# 2. Arquitectura MIPS.

## 2.1. Introducció.

Per a dominar la circuitería d'un computador, es deu parlar el seu llenguatge. Les paraules del llenguatge de la màquina es denominen **instruccions**, i el seu vocabulari es denomina **repertori de instruccions**.

Es podria pensar que el llenguatge de les màquines deu ser tan divers com el dels humans, en realitat són prou similars, però, i es semblen més bé a dialectes regionals què a llengües independents. En el moment en què s'aprèn un, és fàcil aprendre'n els demés. Aquesta similitud es deguda a que tots els computadors es construeixen amb tecnologies de circuitería basades en principis fonamentals semblants, i perquè hi ha un repertori bàsic d'operacions que totes les màquines deuen proporcionar. D'altra banda, els dissenyadors de computadors comparteixen una fi comú: trobar un llenguatge capaç de fer fàcil la construcció de la circuitería i del compilador i que al mateix temps maximice el rendiment i minimitze el cost. Un exemple d'aquest intent de unificació el trobem quan parlem del repertori de instruccions de la màquina MIPS s'empra en grans marques comercials com ho són: NEC, Nintendo, Silicon Graphics i Sony, entre d'altres.

## 2.2. Operacions de la circuitería del computador.

Tots els computadors deuen ser capaços de realitzar càlculs aritmètics. La següent notació del llenguatge ensamblador MIPS,

**add a, b, c**

indica al computador que ha de sumar dues variables **b** i **c**, i guardar el resultat en **a**.

Aquesta notació és rígida, de manera què cadascuna de les instruccions aritmètiques del MIPS realitza tan sols una operació, i sempre deu tindre exactament tres variables. Per exemple, pensem en el cas que volguérem guardar la suma de les variables **b**, **c**, **d** i **e** en la variable **a**. La següent seqüència de instruccions suma les variables:

```
add a, b, c    #La suma de b i c es posa en a.
add a, a, d    #La suma de b, c, i d està ara en a.
Add a, a, e    #La suma de b, c, d i e estarà ara en a.
```

Així que es necessiten tres instruccions per a realitzar la suma de quatre variables.

Les paraules que apareixen a la dreta del símbol # en cadascuna de les línies anteriors són comentaris per al lector i són ignorats pel computador. A diferència d'altres llenguatges de programació, cada línia d'aquest llenguatge tan sols pot contindre una sola instrucció. Una altra diferència és que els comentaris sempre acaben al final d'una línia.

El nombre natural d'operands per a una operació com és la suma, és tres: els dos nombres que es sumen junts i el lloc on es guarda el resultat de la suma. El fet què cada instrucció requira tindre exactament tres operands, ni més ni menys, està d'acord amb la filosofia de mantindre la circuitería simple: ja què la circuitería per a un nombre variable d'operands és més complicada que per a un nombre fixe. Aquesta situació il·lustra el primer dels quatre principis fonamentals del disseny de circuitería:

*Principi de disseny 1: La simplicitat afavoreix la uniformitat.*

## 2.3. Operands de la circuitería del computador.

A diferència dels programes escrits en llenguatge d'alt nivell, els operands de les instruccions aritmètiques no poden ser variables; s'ha d'utilitzar un nombre limitat de posicions especials anomenades **registres**. Els registres són els atobons de la construcció del computador, ja què són els conceptes bàsics emprats en el disseny de la circuitería, què són també visibles per al programador quan el computador està acabat. El tamany d'un registre en l'arquitectura MIPS és de 32 bits; aquests grups de 32 bits són tan comuns que se'ls ha donat el nom de **paraula** en l'arquitectura MIPS.

Una diferència ben significativa entre les variables d'un llenguatge de programació i els registres és el limitat nombre d'aquests últims, típicament 32 als computadors actuals. MIPS té 32 registres. Continuant amb l'evolució de la representació simbòlica del llenguatge MIPS, s'ha d'afegir a l'esmentat en l'apartat anterior que els tres operands de les instruccions aritmètiques MIPS deuen ser cadascun d'ells triat d'entre els 32 registres de 32 bits.

La raó del límit de 32 registres es troba al segon dels principis fonamentals de disseny de la tecnologia de la circuitería:

*Principi de disseny 2: A més xicotet, més ràpid.*

Un nombre alt de registres incrementaria el temps del cicle de rellotge, simplement perquè es necessiten senyals electròniques més llargues per tant d'arribar més lluny.

