Estructura y Tecnología de Computadores

Copyright © 2018 Juan Marín Noguera, juan.marinn@um.es.

Esta obra está bajo la licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons (CC-BY-SA 4.0). Para ver una copia de esta licencia, visite https://creativecommons.org/licenses/by-sa/4.0/.

Bibliografía:

- Estructura y Tecnología de Computadores, Manuel Eugenio Acacio Sánchez, Ricardo Fernández Pascual, Pilar González Férez & Alberto Ros Bardisa.

Capítulo 1

Sistemas secuenciales

Existen dos tipos de circuitos:

- **Combinacionales:** La salida depende únicamente de la entrada en ese momento. Se pueden representar mediante grafos acíclicos dirigidos.
- **Secuenciales:** La salida no depende solo de la entrada en ese momento sino también de su **estado**, que depende de las entradas anteriores.

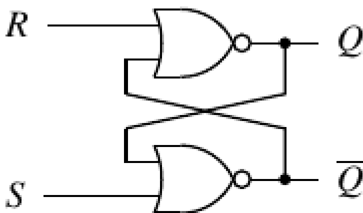
Para representar la evolución en el tiempo de un circuito se emplean **cronogramas**, diagramas con el tiempo en el eje horizontal y el valor lógico (0 ó 1) de ciertas señales (normalmente las entradas y salidas) en el eje vertical.

1.1. *Latches*

Un **biestable asíncrono**, **cerrojo** o ***latch*** es un circuito básico capaz de almacenar un bit. Existen dos tipos que se diferencian en su **ecuación característica** o **función de transición**, que define el valor de salida (Q^*) en función de su entrada y la salida anterior (Q).

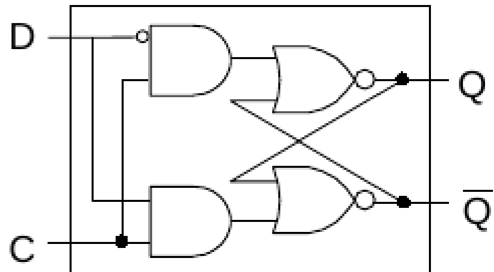
Tipo **S-R** (*Set-Reset*)

$Q^* = S + \overline{R} \cdot Q$



Tipo **D**

$Q^* = D \cdot C + Q \cdot \overline{C}$

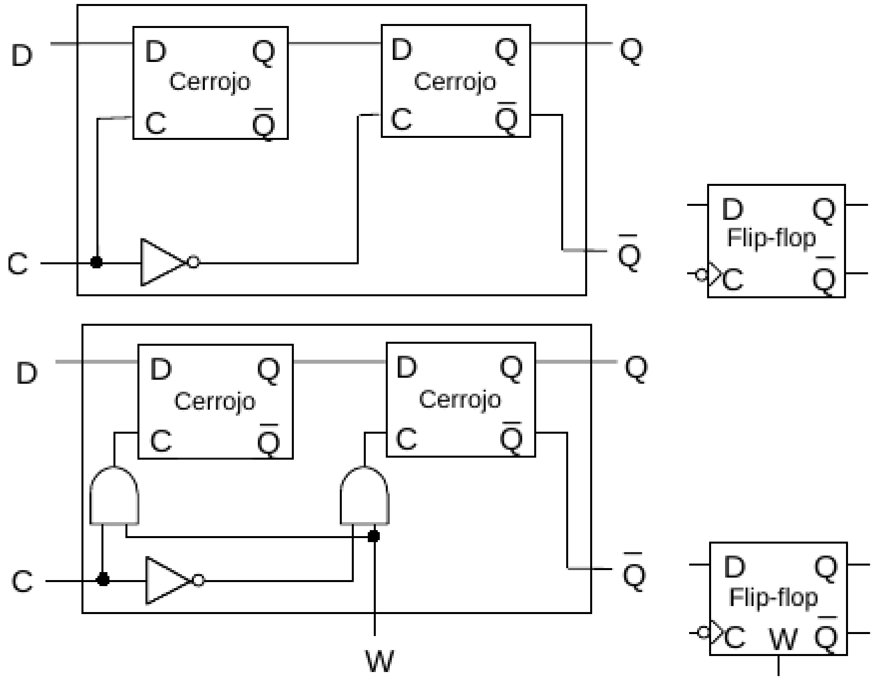


1.2. *Flip-flops*

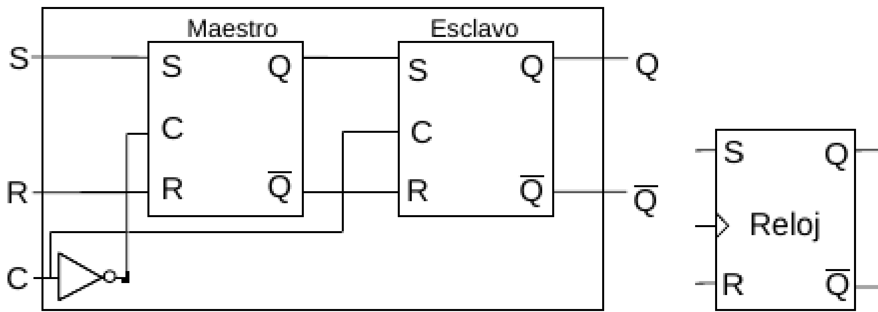
Un **biestable síncrono** o ***flip-flop*** también almacena un bit, pero las señales de entrada solo tienen efecto durante un instante de tiempo. Este depende de una **señal de reloj**, señal periódica encargada de determinar en qué momento el circuito será sensible a su entrada. Llamamos **flanco** a un cambio en la señal de reloj, que puede ser **ascendente** si es de 0 a 1 o **descendente** si es de 1 a 0. Aunque en los cronogramas representamos estas transiciones con líneas verticales, realmente no son instantáneas. Un *flip-flop* puede ser activo en flanco ascendente o en flanco descendente.

En general, los *flip-flops* están formados por dos *latches* en serie, donde el primero se llama **maestro** y el segundo **esclavo**. La entrada de reloj se indica con un triángulo, que tiene además un círculo si el *flip-flop* es activo en flanco descendente. Tipos:

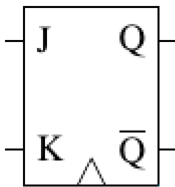
Tipo **D**: Puede tener o no una señal de control W, dependiendo de si queremos actualizar el valor en cada ciclo de reloj o no. $Q^* = D \cdot W + Q \cdot \overline{W}$.



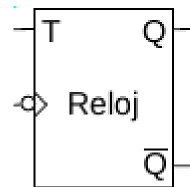
Tipo **S-R**: $Q^* = S + \overline{R} \cdot Q$.



Tipo **J-K**: $Q^* = J \cdot \overline{Q} + \overline{K} \cdot Q$. Similar al S-R con $J \equiv S$ y $K \equiv R$, pero si $J = K = 1$ el estado se alterna.



Tipo **T**: $Q^* = \overline{Q} \cdot T + Q \cdot \overline{T}$. Invierte su estado cuando su entrada valga 1.



1.3. Diseño de un circuito secuencial

Está formado por una serie de **entradas** y **salidas** digitales, así como una serie de bits que determinan su **estado actual** y dos funciones combinacionales:

- **Función de transición**: Determina el estado siguiente a partir del estado actual y la entrada.
- **Función de salida**: Determina la salida a partir del estado del circuito (circuito de **Moore**) y quizá también de las entradas (de **Mealy**).

Fases en el diseño:

1. **Especificación verbal**: Resumen con palabras del funcionamiento deseado.
2. **Especificación del autómata**: Se crea un diagrama de estados llamado «autómata finito determinista» (AFD), en el que se representan los posibles estados del sistema, la función de transición y la función de salida.
3. **Minimización del autómata**: Buscar el mismo comportamiento con menos estados, reduciendo la circuitería necesaria.
4. **Codificación de estados:** Asignar a cada uno de los M estados una combinación de $n = \lceil \log_2 M \rceil$ bits, que se almacenan en n biestables.
5. **Determinación de las funciones** de transición y de salida.
6. **Minimización de las funciones**, por ejemplo, mediante mapas de Karnaugh.

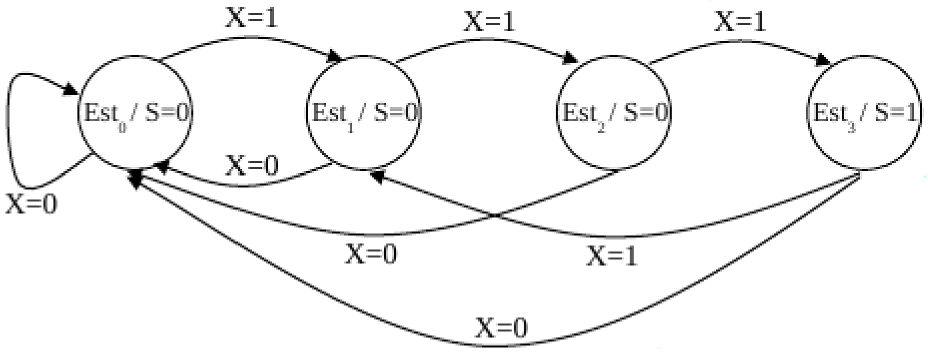


Figura 1.1: Ejemplo de AFD.

7. **Implementación del circuito**.

Debemos tener en cuenta que desde un cambio de señal de reloj hasta el siguiente, las señales de entrada de los *flip-flops* deben ser estables, por lo que la frecuencia de esta señal no debe ser mayor al retardo de los circuitos combinacionales, es decir, la máxima suma de los retardos de puertas lógicas que se usan en serie dentro de estos, incluyendo el retardo de otros biestables que son entradas de los circuitos combinacionales.

