de un programa en un computador con *pipeline*, que tiene *forwarding* (flechas azules en el diagrama), *stalling* por *software* (instrucciones NOP), predicción de salto asumiendo salto no realizado y *flushing* en caso de predicción
errónea (flecha roja).

```
DATA:
      n      2
      i      0
CODE:
      MOV B, (n)
      MOV A, (i)
art:  ADD  A, 1
      JNE start
      MOV  A,5
      ADD  A,B
      MOV B,3
```