Directrius del tipus "a més xicotet, més ràpid" no són absolutes; 31 registres poden no ser més ràpids que 32. La veritat que hi ha darrere d'aquest tipus d'observacions obliga als dissenyadors de computadors a prendre-les de forma seriosa. En aquest cas, el dissenyador deu equilibrar l'ànsia de més registres per part dels programes, amb el desig del dissenyador per mantindre el cicle de rellotge ràpid.

Encara què es poden escriure instruccions simplement emprant nombres per als registres, de 0 a 31, la convenció MIPS gasta dos caràcters precedits per un símbol de dòlar per representar els registres.

Els llenguatges de programació tenen variables simples que contenen dades simples, però també gasten estructures de dades més complexes, com les taules. Aquestes estructures de dades complexes poden contindre molts més elements que registres hi ha en la màquina. Com pot un computador representar i accedir a d'aquest tipus d'estructures grans?

Recordem els components bàsics d'un computador, el processador pot mantindre tan sols una xicoteta quantitat de dades als registres, però la memòria del computador conté milions d'elements. D'ací que les estructures de dades com les taules es guarden en memòria.

Com s'ha explicat abans, les operacions aritmètiques es produeixen entre registres en instruccions MIPS; d'aquesta manera el MIPS deu incloure instruccions que puguen transferir dades entre memòria i registres. Aquest tipus de instruccions s'anomenen instruccions de **transferència de dades**. Per tal d'accedir a una **paraula** en memòria, la instrucció deu proporcionar la direcció de memòria. La memòria és simplement una gran taula unidimensional i la direcció realitza el paper d'índex d'eixa taula, començant pel 0. Per exemple a la figura 2.1, la direcció del tercer element es 2, i per tant el valor de **Memòria[2]** es 10.

La instrucció de transferència que mou dades des de memòria fins a algun registre es denomina, tradicionalment, **load** (carregar). El format de la instrucció load, és el nom de l'operació, seguit pel registre a carregar, una constant i el registre utilitzat per a accedir a memòria. La direcció de memòria es forma mitjançant la suma de la part de la constant de la instrucció i el contingut del segon registre. El nom MIPS real per a aquesta instrucció és **lw**, contracció de **load word** (carregar paraula).

La instrucció complementària a load tradicionalment s'anomena **store** (emmagatzemar). Transfereix dades des d'un registre fins a la memòria. El

format de store és semblant al de load: el nom d'operació, seguit del registre a emmagatzemar, després el desplaçament per a seleccionar l'element de la taula i, finalment, el registre base. De nou, la direcció MIPS s'especifica d'una banda per una constant i per l'altra pel contingut d'un registre. El nom MIPS real es **sw**, contracció de store word (emmagatzemar paraula).

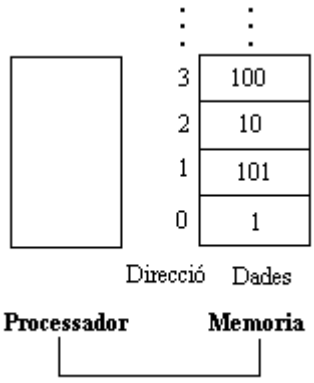


Figura 2.1.

Cal mencionar què a les taules s'accedeix freqüentment amb variables en compte de constants, per tant l'element de la taula seleccionat pot canviar mentre el programa està funcionant.

La figura 2.2 resumeix les parts de la representació simbòlica del repertori de instruccions del MIPS descrites fins al moment. Les instruccions load word i store word són les instruccions que transfereixen paraules entre la memòria i els registres en l'arquitectura MIPS. Altres marques de computadores utilitzen altres instruccions per al cas de la transferència de dades, a més de les ja esmentades load i store. Una arquitectura amb aquest tipus d'alternatives és l'Intel 80x86.

## 2.4. Representació de instruccions al computador

Ara ja es pot explicar la diferència entre la forma en què els humans donen instruccions a les màquines i la forma en la què les màquines veuen aquestes instruccions. Però primer, es farà un ràpid repàs a la forma en la què les màquines representen els nombres.

Als humans se'ls ensenya a pensar en base 10, però els nombres es poden representar en qualsevol base. Per exemple, 123 base 10 es el mateix què 1111011 base 2.

Els nombres es guarden a la circuitería del computador com a series de senyals electròniques altes i baixes, i per tant es consideren nombres en base 2.