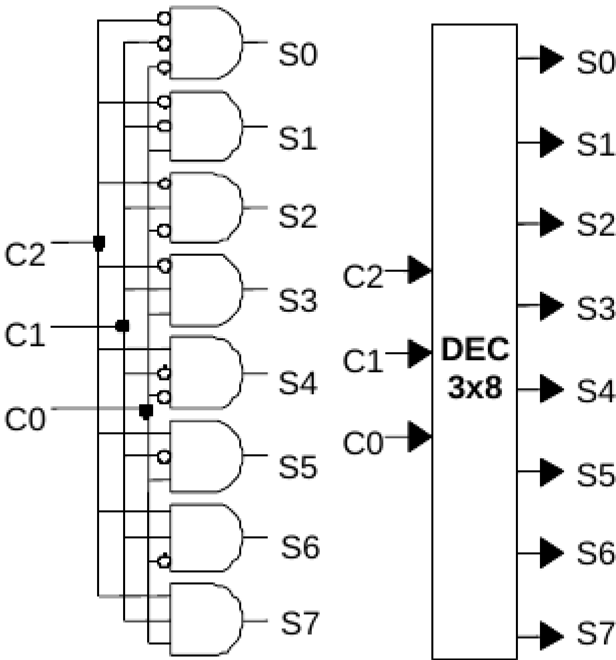
Capítulo 2

Componentes de un procesador

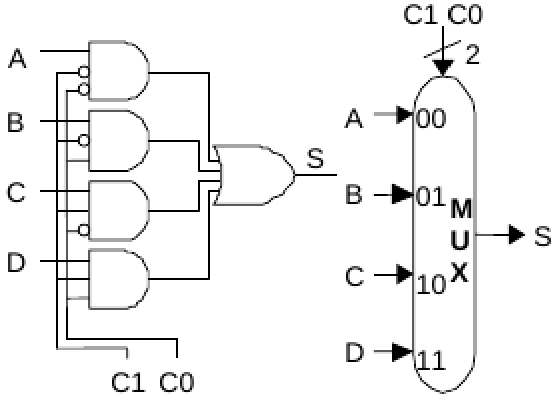
La **CPU** (*Central Processing Unit*) es un bloque lógico complejo que ejecuta instrucciones de un programa escrito de acuerdo a un juego de instrucciones o **ISA** (*Instruction Set Architecture*).

2.1. Componentes combinacionales sencillos

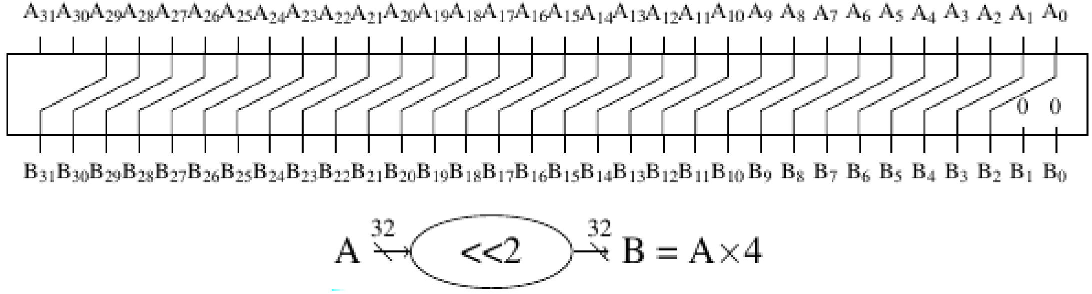
Decodificador Tiene n líneas de entrada y 2^n de salida, y para cada valor posible de la entrada, una y sólo una línea de salida tiene el valor 1.



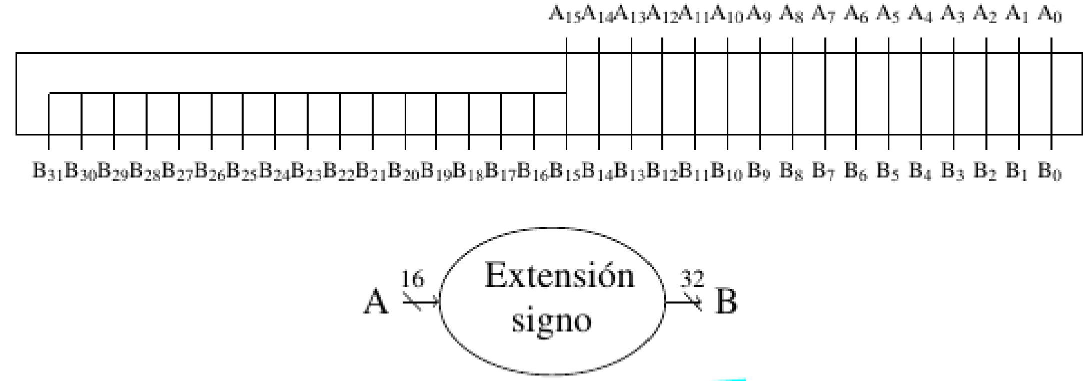
Multiplexor Tiene n líneas de entrada de control y 2^n de datos, y devuelve en su única línea de salida la línea de datos indicada por las de control.



Desplazador Recibe una palabra de n bits y devuelve otra de n bits resultado de desplazar a la izquierda o a la derecha la palabra de entrada un cierto número de bits, que suele ser fijo.

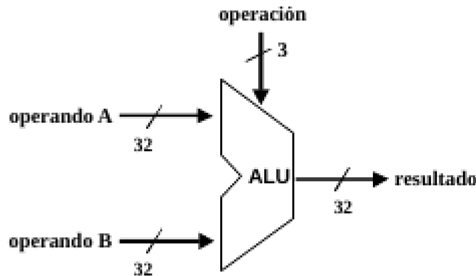


Extensor de signo Recibe un entero de m bits con signo y devuelve el mismo en n bits ($n > m$). En complemento a 2, esto equivale a replicar el bit de signo en los $n - m$ bits más significativos.

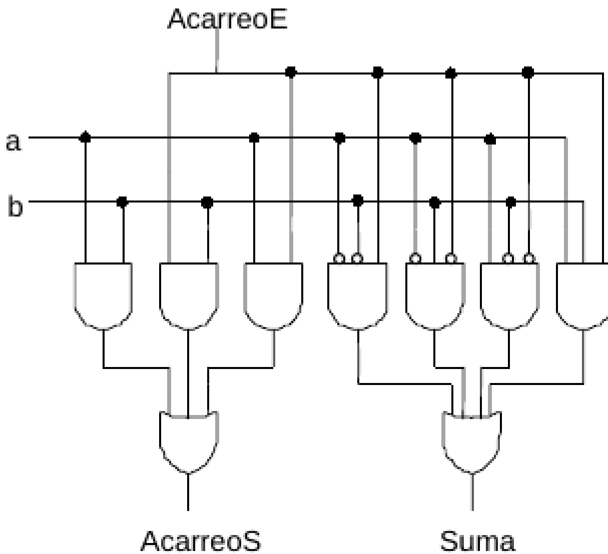


2.2. Unidad aritmético-lógica

Una **ALU** (*Arithmetic Logic Unit*) es un circuito combinacional capaz de realizar operaciones aritméticas y lógicas sobre operandos de entrada para generar una salida. La operación a realizar viene dada por unos bits de control. Como ejemplo mostramos una ALU capaz de realizar las operaciones de conjunción (AND) y disyunción (OR) bit a bit; suma y resta detectando desbordamientos, y comparación de dos números (SLT, *set less than*), comprobando si uno es mayor, menor o igual al otro.



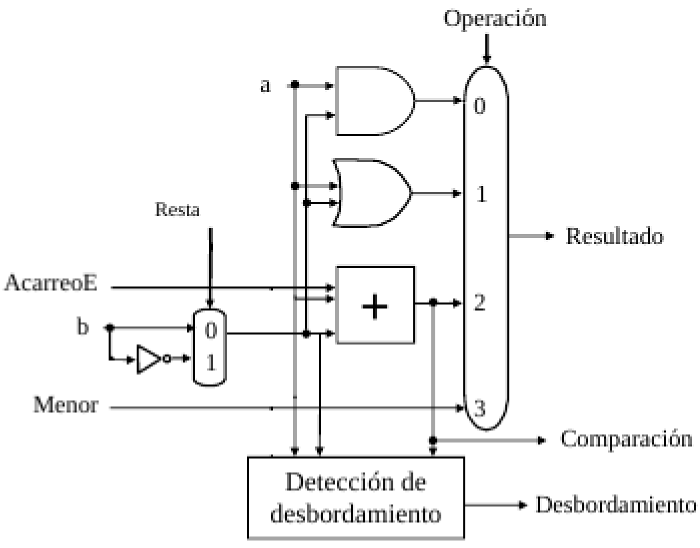
Un **semisumador** (*half adder*) recibe dos entradas y devuelve una salida con la suma, mientras que un **sumador completo** (*full adder*), que es lo que utilizamos, recibe tres entradas, dos con los sumandos y una de acarreo, y devuelve dos salidas para la suma y el acarreo.



Para implementar una ALU de 32 bits conectamos 32 ALUs de 1 bit conectando el acarreo de salida de un sumador con el de entrada del siguiente, obteniendo un **sumador con propagación del acarreo** (*ripple carry adder*). Esto significa que los bits se calculan uno por uno, lo que es ineficiente, por lo que en la práctica se usan **circuitos con acarreo anticipado** (*look-ahead carry*), que no veremos en este curso.