Un simple dígit d'un nombre binari és per tant "l'àtom" de la computació, ja què tota la informació està formada per dígits binaris o bits. Aquest bloc de construcció fonamental pot tindre dos valors, que es poden veure de diferents formes: alt o baix, encès o apagat, vertader o fals, 1 ó 0, etc.

**Operands del MIPS**

| Nom | Exemple | Comentaris |
|---|---|---|
| 32 registres | $s0, $s1, ...<br>$t0, $t1, ... | Ubicació ràpida per a dades. En el MIPS, les dades deuen estar als registres per poder realitzar l'operació aritmètica. |
| $2^{30}$ paraules de memòria | Memòria[0]<br>Memòria[4]<br>Memòria[4294967292] | S'accedeix mitjançant instruccions de transferència de dades. Utilitzen direccions de byte; les direccions difereixen en 4 entre paraules successives. |

**Llenguatge ensamblador del MIPS**

| Categoria | Instrucció | Exemple | Significat | Comentaris |
|---|---|---|---|---|
| Aritmètica | add | add $s1,$s2,$s3 | $s1=$s2+$s3 | Tres operands, dades en registres |
| | substract | sub $s1,$s2,$s3 | $s1=$s2-$s3 | |
| Transferència de dades | load word | lw $s1,100($s2) | $s1=Mem[$s2+100] | Mem a Registres |
| | store word | sw $s1,100($s2) | Mem[$s2+100]=$s1 | Registres a Mem |

Figura 2.2.(Ambdues taules).

A més, les instruccions s'emmagatzemen al computador com a sèries de senyals electròniques altes i baixes i poden representar-se com a nombres. De fet, cada part d'una instrucció es pot considerar com un nombre individual, i ajuntant tots aquests nombres es forma la instrucció.

Degut a què els registres formen part de quasi totes les instruccions, deu haver-ne una convenció per fer correspondre els noms dels registres amb nombres. En el llenguatge ensamblador del MIPS, els registres *$s0* a *$s7* es corresponen amb els registres 16 a 23, i els registres *$t0* a *$t7* es corresponen amb els registres 8 a 15. Així, $s0 es correspon amb el registre 16, $s1 amb el registre 17, $s2 amb el registre 18, ..., $t0 es correspon amb el registre 8, $t1 amb el registre 9, i així successivament. La convenció dels demés registres es descriurà més endavant.

### 2.4.1. Camps MIPS

Als camps MIPS se'ls dóna una sèrie de noms per identificar-los més fàcilment.

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6bits | 5 bits | 5 bits | 5 bits | 5bits | 6 bits |

Ací s'indica el significat de cadascun dels noms dels camps de les instruccions MIPS:

- **op**: Operació bàsica de la instrucció, tradicionalment s'anomena **codi d'operació**.
- **rs**: Primer registre operand.
- **rt**: Segon registre operand.
- **rd**: Registre operand destí, on s'emmagatzema el resultat de l'operació.
- **shamt**: Tamany del desplaçament (**shift amount**).
- **funct**: Funció. Aquest camp selecciona la variant específica de l'operació del camp op, i de vegades se'l denomina **codi de funció**.

Quan una instrucció necessita camps més llargs que els mostrats abans, apareix un problema. Per exemple, la instrucció *load word* deu especificar dos registres i una constant. Si la direcció gastara un dels camps de 5 bits del format anterior, la constant dintre la instrucció de càrrega estaria limitada a tan sols $2^5$ ó 32. Aquesta constant s'utilitza per a seleccionar elements de taules grans o de estructures de dades, i freqüentment necessita ser molt més gran que 32. Aquest camp de 5 bits és massa xicotet per a ser útil.

Ací tenim un conflicte entre el desig de guardar totes les instruccions amb la mateixa longitud i el desig de tindre un únic format de instrucció. Açò condueix al tercer principi de disseny de circuitería:

*Principi de disseny 3: Un bon disseny necessita unes bones solucions de compromís.*

Un compromís triat pels dissenyadors del MIPS va ser guardar totes les instruccions amb la mateixa longitud, per això es requereixen diferents classes de formats de instrucció per a diferents classes de instruccions. Per exemple, el format anterior es denomina *Tipus-R* (de registre) o *Format-R*. Un segon tipus de format de instrucció es denomina *Tipus-I* o *Format-I* i es gastat per instruccions de transferència de dades. Els camps del Format-I, són:

| op | rs | rt | Adreça |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 6 bits |

Els 16 bits d'adreça significa què una instrucció de load pot carregar qualsevol paraula dins una regió $\pm 2^{15}$ ó 32768 bytes ($2^{13}$ ó 8192 paraules) de la direcció del registre base rs.

Analitzem la instrucció load word, abans mencionada:

*lw $t0,32($s3)* #reg. temp. $s0 es carrega amb Mem[32+$s3]

Ací, al camp rs es col·loca 19 (per $s3), al camp rt es col·loca 8 (per $t0) i al camp d'adreça es col·loca 32. Cal mencionar que el significat del camp rt ha canviat per a aquesta instrucció: en una instrucció load word, el camp rt especifica el registre destí, el qual rep el resultat de la càrrega.

Encara que tindre múltiples formats complica la circuitería, es pot reduir la complexitat d'emmagatzemar-los de manera semblant. Per exemple, els tres primers camps dels formats Tipus-R i Tipus-I són del mateix tamany i tenen els mateixos noms; el quart camp del Tipus-I és igual a la longitud dels darrers 3 camps del Tipus-R.

Els formats es distingeixen pel valor del primer camp: a cada format se li assigna un conjunt de valors distints en el primer camp (op) i per tant la circuitería sap si ha de tractar la darrera meitat de la instrucció com a tres camps (Tipus-R) o en canvi com un sols camp (Tipus-I). La figura 2.3 mostra els nombres utilitzats en cadascun dels camps de les instruccions MIPS que hem vist fins ara:

| Instrucció | Format | op | rs | rt | rd | shamt | funct | Adreça |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | reg | reg | reg | 0 | 32 | n.a. |
| sub | R | 0 | reg | reg | reg | 0 | 34 | n.a. |
| lw | I | 35 | reg | reg | n.a. | n.a. | n.a. | Adreça |
| sw | I | 43 | reg | reg | n.a. | n.a. | n.a. | Adreça |

Figura 2.3.

Ara, a la figura 2.4, podem observar una ampliació de la taula de la figura 2.3 així com també de la taula de la figura 2.2, amb exemples numèrics del que s'ha vist fins ara.

| Nom | Format | Exemple | | | | | | Comentaris |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | 18 | 19 | 17 | 0 | 32 | add $s1,$s2,$s3 |
| sub | R | 0 | 18 | 19 | 17 | 0 | 34 | sub $s1,$s2,$s3 |
| lw | I | 35 | 19 | 18 | 100 | | | lw $s1,100($s2) |
| sw | I | 43 | 19 | 18 | 100 | | | sw $s1,100($s2) |
| Tam. camp | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | 32 bits |
| Format-R | R | op | rs | rt | rd | shamt | funct | Aritmètica |
| Format-I | I | op | rs | rt | Adreça | | | Transferència |

Figura 2.4.

Els computadors d'avui en dia es construeixen en base a dos principis claus:

1. Les instruccions es representen com a nombres.
2. Els programes es poden emmagatzemar en memòria per ser llegits o escrits precisament com a nombres.

Aquests dos principis ens porten al concepte de **programa emmagatzemat**; la seua invenció ha deixat l'ingeni a l'aire. La figura 2.5 mostra la potència d'aquest concepte; específicament, la memòria pot contindre el codi font d'un programa editor, el corresponent codi màquina compilat, el text del que el programa compilant està utilitzant i a més a més el compilador que ha generat el programa.

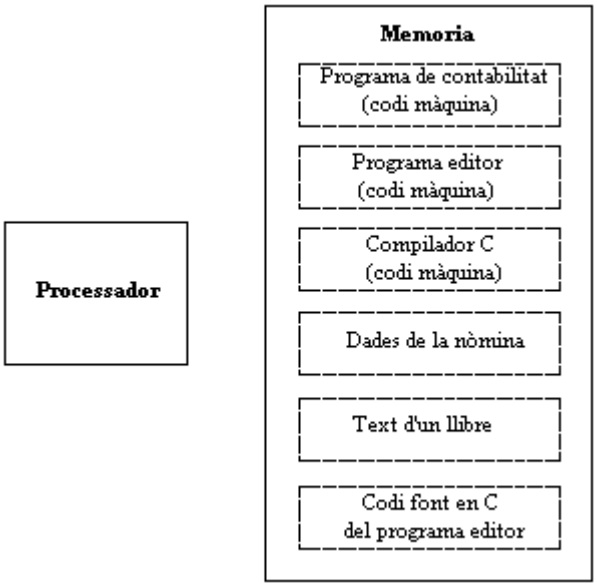


Figura 2.5.