Restar equivale a sumar el minuendo con el opuesto del sustraendo, que se obtiene a su vez negando cada bit de este y sumando 1 al resultado (o estableciendo el acarreo de entrada del

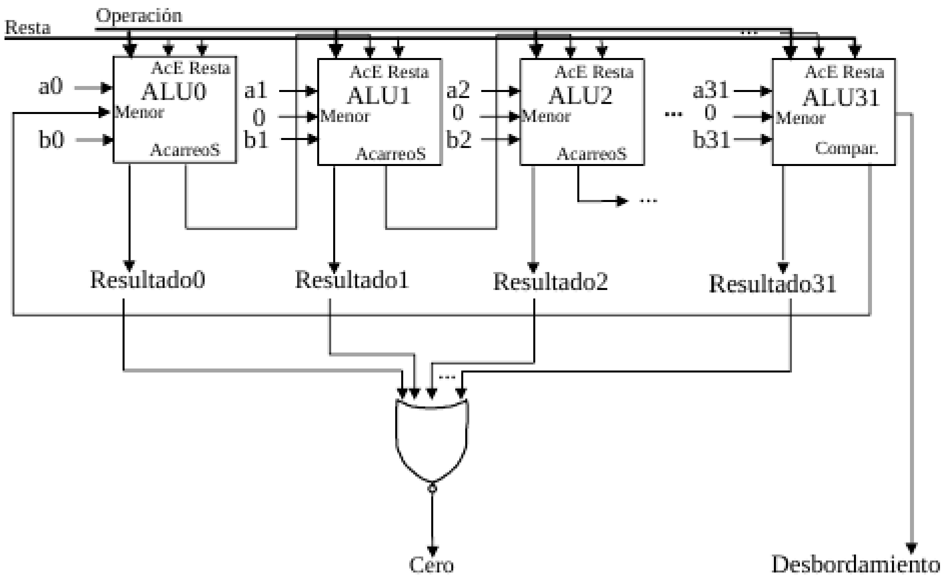
sumador menos significativo a 1). Por su parte, podemos implementar la comparación (SLT) restando ambos números y comprobando el bit de signo del resultado. Nos queda por tanto lo siguiente:

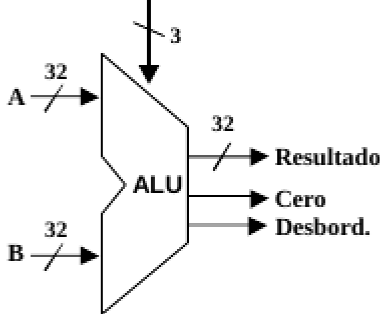


Aquí «Menor» es cero salvo para el bit menos significativo, que entonces es el resultado de la suma en el bit más significativo, indicado por «Comparación». Para el desbordamiento, nótese que ocurre si el resultado de la suma es negativo con ambos operandos positivos o positivo con ambos operandos negativos (si se trata de una resta, el signo del sustraendo se obtiene una vez éste ha sido negado), obteniendo el siguiente mapa de Karnaugh:

	A,B	00	01	11	10
Res.	0	0	0	1	0
	1	1	0	0	0

La ALU nos queda de la siguiente forma:

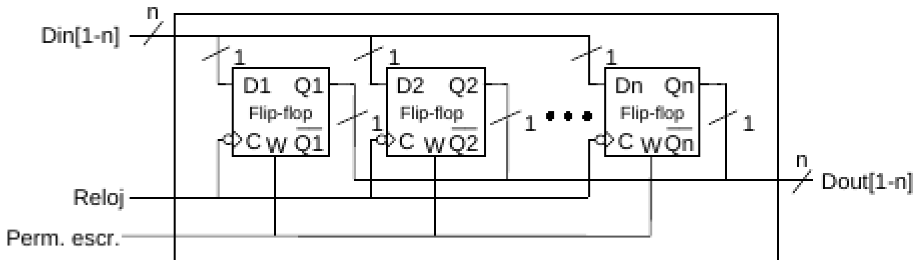




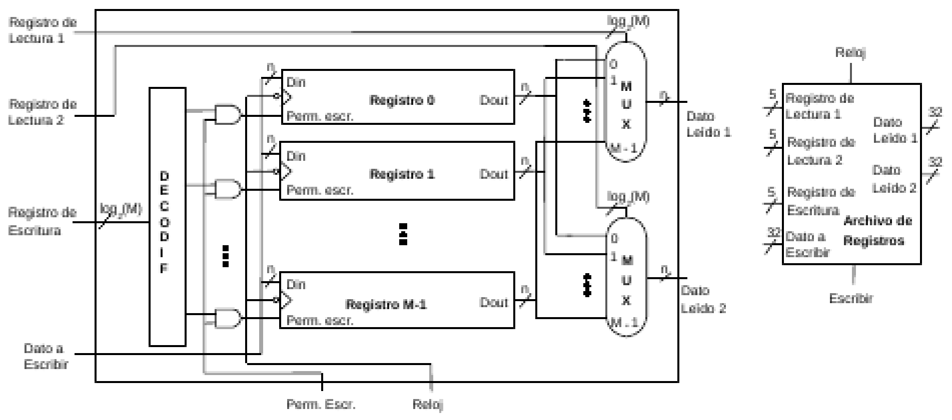
Op.	R	Op_1	Op_0
AND	0	0	0
OR	0	0	1
ADD	0	1	0
SUB	1	1	0
SLT	1	1	1

2.3. Registros

Un registro es una concatenación de *flip-flops* que comparten las señales de reloj y de permiso de escritura.

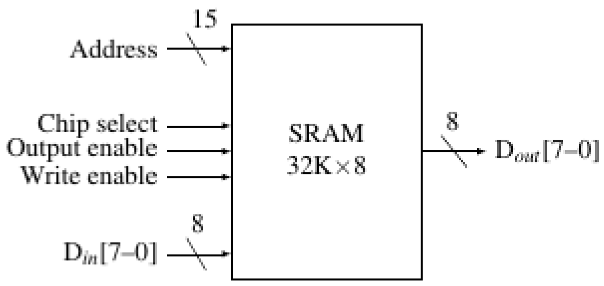


Un **banco de registros** es un conjunto de registros que pueden ser leídos y escritos selectivamente a través de **puertos de escritura** y **puertos de lectura**. Se usan multiplexores y decodificadores para seleccionar los registros.

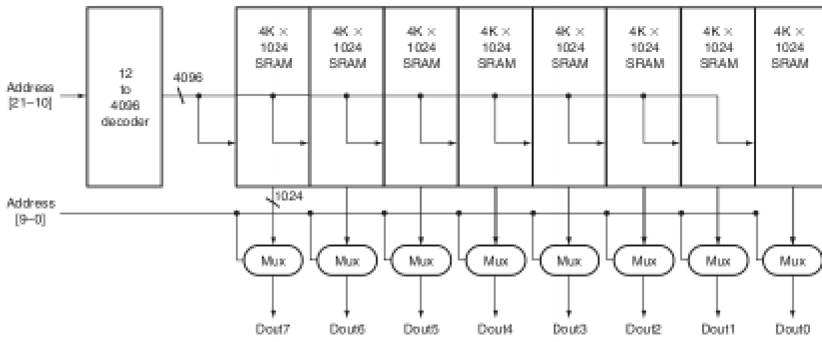


2.4. Memoria

Las memorias **SRAM** (*Static Random Access Memory*) son *arrays* de memoria usadas principalmente en caché por tener un tiempo de acceso muy corto. Llamamos **altura** al número de posiciones direccionables y **anchura** al número de bits por unidad de memoria.



Como direccionar una SRAM con un multiplexor sería demasiado costoso (por el tamaño), se usan líneas de salida compartidas (**líneas de bits**) que permiten que varias fuentes compartan una sola línea de datos, utilizando un ***buffer* triestado**. Este tiene dos entradas (*data* y *enable*) y una salida (*out*), y la salida es igual a *data* si *enable* está activa y en otro caso permanece con alta impedancia permitiendo el uso de la línea de salida a los otros *buffers*. Como todavía se requiere un decodificador a la entrada que sería demasiado grande, las memorias se organizan de forma bidimensional con decodificación en dos pasos.



Las memorias **DRAM** (*Dynamic Random Access Memory*) almacenan cada bit con un único transistor, con lo que son mucho más baratas y densas. Como la información se almacena en un condensador, este se va descargando, con lo que debe ser refrescada periódicamente, normalmente por un circuito en el propio chip, que lee su contenido y lo vuelve a escribir.

Capítulo 3

Diseño de un procesador

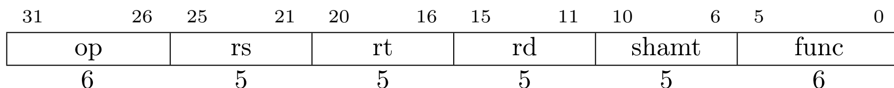
El **repertorio de instrucciones** o **ISA** de un procesador define el conjunto de instrucciones que implementa, y otros aspectos como el número y tipo de registros, los modos de direccionamiento y el manejo de excepciones. MIPS implementa un ISA RISC (*Reduced Instruction Set Computer*), que se caracteriza por tener un número de instrucciones relativamente pequeño, normalmente sencillas, que operan con datos en los registros, utilizan pocos modos de direccionamiento y se codifican todas con el mismo número de bits (en el caso de MIPS, 32 bits). En este capítulo implementamos una versión simplificada del ISA MIPS de 32 bits con unas pocas instrucciones de cada tipo.

3.1. Codificación de las instrucciones

MIPS dispone de 32 registros de propósito general con 32 bits cada uno, además de algunos específicos como el contador de programa (PC), y ve la memoria como un conjunto de celdas de 1 byte cada una con una dirección de 32 bits, permitiendo indexar hasta 4 GiB de memoria. Usa cinco modos de direccionamiento:

- **Registro:** El operando se encuentra en un registro, cuyo número está codificado en la instrucción en un campo de 5 bits ($2^5 = 32$ registros).
- **Base más desplazamiento:** Indica una dirección de memoria a leer o escribir mediante un registro base (codificado con 5 bits) al que se le suma una constante de 16 bits, codificada en el código de instrucción como un entero con signo en complemento a dos.
- **Inmediato:** Constante en un campo de 16 bits.
- **Relativo al PC:** Indica una dirección de destino de un salto mediante un campo de 16 bits en complemento a dos que indica el número de **palabras** (32 bits o 4 bytes) desde el valor actual del registro PC, que será la dirección de la instrucción inmediatamente posterior a la que se está ejecutando.
- **Pseudodirecto:** Indica una dirección de destino mediante un campo de 26 bits. Como las direcciones de memoria son de 32, a este se le añaden al final 2 bits a 0 (porque las instrucciones están alineadas a la palabra) y al principio los 4 bits más significativos del PC.

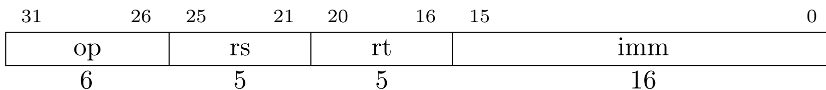
3.1.1. Formato de instrucción R



Donde **op** es el código de operación, que aparece en todos los formatos de instrucción, **rs** y **rt** son los registros fuente, **rd** el registro de destino y **func** indica a la ALU qué función debe realizar. **shamt** es el tamaño de desplazamiento en las instrucciones de desplazamiento y rotación, y vale 0 en el resto.

Este formato se usa para operaciones aritmético-lógicas con modo de direccionamiento registro ($\boldsymbol{op} = 0$), cuyo formato es `func $rd, $rs, $rt`, y de las cuales implementaremos `and` ($\boldsymbol{func} = 44|_8$), `or` ($\boldsymbol{func} = 45|_8$), `add` ($\boldsymbol{func} = 40|_8$), `sub` ($\boldsymbol{func} = 42|_8$) y `slt` ($\boldsymbol{func} = 52|_8$), que ya implementamos en la ALU del capítulo anterior.

3.1.2. Formato de instrucción I



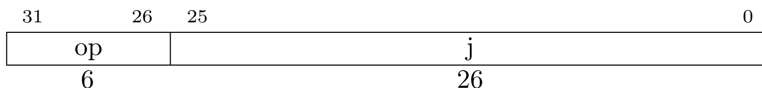
Usado para:

- Instrucciones aritmético-lógicas con un operando constante, por direccionamiento inmediato.

- Instrucciones de acceso a memoria, por direccionamiento base más desplazamiento. El formato es `instr $rt, imm($rs)`, y veremos `lw` ($\boldsymbol{op} = 35$) y `sw` ($\boldsymbol{op} = 43$), que cargan una palabra de una dirección de memoria a un registro y viceversa, respectivamente.

- Instrucciones de salto condicional, por direccionamiento relativo a PC. El formato es `instr $rs, $rt, label`, con `label` $= PC + 4\boldsymbol{imm}$, y veremos `beq` ($\boldsymbol{op} = 4$), que salta a una dirección si los dos registros contienen lo mismo.

3.1.3. Formato de instrucción J



Donde **j** almacena la dirección de destino por direccionamiento pseudodirecto. Lo usa la instrucción de salto incondicional `j` ($\boldsymbol{op} = 2$), cuyo formato es `j label`.

3.2. Modelo del tiempo de ejecución

$$T_{CPU} = N_{inst} \cdot CPI \cdot T_{ciclo}$$

Donde T_{CPU} es el tiempo que tarda la CPU en ejecutar un programa, N_{inst} es el número de **instrucciones dinámicas** que se ejecutan (cada **instrucción estática** del código se cuenta

tantas veces como se ejecuta), CPI es el promedio del número de ciclos que tarda en ejecutarse cada instrucción y T_{ciclo} es la duración del ciclo de reloj (tiempo entre dos flancos activos), que debe ser lo suficientemente largo para permitir que se estabilicen todas las señales de entrada a los elementos secuenciales.

En una implementación **monociclo**, todas las instrucciones tardan exactamente un ciclo en ejecutarse. El tiempo de ciclo tiene que ser suficiente para ejecutar la instrucción más larga, por lo que sobrará tiempo para las cortas, reduciendo el rendimiento. Construiremos un procesador **multiciclo**, en el que las instrucciones se dividen en pasos y cada uno ocupa un ciclo, y el tiempo de ciclo es el necesario para ejecutar el paso más largo. También se pueden realizar implementaciones con $CPI < 1$ mediante técnicas para ejecutar varias instrucciones a la vez, que no veremos aquí.

3.3. El camino de datos

Un procesador está formado por un **camino de datos**, donde se encuentran los elementos que realizan el trabajo indicado, y una **unidad de control**, que gestiona el camino de datos.

Para el camino de datos, tomamos como componentes principales la memoria, el banco de registros y la ALU, y suponemos que solo estos conllevan un retardo significativo. Entonces toda operación con uno de estos componentes consume un ciclo, y la salida de estos debe almacenarse en registros auxiliares para ser usada en el ciclo siguiente. Con esto, nuestro camino de datos tendrá los siguientes componentes:

- **Memoria:** Acepta direcciones de 32 bits y permite leer o escribir una palabra de 4 bytes en cada ciclo. Tiene dos señales de control para habilitación de lectura y escritura, así como puertos de entrada para la dirección y el dato a escribir y de salida para el dato leído.

- **Banco de registros (Reg):** Banco de 32 registros capaz de leer dos registros y escribir otro en el mismo ciclo. El registro 0 es virtual y contiene siempre el valor 0, por lo que toda escritura en este es ignorada.

- **ALU:** La que vimos en el capítulo anterior.

- **Contador de programa (PC):** De 32 bits, con señal de habilitación de escritura.

- **Registro de instrucción (IR):** De 32 bits con habilitación de escritura, almacena la instrucción que se está ejecutando actualmente, y tiene varios puertos de salida para los distintos campos.

- **Registro de datos de memoria (MDR):** De 32 bits sin habilitación de escritura, almacena un valor leído de memoria.

- **Registros A y B:** De 32 bits sin habilitación de escritura, almacenan los valores leídos del banco de registros.

- **Registro ALUOut:** De 32 bits sin habilitación de escritura, almacena la salida de la ALU.

- Algunos desplazadores y extensores de signo, así como multiplexores para permitir distintas conexiones entre unidades funcionales.

Veamos ahora la descomposición de instrucciones en pasos. Para esta parte usaremos **lenguaje de transferencia entre registros** o **RTL** (*Register Transfer Language*). La transferencia de un registro o resultado de una operación a un registro se escribe como `A <- B`, donde `B` indica una operación y `A` es el registro al que se transfiere el resultado. Si una sentencia se debe ejecutar sólo bajo cierta condición `C`, que puede incluir operadores booleanos, se indica con `C: A <- B`. Finalmente, para indicar que varias sentencias se ejecutan en paralelo, se separan por comas.

Los dos primeros pasos son comunes a todas las instrucciones:

1. Lectura de instrucción desde la memoria (memoria) y cálculo de la dirección de la instrucción siguiente (ALU).
   `IR <- Memoria[PC], PC <- PC + 4`

2. Decodificación de la instrucción, lectura de operandos (banco de registros) y cálculo de la dirección de destino de salto condicional (ALU).
   `A <- Reg[IR[25-21]], B <- Reg[IR[20-16]],`
   `ALUOut <- PC + sign_extend(IR[15-0]) <<2`
   Como hasta que no acaba el segundo paso no se ha decodificado la instrucción, el resto de acciones se hacen de manera **especulativa**, por si resultaran útiles luego, pues las unidades funcionales necesarias están desocupadas.

A continuación, para una instrucción aritmético-lógica:

3a. Realización de la operación (ALU).
    `ALUOut <- A func B`

4a. Escritura del resultado (banco de registros).
    `Reg[IR[15-11]] <- ALUOut`

Para `lw`:

3b. Cálculo de la dirección de memoria (ALU).
    `ALUOut <- A + sign_extend(IR[15-0])`

4b. Lectura del dato de memoria (memoria).
    `MDR <- Mem[ALUOut]`

5. Escritura del dato leído (banco de registros).
   `Reg[IR[20-16]] <- MDR`

Para `sw`:

3b. Cálculo de la dirección de memoria (ALU).

4c. Escritura del dato en memoria (memoria).
    `Mem[ALUOut] <- B`

Para `beq`:

3c. Comprobación de la condición del salto (ALU) y actualización del contador de programa si procede.

```
A=B: PC <- ALUOut
```

Para `j`:

3d. Actualización del contador de programa.

```
PC <- (PC[31-28] <<28) | (IR[25-0] <<2)
```

Nos queda por tanto lo siguiente:

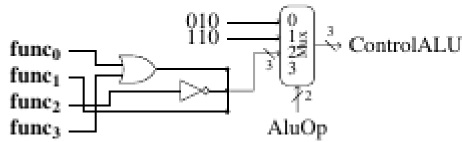
```
T1: IR <- Mem[PC], PC <- PC + 4
T2: A <- Reg[IR[25-21]], B <- Reg[IR[20-16]],
        ALUOut <- PC + sign_extend(IR[15-0]) << 2
T3 && op=0: ALUOut <- A func B
T3 && (op=35 || op=43): ALUOut <- A + sign_extend(IR[15-0])
T3 && (op=4) && (A=B): PC <- ALUOut
T3 && (op=2): PC <- (PC[31-28] << 28) | (IR[25-0] << 2)
T4 && (op=0): Reg[IR[15-11]] <- ALUOut
T4 && (op=35): MDR <- Mem[ALUOut]
T4 && (op=43): Mem[ALUOut] <- B
T5 && (op=35): Reg[IR[20-16]] <- MDR
```

Las conexiones que necesitamos dependen del ciclo actual y la instrucción que se esté ejecutando, por lo que tenemos que añadir algunos multiplexores en algunos puertos de entrada para seleccionar:

Nombre	Puerto	Unidad	V.	Entrada	Pasos	V.	Entrada	Pasos
IoD	Dirección	Memoria	0	PC	1	1	ALUOut	4b,4c
DestReg	Reg. a escr.	B. de regs.	0	**rt**	5	1	**rd**	4a
MemAReg	Dato a escr.	B. de regs.	0	ALUOut	4a	1	MDR	5
SelALUA	1er op.	ALU	0	PC	1,2	1	A	3a,3b,3c
SelALUB	2° op.	ALU	00	B	3a	01	Const. 4	1
			10	**imm**	3b	11	**imm***4	2
PCSrc	Entrada	PC	00	ALU	1	01	ALUOut	3c
			10	...**j**...	3d			

3.4. La unidad de control

Debemos asegurar que el valor de las señales de control de los multiplexores y de todas las unidades funcionales sea el adecuado. Primero haremos un circuito de control de la ALU, que debe realizar una suma en los pasos 1, 2 y 3b, una resta en el paso 3c y una función en 3a que depende de los bits del campo **func**. Para ello diseñamos un circuito combinacional con dos bits de entrada (`ALUOp`) además de los seis del campo **func** y tres bits de salida correspondientes a `ALUCtl`, como se muestra en la figura. No necesitamos los dos bytes más significativos de `func`, pues son iguales en las cinco operaciones.



Hecho esto, las señales que debe controlar la unidad de control principal son: las señales de control de los multiplexores indicados, incluyendo `ALUOp`; las señales `MemR` y `MemW` de habilitación de lectura y escritura de memoria, y la señales de habilitación de escritura `WrtIR`, `WrtPC`, `WrtPCCond` (similar pero solo escribe si la señal «Cero» de la ALU está activa, para implementar la condición `A=B`) y `WrtReg`.