## 2.5. Instruccions per a la presa de decisions.

El que distingeix a un computador d'una simple calculadora és l'habilitat de prendre decisions. Tenint en compte les dades d'entrada i els valors creats al llarg de la computació, el computador executa diferents instruccions. La presa de decisions es representa de manera comuna als llenguatges de programació utilitzant la sentència **if** (si condicional), combinada de vegades amb sentències **go to** (anar a) i etiquetes. El llenguatge ensamblador del MIPS inclou dos instruccions de presa de decisions, similars a una sentència if amb un go to. La primera instrucció és:

*beq registre1,registre2,L1*

Aquesta instrucció significa anar a la sentència etiquetada per L1 si el valor de registre1 és igual al valor de registre2. El mnemotècnic beq significa **branch if equal** (botar si és igual). La segona instrucció és:

*bne registre1,registre2,L1*

Açò significa anar a la sentència etiquetada per L1 si el valor de registre1 no és igual al valor de registre2. El mnemotècnic bne significa **branch if not equal** (botar si no és igual). Ambdues instruccions es coneixen tradicionalment per **bots condicionals**.

### 2.5.1. Bucles.

Les decisions són importants, tant per escollir entre dues alternatives, que apareixen a les sentències **si**, com per a iterar una conmputació, com passa als bucles. Les mateixes instruccions ensamblador són els blocs bàsics de construcció per a ambdós casos.

La prova d'igualtat i desigualtat és probablement la més habitual de totes, però hi ha vegades en què és útil saber si el valor d'una variable és menor que el d'altra. Per exemple, en un bucle **for** (per a) es pot voler comprovar si el índex d'una variable és menor que 0. Aquest tipus de comparacions es realitzen al llenguatge ensamblador del MIPS amb una instrucció que compara dos registres i deixa el valor 1 en un tercer registre si el primer és menor que el segon; d'altra manera s'actualitza amb el valor 0. La instrucció MIPS s'anomena **set on less than** (activar si és menor que) o **slt**. Per exemple,

*slt $t0,$s3,$s4*

significa què el registre $t0 es carregarà amb el valor 1 si el valor de $s3 és menor que el de $s4, d'altra manera $t0 es carregarà amb el valor 0.

Tenint en compte l'advertència de von Newmann respecte a la simplicitat de "l'equip", l'arquitectura MIPS no inclou botar si menor que, donat què resultaria massa complicada; o bé allargaria el temps del cicle de rellotge, o la instrucció necessitaria cicles extra. Dos instruccions ràpides són més útils.

### 2.5.2. La sentència alternativa Case/Switch.

Molts llenguatges de programació tenen una sentència alternativa case o switch, que permet al programador seleccionar una de les moltes alternatives depenent d'un únic valor. Una manera de realitzar l'switch és a través d'una seqüència de proves condicionals, convertint la sentència switch en una cadena

de sentències if-then-else (si-aleshores-sino). De vegades, però, les alternatives es poden codificar de manera eficient com una taula de direccions de seqüències de instruccions alternatives, anomenada **taula de direccions de bots (jumps adress table)**, i el programa necessita tan sols accedir a aquesta taula i després botar a la seqüència apropiada. Aleshores la taula de bots és, simplement, una taula de paraules que conté direccions que es corresponen amb etiquetes en el codi.

Per permetre aquest tipus de situacions, els computadors com el MIPS inclouen una instrucció **jump register (jr)**, que significa un bot incondicional fins a la direcció especificada pel registre. El programa carrega l'entrada apropiada de la taula de bots en un registre, i després bota a la direcció indicada utilitzant un registre de bot.

La figura 2.6 resumeix les parts de codi del llenguatge ensamblador MIPS esmentades en el present apartat, com ara són els bots incondicionals i els bots condicionals.