El valor de las señales de control depende sólo del paso actual, mientras que la transición de un paso a otro depende de la instrucción en ejecución, por lo que podemos implementar la unidad de control como un autómata de Moore cuya entrada es el campo **op** de la instrucción en IR y cuya salida son los valores de las señales de control.

1
IoD = 0
LeerMem = 1
EscrIR = 1
SelALUA = 0
SelALUB = 01
ALUOp = 00
FuentePC = 00
EscrPC = 1

2
SelALUA = 0
SelALUB = 11
ALUOp = 00

op = 35 ó
op = 43

3b
SelALUA = 1
SelALUB = 10
ALUOp = 00

op = 43

4c
IoD = 1
EscrMem = 1

op = 2

op = 4

op = 0

op = 35

3d
FuentePC = 10
EscrPC = 1

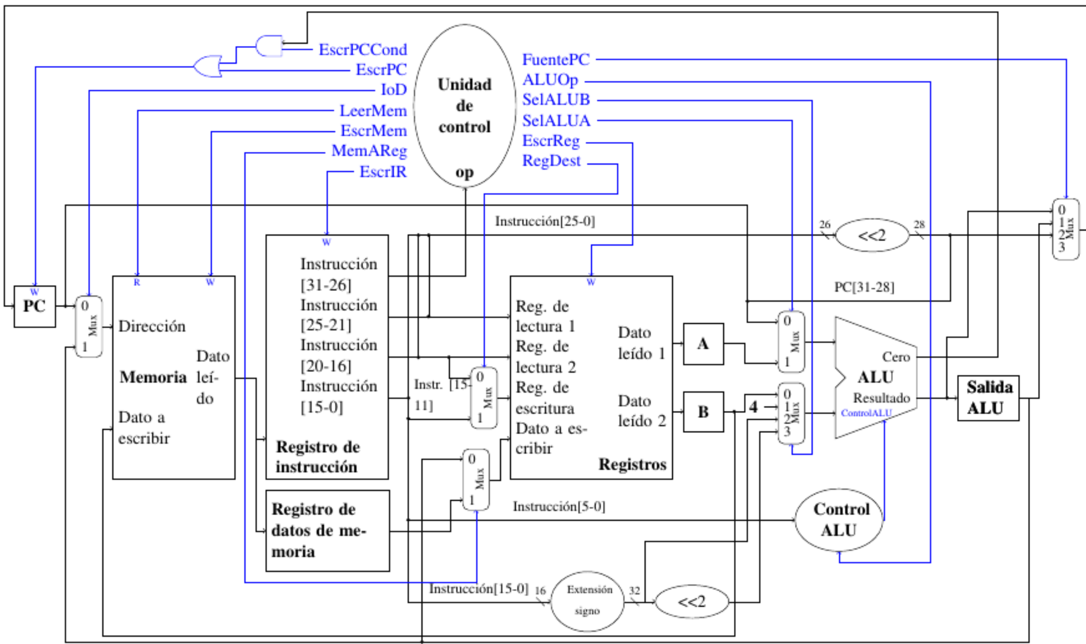
3c
SelALUA = 1
SelALUB = 00
ALUOp = 01
FuentePC = 01
EscrPCCond = 1

3a
SelALUA = 1
SelALUB = 00
ALUOp = 10

4b
IoD = 1
LeerMem = 1

4a
RegDest = 1
MemAReg = 0
EscrReg = 1

5
RegDest = 0
MemAReg = 1
EscrReg = 1



3.5. Metodología para añadir instrucciones

1. **Análisis**

   a) **Especificación semántica precisa:** Traducir la descripción verbal de la instrucción a notación RTL.

   b) **Identificación del trabajo:** Identificar qué acciones realiza cada unidad funcional.

   c) **Establecimiento del orden de precedencia:** Ver qué relaciones de dependencia existen entre las acciones.

2. **Diseño**

   a) **Definición de la codificación:** Asignar una codificación ajustándose a uno de los tres formatos de instrucción existentes (salvo que sea imposible), de forma que se pueda diferenciar la instrucción de las ya existentes. Para decidir la colocación de los operandos, conviene tener en cuenta el uso que se va a hacer de ellos y las conexiones ya disponibles, minimizando el número de cambios a realizar.

   b) **División del trabajo en ciclos:** Respetando las dependencias, sin usar la misma unidad dos veces en el mismo ciclo, e intentando aprovechar las instrucciones que ya se realizan en los dos primeros ciclos.

   c) **Extensión del camino de datos:** Detallar las modificaciones a realizar en este para permitir la realización de las acciones indicadas.

   d) **Extensión del control:** Indicar los nuevos estados, el paso de uno a otro y el valor de las señales de control en cada uno.

Capítulo 4

Gestión de caché

Hoy en día (a precio razonable) podemos construir memorias de gran capacidad pero lentas, o memorias rápidas pero con poca capacidad. Para crear la ilusión de que tenemos una memoria con ambas características, combinamos tipos de memoria con distintas velocidades y tamaños en una **jerarquía de memoria**, formada normalmente por los siguientes componentes, de mayor a menor cercanía a la CPU:

- Registros de la CPU.
- **Memoria caché** (SRAM). Suele haber 3 niveles (llamados L1–L3), donde el L3 puede estar fuera de la CPU (los demás están dentro) y L1 es el de menor capacidad pero menor tiempo de acceso, y por tanto el más cercano a la CPU.
- Memoria principal (**SDRAM**, *Synchronous DRAM*).
- Almacenamiento secundario local (discos SSD o magnéticos). Los **discos SSD** (*Solid State Drive*) usan memoria flash para almacenar los datos y por tanto son más rápidos que los **discos duros** o **magnéticos**, pues no presentan limitaciones mecánicas, si bien quedan lejos de las memorias RAM.
- Almacenamiento secundario remoto (sistemas de ficheros distribuidos, servidores, etc.).

El objetivo es proporcionar la máxima capacidad de memoria con la tecnología más barata pero con un tiempo medio de acceso similar al de la tecnología más rápida. Por lo general, los datos solo se transfieren entre niveles adyacentes.

4.1. Funcionamiento

Principio de localidad: en un momento concreto, los programas acceden a una parte relativamente pequeña de su espacio de direcciones. Tipos:

- **Localidad temporal**: Si se consulta un dato, probablemente será consultado próximamente. En los programas aparecen multitud de bucles, por lo que se accederá repetidamente a instrucciones y datos.

- **Localidad espacial**: Si se consulta un dato, probablemente serán consultados otros cercanos. A las instrucciones se suele acceder secuencialmente, así como a los elementos de una tabla.

Para aprovechar la localidad espacial, la información se transfiere entre niveles adyacentes en **bloques**, cuyo tamaño depende de los niveles. El **tamaño de bloque** es el número de bytes que contiene, y para las cachés suele ser 64. Decimos que hay un **acierto** (*hit*) cuando la información pedida por el procesador se encuentra en el nivel superior, y un **fallo** (*miss*) cuando no.

La **tasa de aciertos** (*hit rate*) es la fracción de accesos a memoria que son aciertos, y se usa como medida del rendimiento, mientras que la **tasa de fallos** (*miss rate*) es 1 menos la tasa de aciertos. El **tiempo de acierto** es el tiempo necesario para acceder al nivel superior de la memoria, incluyendo el necesario para determinar si el acceso es un acierto o un fallo, y la **penalización por fallo** es el tiempo necesario para reemplazar un bloque del nivel superior por otro del nivel inferior.

Las direcciones de memoria se asignan a bytes individuales, lo que llamamos **dirección de byte** (D_{byte}). Como las palabras (en general) son de varios bytes, cada palabra tiene varias direcciones de byte, a las que asignamos una **dirección de palabra** con $D_{palabra} = \left\lfloor \frac{D_{byte}}{T_{palabra}} \right\rfloor$, donde $T_{palabra}$ es el número de bytes de la palabra, y el resto de esta división es el desplazamiento de byte dentro de la palabra. Igualmente, podemos asignar a cada bloque una **dirección de bloque**, de forma que $D_{bloque} = \left\lfloor \frac{D_{byte}}{T_{bloque}} \right\rfloor$, siendo T_{bloque} el tamaño del bloque, y el resto de esta división es el desplazamiento de byte dentro del bloque. Tanto el tamaño de palabra como el de bloque son potencias de 2, por lo que el cálculo de las direcciones y desplazamientos es trivial.

4.2. Estructura

Una caché esta organizada en una serie de **conjuntos** de bloques o **huecos**, y una serie de **vías**, que son el número de bloques por conjunto. Llamamos **asociatividad** de la caché al número de vías que contiene. A cada bloque le corresponde un conjunto, dado por el resto de la dirección de bloque entre el número de conjuntos, y dentro de este puede ocupar cualquiera de los huecos.

Una mayor asociatividad aumenta la tasa de acierto por tener más libertad a la hora de elegir qué bloques quitar, pero también aumenta el número de comparaciones que hay que realizar, por lo que aumenta el coste al necesitar más comparadores, así como el tiempo de acierto.

Una caché se dice **de correspondencia directa** si tiene sólo una vía (un bloque por conjunto), y **totalmente asociativa** si sólo tiene un conjunto. A cada posición de la caché se le añade una **etiqueta** (*tag*) dada por $E = \left\lfloor \frac{D_{bloque}}{N_S} \right\rfloor$, donde N_S es el número de conjuntos. Además, es necesario un **bit de validez** en cada hueco que indique que el bloque tiene información válida.

4.3. Políticas de escritura

- **Escritura directa** (*write through*): Las escrituras se hacen a la vez en la caché y en memoria. Si el bloque no está en caché, se suele escribir directamente la palabra en memoria principal (*no write allocate*), si bien también se puede traer el bloque a la caché (*write allocate*), pero esto es mucho menos común.

  Esta política es más fácil de implementar y tiene la ventaja de que los fallos son menos costosos. Sin embargo, en la práctica es necesario un ***buffer* de escrituras** que almacene los datos para ser escritos de forma que el procesador no tenga que detenerse hasta que acabe la escritura, salvo que el *buffer* esté lleno.

- **Pos-escritura** (*write back*): Las escrituras se hacen sólo en la caché, y sólo se actualiza la información en memoria al sacar el bloque de la caché. Es necesario un **bit de modificación** o **de sucio** (*dirty bit*) en cada bloque, y si el bloque no está en caché se debe traer a caché.

  Esta política tiene la ventaja de que en caso de acierto se puede escribir más rápidamente, y que múltiples escrituras en un bloque requieren una sola escritura en memoria, pudiendo hacer uso de **escritura en ráfaga** para conseguir mayor ancho de banda.

4.4. Políticas de reemplazo

En una caché asociativa (con más de una vía), al traer un bloque se puede poner en cualquier hueco del conjunto, y si el conjunto está lleno (como ocurre normalmente) hay que elegir qué bloque sacar de la cache. Para ello hay varias políticas:

- **Aleatoria**: Se elige un bloque al azar. Es la más sencilla de construir.
- **LRU** (*Least Recently Used*, Menos Recientemente Usado): Se elige el bloque que haya estado más tiempo sin ser accedido. Consigue menores tasas de fallo que el aleatorio, pero al aumentar la asociatividad se vuelve demasiado costoso, pues necesita mantener mucha información.
- **Pseudo-LRU**: Esquemas que «imitan» al LRU pero con implementación más sencilla. Por ejemplo, el que utiliza un **bit de uso** o **de referencia** en cada bloque, que se pone a cero al traer el bloque y a 1 cada vez que se accede a él, volviéndose a poner a cero en todos los bloques periódicamente. A la hora de elegir un bloque que sustituir, se prefiere uno que tenga a 0 el bit de uso.
- **NRU** (*Not Recently Used*): Para una caché de asociatividad 2, se puede implementar el LRU con un sólo bit por conjunto.

4.5. Tamaño real y rendimiento

Si N_S es el número de conjuntos de la caché, A es la asociatividad, T_{bloque} es el tamaño de bloque, W_{dir} el número de bits de las direcciones de memoria y $N_{bits-control}$ el número de bits de control necesarios por bloque (de 1 a 3 según lo visto y puede que algunos más en una caché real), el tamaño útil de la caché en bytes será $T_{útil} = N_S A T_{bloque}$.

Por su parte, el tamaño total en bits será $T_{total} = N_S T_{conjunto}$, con $T_{conjunto} = AT_{entrada}$, $T_{entrada} = N_{bits-control} + W_{etiqueta} + 8T_{bloque}$ y $W_{etiqueta} = W_{dir} - \log_2 N_S - \log_2 T_{bloque}$. En resumen,

$$T_{total} = N_S A(8T_{bloque} + N_{bits-control} + W_{dir} - \log_2 N_S - \log_2 T_{bloque})$$

Para hallar el rendimiento de la caché usamos que $T_{ejec} = T_{CPU} + T_{bloqueo}$, siendo T_{CPU} el tiempo de ejecución normal de CPU, contando aciertos en accesos de memoria, y $T_{bloqueo}$ el bloqueo debido a fallos de caché. Así, $T_{bloqueo} = N_{accesos} T_F P_F$, donde $N_{accesos}$ es el total de accesos, T_F la tasa de fallos y P_F la penalización por fallos.

Ahora bien, normalmente un procesador tiene cachés separadas para datos e instrucciones para evitar que los datos «expulsen» instrucciones y viceversa y mejorar así el rendimiento. Por tanto, si la caché de instrucciones tiene tasa de fallos T_{Fi} y penalización P_{Fi} y la de datos tiene tasa de fallos T_{Fd} y penalización P_{Fd}, y D/I es la tasa de acceso a datos por instrucción (accesos a datos por número de instrucciones), nos queda que

$$T_{bloqueo} = N_{instrucciones}(T_{Fi} P_{Fi} + D/I \cdot T_{Fd} P_{Fd})$$

donde $N_{instrucciones}$ es el número de instrucciones que se ejecutan.

Capítulo 5

Memoria virtual

La **memoria virtual** es una técnica consistente en usar la memoria principal como caché para almacenamiento secundario, normalmente discos duros. Objetivos:

- **Proporcionar un gran espacio de direcciones:** Anteriormente, si un programa era demasiado grande, el programador debía dividir el programa en segmentos independientes (*overlays*) cargados y liberados de memoria por el propio programa, lo que dificultaba la programación. Además, debía haber un hueco de memoria libre contigua para cargar estos segmentos, pero esta podía estar fragmentada en varios huecos demasiado pequeños. Con memoria virtual, el programador trabaja como si hubiera una gran memoria sólo para el programa.
- **Permitir la compartición segura y eficiente de memoria entre varios programas:** La memoria principal sólo tiene que contener lo que se ejecuta en un momento dado, por lo que se aprovecha mejor, y consigue que un proceso sólo pueda leer y escribir la memoria que tiene asignada.

El sistema de memoria virtual crea un **espacio de direcciones virtuales** para cada proceso, de forma que la CPU produce direcciones virtuales y la **MMU** (*Memory Management Unit*) realiza la **traducción de direcciones** (*address translation*) a **direcciones físicas**. Un bloque de memoria virtual es una **página**, un acierto es un **acierto de página** y un fallo es un **fallo de página**.

Las páginas deben ser lo bastante grandes para amortizar el tiempo de acceso a disco, normalmente 4 KiB para páginas pequeñas y tamaños mucho mas grandes (de megabytes) para ciertos casos. Como la penalización por fallo es muy alta (de millones de ciclos e incluso visible al usuario), es muy importante reducir la tasa de fallos, por lo que se usa un esquema totalmente asociativo, y como los fallos de página los gestiona el sistema operativo (porque la sobrecarga de usar software aquí es despreciable) se pueden usar algoritmos de reemplazo más complejos. Siempre se usa postescritura.

5.1. La tabla de páginas

Una página virtual puede estar asociada a cualquier página física o a ninguna, y esta información se almacena en la **tabla de páginas**, una tabla multinivel manejada por el sistema

operativo, almacenada en memoria principal e indexada por el número de página virtual, que en cada entrada contiene el número de página física junto con información adicional como el bit de validez, el bit de sucio, bits de protección (permisos de lectura, escritura y ejecución) y un bit de uso para políticas de reemplazo o direcciones de disco, y su tamaño se redondea por exceso al tamaño de palabra para facilitar el indexado.

La MMU incluye una caché especial, el **buffer de traducción adelantada de direcciones** o **TLB** (*Translation Lookaside Buffer*) que guarda traducciones de número de página virtual a número de página física. Suele emplear postescritura (principalmente para actualizar el bit de uso), pues se espera una tasa de fallos pequeña.

5.2. Tratamiento de los fallos de página

Al producirse un fallo de página se produce una **excepción por fallo de página**, transfiriendo el control de la CPU al sistema operativo que, una vez guardado el estado del programa en ejecución, debe encontrar la página solicitada en memoria secundaria y colocarla en la principal, o notificar de un **fallo de segmentación** (*segmentation fault*) si la página solicitada no se corresponde con ningún dato del proceso. El sistema operativo mantiene información de la posición en memoria secundaria donde se guarda cada página virtual, bien en la propia tabla de páginas o en una estructura aparte.

Normalmente se usa una política de reemplazo similar a LRU (pseudo-LRU), pues se quieren minimizar los fallos de página pero un LRU puro es demasiado caro por requerir actualizar la estructura en cada acceso a memoria. Por ello algunas MMU proporcionan un **bit de uso** o **de referencia** que se activa por hardware cuando se accede a la página, y periódicamente el sistema operativo desactiva todos los bits de uso, de forma similar a como se hacía en las cachés. Este esquema se puede perfeccionar más si en vez de tener en cuenta el bit de uso en el último periodo se tiene en cuenta el bit de uso en un número fijo de periodos anteriores.

5.3. Protección

El sistema operativo mantiene separadas las páginas virtuales de programas distintos para que uno no pueda acceder a los datos de otro, y cuando cambia de proceso (**cambio de contexto** o **de proceso**), cambia el valor del registro de tabla de páginas y, si hay TLB, lo vacía.

Para que esta protección sea efectiva, la CPU tiene (al menos) dos modos de funcionamiento: **modo usuario**, para programas de usuario, y **modo supervisor** o **núcleo**, para el sistema operativo. Desde el modo usuario, la lectura y escritura queda limitada a su espacio de direcciones virtuales, puede que con ciertas restricciones adicionales, impidiendo además que escriba en la tabla de páginas, la TLB o el registro que apunta a la tabla de páginas, o cambie el modo de la CPU.

La CPU cambia de modo usuario a supervisor si se produce una interrupción, una excepción producida por el programa o una llamada al sistema, basada en una instrucción especial (*syscall*) que transfiere el control al sistema operativo. El retorno a modo usuario se realiza mediante una **instrucción de retorno de excepción** (RFE).

Capítulo 6

Gestión de la E/S

Según el **modelo von Neumann**, un computador está formado por: procesador, memoria y entrada/salida (E/S). Podemos clasificar los dispositivos de E/S en dispositivos de **almacenamiento**, **interfaz con el usuario**, **visualización y multimedia**, **comunicaciones** y **adquisición de datos**. También podemos clasificarlos según su ancho de banda.

6.1. Puertos y buses

Los periféricos tienen una serie de **puertos** de E/S, registros externos a la CPU a través de los cuales se comunican la CPU y los dispositivos, integrados en la **controladora** del dispositivo. Tipos de puertos:

- **De datos**: Lectura o escritura del dato a transferir.
- **De control**: Donde la CPU escribe las órdenes.
- **De estado**: Indica el estado en que se encuentra el dispositivo (por ejemplo, *ready*/*not ready*).

La comunicación se realiza por **buses** o **canales compartidos**, canales de comunicación en los que existen puntos de acceso en los que un dispositivo puede conectarse para formar parte del bus y comunicarse con el resto de dispositivos conectados. El hecho de ser compartido implica que las señales transmitidas por un dispositivo están disponibles para el resto.

Si se conectan muchos dispositivos a un bus, sus prestaciones disminuyen, al aumentar la latencia por el tiempo de coordinación, y se forma un **cuello de botella** o **congestión** por estar los dispositivos esperando a su turno para usar el bus. Este problema se soluciona usando varios buses de distintas velocidades organizados de forma jerárquica, con los dispositivos más exigentes conectados a los buses más rápidos y cercanos al procesador.