**Llenguatge ensamblador del MIPS**

| Categoria | Instrucció | Exemple | Significat | Comentaris |
|---|---|---|---|---|
| Bot condicional | branch on equal | beq $s1,$s2,L | si ($s1==$s2) anar a L | Comprovació d'igualtat |
| | branch on not equal | bne $s1,$s2,L | si ($s1!=$s2) anar a L | Comprovació de no igualtat |
| | set on less than | slt $s1,$s2,$s3 | si ($s2<$s3) $s1=1 sinó $s1=0 | Comparar menor que |
| Bot incondicional | jump | j 2500 | anar a 10000 | Bot a l'adreça |
| | jump register | jr $t1 | anar a $t1 | Per a switch |

**Llenguatge màquina del MIPS**

| Nom | Format | Exemple | | | | | | Comentaris |
|---|---|---|---|---|---|---|---|---|
| beq | I | 4 | 17 | 18 | 25 | | | beq $s1,$s2,100 |
| bne | I | 5 | 17 | 18 | 25 | | | bne $s1,$s2,100 |
| slt | R | 0 | 18 | 19 | 17 | 0 | 42 | slt $s1,$s2,$s3 |
| j | J | 2 | 2500 | | | | | j 10000 |
| jr | R | 0 | 9 | 0 | 0 | 0 | 8 | jr $t1 |
| Tam. camp | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | 32 bits |
| Format R | R | op | rs | rt | rd | shamt | funct | Aritmètiques |
| Format I | I | op | rs | rt | Adreça | | | Transferència, bots |

Figura 2.6

## 2.6. Suport per a procediments en la circuitería del computador.

Un procediment o subrutina és una ferramenta que els programadors utilitzen per a estructurar programes amb la fi què el codi siga reutilitzat. Els procediments permeten al programador concentrar-se en una sola part del treball cada vegada. Els paràmetres actuen a mode de barrera entre el procediment i el restant codi de programa i les dades, permetent passar valors al procediment i que aquest retorne resultats.

Es pot veure un procediment com un espia que funciona amb un pla secret, adquireix recursos, realitza tarees, borra les empremtes i després retorna al punt original amb el resultat desitjat. Cap altra cosa deuria veures pertorbada una vegada la missió s'ha completat.

De manera molt similar, en l'execució d'un procediment, el programa deu seguir els següents passos:

- Situar els paràmetres en un lloc des de el qual el procediment puga accedir.

- Transferir el control al procediment.

- Adquirir els recursos d'emmagatzematge necessaris per al procediment.

- Realitzar la feina desitjada.

- Situar el valor del resultat en un lloc on el programa que l'ha cridat puga accedir-hi.

- Retornar el control al punt d'origen.

Tal com s'ha mencionat anteriorment, els registres són el lloc més ràpid per a situar dades en el computador, per tant es procura utilitzar-los el major nombre de vegades possible. D'ací que els programes MIPS, assignen els següents registres dels 32 disponibles en cada cridada al procediment:

- $a0 - $a3: quatre registres d'arguments en els quals es passen paràmetres.
- $v0 - $v1: dos registres de valors en els quals es retornen valors.
- $ra: un registre de retorn de direcció per a tornar al punt d'eixida.

A més d'aquesta assignació de registres, el llenguatge ensamblador del MIPS inclou una instrucció sols per a procediments: bota a una direcció i al mateix temps salva la direcció de la següent direcció en el registre $ra. La

instrucció **jump-and-link (jal)** (instrucció de bot i enllaç) simplement s'escriu com:

*jal AdreçaDelProcediment*

La part d'enllaç (link) significa que una adreça o enllaç està format per aquells punts del lloc de la cridada que permeten al procediment tornar a la adreça apropiada. Aquest "enllaç" emmagatzemat en el registre $ra s'anomena **adreça de retorn**. L'adreça de retorn és necessària perquè el mateix procediment es pot cridar des de diferents lloc del programa.

En la idea de programa emmagatzemat, es troba implícita la necessitat de tindre un registre per a guardar l'adreça de la instrucció actual que està sent executada. Per raons històriques, aquest registre quasi sempre s'anomena **comptador de programa**, abreviat per PC (**Progran Counter**) en l'arquitectura MIPS, encara que un nom més apropiat deuria ser registre de adreça d'instrucció. La instrucció jal guarda PC+4 en el registre $ra per a seguir amb la següent instrucció del programa amb la tornada del procediment.

Per a realitzar el bot de retorn ja disposem de la instrucció adequada:

*jr $ra*

La instrucció jump register, que s'ha utilitzat anteriorment en la sentència switch, bota a l'adreça emmagatzemada en el registre $ra, què és justament el que es desitja. D'aquesta manera, el programa que crida al procediment, o invocador (**caller**), posa els valors dels paràmetres en $a0-$a3, i utilitza jal X per a botar fins al procediment X (de vegades conegut per invocat (**callee**)). El procediment invocat després realitza els càlculs, posa els resultats en $v0-$v1 i torna el control al invocador utilitzant jr $ra.