El **control de acceso al bus** o **arbitraje del bus** es un mecanismo para resolver conflictos en el uso del bus, como que varios dispositivos intentasen realizar una operación a través del bus a la vez. Este mecanismo decide qué dispositivo puede tomar control en cada instante.

La forma más simple es establecer un **amo** del bus (*master*), normalmente la CPU, que es el único elemento que puede ordenar transferencias, y el resto de dispositivos deben enviar una señal al amo para realizarlas; sin embargo esto no suele ser posible.

Un bus está formado por:

- **Líneas de control**: para gestionar el acceso y uso de las líneas de información.
- **Líneas de información**.
  - **Líneas de datos**: Para transmitir datos entre dispositivos. El conjunto de estas es el **bus de datos**, y su cardinal es la **anchura del bus de datos**, factor clave para determinar las prestaciones del sistema.
  - **Líneas de dirección**: Para determinar la fuente o destino del dato. El conjunto de estas líneas es el **bus de direcciones**, y su cardinal es la **anchura del bus de direcciones**, que determina el número de direcciones disponibles y por tanto el máximo de memoria y puertos direccionables.

Las líneas de información pueden ser **multiplexadas**, si se usa el mismo conjunto de líneas para direcciones y datos en instantes distintos definidos por un protocolo, o **dedicadas**, en las que cada grupo de líneas tiene una función específica.

La **anchura del bus** es el total de líneas de información, y distinguimos entre **buses en serie**, con una sola línea de información, y **buses en paralelo**, con varias líneas transmitiendo bits simultáneamente.

La **longitud del bus** es su longitud física, desde menos de un metro hasta cientos de metros. Los eventos en el bus se coordinan con una **temporización**, que puede ser:

- **Síncrona**: Una de las líneas de control es de reloj, y en esta se transmite una secuencia alterna de unos y ceros a intervalos iguales que marcan cuándo suceden los eventos. El tiempo entre dos flancos del mismo tipo es el **tiempo de ciclo de reloj** o **ciclo de bus**. Estos protocolos permiten un elevado ancho de banda, pero a cambio todos los dispositivos deben funcionar a la misma frecuencia y puede aparecer un problema por **sesgo de reloj**, la diferencia de tiempo entre que dos elementos ven un flanco de reloj, por lo que la señal de reloj debe ser encauzada cuidadosamente para minimizar el sesgo. Por ello el bus tiene un tamaño limitado.
- **Asíncrona**: No hay señal de reloj, por lo que el bus puede ser todo lo largo que queramos, y podemos conectar dispositivos con distintas frecuencias de funcionameintos. Sin embargo, son más lentos que los síncronos, necesitan más líneas de control y puede haber fallos de sincronización. Se usa un **protocolo de presentación** (*handshaking*) con una serie de pasos de modo que emisor y receptor solo proceden al siguiente paso si están de acuerdo.

La **frecuencia de funcionamiento** de un bus síncrono es la de la señal de reloj que rige las transferencias. El **ancho de banda** (*bandwidth*) **teórico** es la cantidad de información que puede ser transmitida por un bus, en cantidad de información por unidad de tiempo. Su valor en bytes por segundo es $\frac{fn}{8}$, donde n es la anchura del bus de datos y f es la frecuencia de funcionamiento en hercios. Por su parte, el **ancho de banda efectivo** se refiere a la cantidad de información por unidad de tiempo que realmente se transmite, pues puede ser necesario dedicar varios ciclos para el protocolo de acceso y el arbitraje del bus.

6.2. Direccionamiento

El acceso a los dispositivos se puede hacer de dos formas, que afectan al bus de direcciones y a la forma de programarlos:

- **E/S mapeada en memoria**: Parte del espacio de direcciones de memoria se asocia a los dispositivos de E/S.
- **E/S aislada**: Cada puerto de un dispositivo tiene un **número de puerto**, al que se accede con instrucciones de la ISA específicas, y estos números forman un espacio de direcciones de E/S dedicado.

La E/S aislada no consume parte del espacio de direcciones de memoria, lo que era útil en los procesadores de 8 y 16 bits como Intel 8086 que tenían un espacio de direcciones limitado. Sin embargo, supone mayor complejidad de la CPU por tener que implementar instrucciones de E/S adicionales, y resulta en un menor repertorio de instrucciones para realizar estas operaciones.

6.3. Manejo de la E/S

La técnica más simple es el **sondeo** (*polling*), **encuesta** o **escrutinio**, en la el procesador «sondea» los puertos para, una vez detectado un cambio de estado, actuar en consecuencia. La encuesta puede ser:

- **Continua** (**espera activa**): El procesador se dedica exclusivamente a esto para detectar un cambio de estado. Sólo es permisible en dispositivos dedicados (sistemas empotrados).
- **Periódica**: Se sondea cada cierta cantidad de tiempo. Lo habitual es usarla sólo para algunos dispositivos como el ratón.

El sondeo es la técnica con menor latencia, pero también supone una gran pérdida de tiempo de CPU, por lo que en la práctica los dispositivos se manejan por **interrupciones**: La CPU encarga al dispositivo una transferencia y continúa haciendo otras cosas, y cuando la tarea termina, el dispositivo avisa a la CPU mediante una interrupción externa. Entonces la CPU:

- Deja automáticamente lo que esté haciendo.
- Identifica qué dispositivo ha enviado la interrupción. Para ello, bien existe una línea de interrupción dedicada para cada dispositivo, como ocurre en MIPS, o se activa una línea de interrupción única y el dispositivo se identifica insertando un **número de interrupción** en el bus de datos de la CPU, como ocurre en IA32.
- Salta a la **rutina de servicio de la interrupción** (**RSI**) o **manejador**. Este puede estar en una dirección de memoria fija, como ocurre en MIPS, y contener código para comprobar qué interrupción concreta se ha producido, o puede saltar a una dirección variable indicada en una tabla de direcciones indexada por el número de interrupción, como ocurre en IA32, lo que se conoce como **interrupciones vectorizadas**.
  Antes de saltar a esta rutina es necesario guardar, como mínimo, el contador de programa y posiblemente el registro de estado para las condiciones (*flags*). El resto de registros los puede guardar la propia rutina.

- Una vez ejecutada la RSI, recupera el estado y reanuda el proceso interrumpido.

Este método, si bien puede mejorar el rendimiento respecto al sondeo, también puede incluso empeorarlo, por lo que normalmente se usa junto con DMA.

El **acceso directo a memoria** o **DMA** (*Direct Memory Access*) es un mecanismo que permite la transferencia de datos desde un dispositivo a memoria, o viceversa, sin intervención del procesador. Para ello se una una **controladora de DMA** (normalmente varias), circuito especializado en transferir datos entre dispositivos y memoria. Esto conlleva que el bus tenga varios amos (CPU y DMA), por lo que es necesario un sistema de arbitraje. Para realizar una transferencia DMA:

- La CPU inicializa la controladora de DMA con datos como origen y destino de datos, número de bytes a transferir y sentido del desplazamiento (direcciones crecientes, decrecientes o fijas para origen y destino).

- La controladora de DMA pide el bus, y cuando lo consigue va realizando las operaciones solicitadas.

- Finalmente, la controladora de DMA genera una interrupción indicando fin de transferencia o error.

6.4. El sistema operativo

Cuando arranca el ordenador, se realizan algunas comprobaciones y operaciones iniciales y a continuación se carga el sistema operativo, que a su vez carga los *drivers* de los dispositivos, es decir, las rutinas de petición de E/S y las posibles RSI asociadas. Sólo el sistema operativo tiene conocimiento de los puertos, órdenes, etc., y por seguridad es el único que puede acceder a E/S, mientras que el resto de programas deben solicitar sus servicios mediante **llamadas al sistema** (*syscalls*).

Apéndice A

Ensamblador de MIPS

A.1. Registros

Número	**Nombre ABI**	**Uso**
`$0`	`$zero`	Conectado al valor 0.
`$1`	`$at`	Para uso temporal por el ensamblador.
`$2`, `$3`	`$v0`, `$v1`	Resultados de llamadas a procedimiento.
`$4`–`$7`	`$a0`–`$a3`	Parámetros de las llamadas a procedimiento.
`$8`–`$15`, `$24`, `$25`	`$t0`–`$t9`	Valores temporales.
`$16`–`$23`	`$s0`–`$s7`	Variables locales, preservadas entre llamadas.
`$26`, `$27`	`$k0`, `$k1`	Reservados para su uso por el sistema operativo.
`$28`	`$gp`	Puntero al segmento de datos (no us. en práct.).
`$29`	`$sp`	Puntero de pila.
`$30`	`$fp`	Puntero de marco (no usado en prácticas).
`$31`	`$ra`	Dirección de retorno, preservada entre llamadas.

Además:

- El coprocesador 0, de manejo de excepciones, posee los registros `$8` (`vaddr`), `$12` (`status`), `$13` (`cause`) y `$14` (`epc`), de 32 bits cada uno.
- El coprocesador 1, de punto flotante, posee los registros `$f0`–`$f31`, de 32 bits cada uno, que representan un entero de simple precisión o se combinan en parejas (nombradas como el registro de menor número, que será par) para representar enteros de doble precisión.
- El registro `pc` contiene la *siguiente* instrucción a ejecutar.
- Los registros `hi` y `lo` se combinan para representar un entero de 64 bits, usado en multiplicaciones.

Al inicio de un procedimiento, se disminuye `$sp` tanto como sea necesario y se guardan, en las direcciones `0($sp)`, `4($sp)`, etc., si es necesario, el valor del registro `$ra` y el de los registros `$s`*x* que vayamos a utilizar. Al final, se vuelven a cargar estos valores en los registros correspondientes, se aumenta `$sp` y se salta a la dirección apuntada por `$ra`.

A.2. Instrucciones

- **Aritmético-lógicas:** `add[i][u]` ($R_d \leftarrow R_s + Op2$), `and[i]` ($R_d \leftarrow R_s \wedge Op2$), `div[u]` ($LO \leftarrow \left\lfloor \frac{R_s}{R_t} \right\rfloor$, $HI \leftarrow R_s - R_t LO$), `madd[u]` ($HI:LO \leftarrow HI:LO + R_s \cdot R_t$), `msub[u]` ($HI:LO \leftarrow HI:LO - R_s \cdot R_t$), `mul` ($HI:LO \leftarrow R_s \cdot R_t$, seguido de $R_d \leftarrow LO$, sin desbordamiento), `mult[u]` ($HI:LO \leftarrow R_s \cdot R_t$), `nor` ($R_d \leftarrow \neg(R_s \vee R_t)$), `or[i]` ($R_d \leftarrow R_s + R_t$), `slt[i][u]` ($R_d \leftarrow \begin{cases} 1 & \text{si } R_s < Op2 \\ 0 & \text{si } R_s \geq Op2 \end{cases}$), `sub[u]` ($R_d \leftarrow R_s - R_t$), `xor[i]` ($R_d \leftarrow R_s \text{ XOR } R_t$).
  Parámetros: R_d, R_s y $Op2$, donde $Op2$ es un inmediato si se añade `i` o un registro si no. La `u` indica que se ignora el posible desbordamiento, salvo en instrucciones de multiplicación, en cuyo caso indica que la multiplicación es sin signo. Las operaciones lógicas son bit a bit.

- **De desplazamiento de bits:** `sll[v]` ($R_d \leftarrow R_s\,\acute{}\,Op2$), `sra[v]` ($R_d \leftarrow R_s\,\check{}\,Op2$), `srl[v]` ($R_d \leftarrow (unsigned)R_s\,\check{}\,Op2$). Parámetros: R_d, R_s y $Op2$, donde $Op2$ es un registro si se añade `v` o un inmediato si no.

- **De punto flotante:** `abs.[d|s]` ($R_d \leftarrow |R_s|$), `add.[d|s]` ($R_d \leftarrow R_s + R_t$), `c.[eq|le|lt].[d|s]` ($coproc1.cond[l] \leftarrow R_s[= \mid \leq \mid <]R_t$), `ceil.w.[d|s]` ($((int)R_d) \leftarrow \lceil R_s \rceil$), `cvt.[d.[s|w]|s.[d|w]|w.[d.s]]` ($((double|float|int)R_d) \leftarrow ((double|float|int)R_s)$, `div.[d|s]` ($R_d \leftarrow \frac{R_s}{R_t}$), `floor.w.[d|s]` ($((int)R_d) = \lfloor R_s \rfloor$), `mul.[d|s]` ($R_d \leftarrow R_s \cdot R_t$), `round.[d|s]` ($((int)R_d) \leftarrow round(R_s)$), `sqrt.[d|s]` ($R_d \leftarrow \sqrt{R_s}$), `sub.[d|s]` ($R_d \leftarrow R_s - R_t$), `trunc.w.[d|s]` ($((int)R_d) \leftarrow \lfloor R_s \rfloor$).
  Parámetros: l opcional para `c.[eq|le|lt].[d|s]` (por defecto 0), seguido de R_d, R_s y R_t (sólo los que se indican en la descripción de instrucción). Estos corresponden a los registros del coprocesador 1, tratados como de simple precisión, si la instrucción termina en `.s`, o parejas de estos, tratados como de doble precisión, si termina en `.d`; sin embargo, el indicar $float$, $double$ o int junto a estos indica que se trata de números en punto flotante de simple precisión, de doble precisión (mediante parejas de registros) o enteros de 32 bits (en los tres casos registros del coprocesador 1).

- **De salto:** `bc1f` ($coproc1.cond[l] = 0$), `bc1t` ($coproc1.cond[l] = 1$), `beq` ($R_s = R_t$), `bgez[al]` ($R_s \geq 0$), `bgtz` ($R_s > 0$), `blez` ($R_s \leq 0$), `bltz[al]` ($R_s < 0$), `bne` ($R_s \neq R_t$), `j[al]` (siempre). Entre paréntesis se indica la condición necesaria para $PC \leftarrow label$. `al` además hace antes \$`ra` $\leftarrow PC$. Parámetros: l opcional para `bc1f` o `bc1t` (por defecto 0) o uno o dos registros (`Rs[, Rt]`) según requiera la condición, seguidos de $label$.
  También, `jr Rs` para $PC \leftarrow R_s$, o `jalr [Rd, ]Rs` (por defecto R_d = \$`ra`) para antes hacer $R_d \leftarrow PC$.

- **De excepción:** `teq[i]` ($R_s = Op2$), `tge[i][u]` ($R_s \geq Op2$), `tlt[i][u]` ($R_s < Op2$), `tne[i]` ($R_s \neq Op2$). Entre paréntesis se indica la condición para provocar una excepción de software. Parámetros: R_s y $Op2$, donde $Op2$ es un inmediato si se añade `i` o un registro si no. La comparación es con signo salvo si se especifica `u`.

- **De acceso a memoria:** `l[b][u]` ($R_d \leftarrow (s16|u16)Mem)$), `l[d|w]c1` ($((double|float)R_d) \leftarrow Mem$), `lh[u]` ($R_d \leftarrow (s16|u16)Mem$), `lw` ($R_d \leftarrow (int)Mem$),

sb ($(s8|u8)Mem \leftarrow R_t$), s[d|w]c1 ($(double|float)Mem \leftarrow ((double|float)R_t)$), sh ($(s16|u16)Mem \leftarrow R_t$), sw ($(int)Mem \leftarrow R_t$). Parámetros: R_t, [$despl.$][(R_s)] (por defecto, $despl. = 0$ y $R_s = \$0$), con $Mem = *(despl. + R_s)$.
ll es como lw pero, en sistemas multinúcleo funciona con sc, similar a sw, para lectura-modificación-escritura atómica. lwl y lwr cargan de 1 a 4 bytes, justificados respectivamente a izquierda o derecha, desde la dirección dada hasta el byte, respectivamente, menos o más significativo de la palabra. swl y swr hacen lo mismo pero al revés (almacenan).

- **De transferencia:** lui ($R_d \leftarrow 2^{16} \cdot imm$), mfc0 ($R_d \leftarrow E_s$), mfc1 ($R_d \leftarrow F_s$), mfhi ($R_d \leftarrow HI$), mflo ($R_d \leftarrow LO$), mov.d ($D_d \leftarrow D_s$), mov.s ($F_d \leftarrow F_s$), movf[|.d|.s] ($coproc1.cond[l] = 0 : G_d \leftarrow G_s$), movt[|.d|.s] ($coproc1.cond[l] = 1 : G_d \leftarrow G_s$), movn[|.d|.s] ($R_t \neq 0 : G_d \leftarrow G_s$), movz[|.d|.s] ($R_t = 0 : G_d \leftarrow G_s$), mtc0 ($E_t \leftarrow R_s$), mtc1 ($F_t \leftarrow R_s$), mthi ($HI \leftarrow R_s$), mtlo ($LO \leftarrow R_s$).
Parámetros: En orden, X_d, X_s, X_t e imm (de estos sólo los que aparezcan en la descripción), más l opcional (por defecto 0) para movf[|.d|.s] y movt[|.d|.s]. En la descripción, la X aparece como una R para registros del procesador, E si son del coprocesador 0, F si son del 1 y D si son parejas de registros del 1 para doble precisión, y G indica F cuando la instrucción acaba en .s, D cuando acaba en .d o R en caso contrario.

- **Miscelánea:** break [imm] (termina la ejecución con una excepción, con código especificado opcionalmente por imm), clo Rd, Rs (establece R_d como el número de 1s desde el bit más significativo de R_s), clz Rd, Rs (igual pero con 0s), eret (vuelve de una excepción, $PC \leftarrow$ epc, status$[1] \leftarrow 0$), nop (no hace nada), syscall (realiza una llamada al sistema).

A.3. Pseudoinstrucciones

- **Aritmético-lógicas**: Poder omitir R_d si $R_d = R_s$. Poder especificar un inmediato mayor que $2^{16} - 1$ (o fuera del rango $[-2^{15}, 2^{15} - 1]$ si es con signo). abs Rd, Rs ($R_d \leftarrow |R_s|$). div[u] ($R_d \leftarrow \left\lfloor \frac{R_s}{R_t} \right\rfloor$). mulo[u] (como mul[u] pero con detección de desbordamiento). neg[u] ($R_d \leftarrow -R_s$, la u indica que no se detectan desbordamientos). not ($R_d \leftarrow \neg R_s$). rem[u] ($R_d \leftarrow R_s - R_t \left\lfloor \frac{R_s}{R_t} \right\rfloor$). seq, sge[u], sgt[u], sle[u], sne ($R_d = \begin{cases} 1 & \text{si } R_s[= | \geq | > | \leq | \neq]Op2 \\ 0 & \text{si } R_s[\neq | < | \leq | > | =]Op2 \end{cases}$).

- **De desplazamiento de bits**: ro[l|r] Rd, Rs, Op2 (rota los bits de R_s, respectivamente a izquierda o derecha, en un número de bits indicado por $Op2$, que puede ser registro o inmediato).

- **De salto**: Poder comparar con inmediatos y de tamaño arbitrario. b label (incondicional). bge[u] ($R_s \geq Op2$), bgt[u] ($R_s > Op2$), ble[u] ($R_s \leq Op2$) y blt[u] ($R_s < Op2$), donde la u indica que la comparación es sin signo. beqz ($R_s = 0$) y bnez ($R_s \neq 0$).

- **De acceso a memoria**: Poder especificar un desplazamiento fuera del rango $[-2^{15}, 2^{15} - 1]$, o una etiqueta, o la suma de $etiqueta$+$despl.$.

`l.[d|s]` $(((double|float)R_d) \leftarrow (double|float)Mem)$. `ld` $(R_{d+1} : R_d \leftarrow (long)Mem)$. Análogamente se definen `s.[d|s]` y `sd`. También `ulh[u]`, `ulw`, `ush`, `usw` (similares, respectivamente, a `lh[u]`, `lw`, `sh` y `sw` pero no requieren que la dirección dada sea múltiplo del tamaño del dato a mover).

- **De transferencia**: `la` (similar a `lb` pero carga la *dirección* dada en vez de su contenido). `li` ($R_d \leftarrow imm$, con imm de tamaño arbitrario). `mfc1.d` ($R_{d+1} : R_d \leftarrow D_s$). `mtc1.d` ($D_s \leftarrow R_{t+1} : R_t$).