### 2.6.1. L'ús de més registres.

Suposem què un compilador necessita més registres per a un procediment que les quatre registres d'arguments i els dos registres de retorn de valors. Com se suposa que deuen esborrar-se les empremtes després que la missió ha finalitzat, qualsevol registre necessari pel invocador deu ser restaurat amb els valors que contenia abans que el procediment fóra invocat. Aquesta situació es un exemple prou clar de quan és necessari bolcar el contingut de registres en memòria.

L'estructura de dades ideal per a bolcar registres és una **pila**, una cua en la que l'últim a entrar és el primer a eixir (LIFO). Una pila necessita un punter a l'última adreça utilitzada per la pila, per a guardar on es deu situar el següent procediment els registres que necessitarà bolcar, o per a saber on trobar els valors

antics dels registres. El punter de pila s'ajusta al tamany d'una paraula, per cada registre que es salva o restaura. Les piles són tan habituals que disposen dels seus noms propis per a transferir dades cap a la pila i des de ella: posar dades a la pila es coneix per **push** (o apilar) i extraure dades de la pila es coneix per **pop** (desapilar).

Els programes dels MIPS reserven un altre registre sols per a pila: **stack pointer** ($sp) (punter de pila) utilitzat per a salvar els registres necessaris pel invocador. Per precedents històrics, la pila "creix" de direccions superiors a direccions inferiors. Aquesta convenció significa què es poden posar valors en la pila restant al punter de pila. Sumar al punter de pila la redueix, substraent d'aquesta manera valors de pila.

Per a evitar salvar i restaurar un registre que conté un valor que mai es gasta, cosa que podria succeir amb un registre temporal, els programes MIPS ofereixen dues classes de registres:

- $t0 - $t9: 10 registres temporals que no són preservats pel invocat (procediment anomenat) en una cridada de procediment.
- $s0 - $s7: 8 registres salvats que deuen ser preservats en una cridada de procediment (si els utilitza, el invocat els guarda i els restaura).

Esta simple convenció redueix el bolcat de registres. A l'exemple anterior, com l'invocador (el procediment que fa la crida) no espera que els registres $t0-$t1 siguen preservats al llarg d'una crida de procediment, es pot evitar dos emmagatzematges i dues càrregues al codi. Encara se salvarà i restaurarà $s0, ja què l'invocat deu supondre que l'invocador necessita el seu valor.

## 2.6.2. Procediments anidats.

Els procediments que no criden a altres s'anomenen procediments **fulla**. La vida seria senzilla si tots els procediments foren procediments fulla, però no és així. De la mateixa manera que un espia pot necessitar a altres espies per a complimentar part de la seua missió, al mateix temps podrien necessitar inclús ajuda de més espies, els procediments invoquen a altres procediments. Encara més, els procediments recursius invoquen "còpies" de si mateix. De la mateixa manera que s'ha d'anar amb compte quan s'utilitzen registres en els procediments, més acula es deu tindre quan s'invoquen procediments no-fulla.

Per exemple, suposem que el programa principal crida al procediment A amb un argument de valor 3, situat aquest valor (3) al registre $a0 i després utilitza jal A. Després, suposem que el procediment A crida al procediment B, mitjançant la instrucció jal B, amb un argument de valor 7, també situat en el registre $a0. Ja què A no ha acabat encara la seua tasca, hi ha un conflicte amb l'ús del registre $a0. De la mateixa manera, hi ha un conflicte amb l'adreça de

retorn en el registre $ra, ja què ara té l'adreça de retorn per a B. Encara que es prenguen precaucions per previndre el problema, aquest conflicte eliminarà la capacitat del procediment A per a tornar al seu invocador.

Una possible solució és posar tots els demés registres que deuen preservar en la pila, tal com s'ha fet per a guardar els registres. L'invocador apila qualsevol registre d'arguments ($a0-$a3) o registres temporals ($t0-$t9), que li siguen necessaris després de la crida. L'invocador apila els registres de l'adreça de retorn $ra i qualsevol registre salvat ($s0-$s7) utilitzat per l'invocat. El punter de pila $sp s'ajusta per a contar el nombre de registres situats en la pila. Abans del retorn, els registres són restaurats de memòria i el punter de pila reajustat.

En la figura 2.7 resumeix què és el que es preserva al llarg d'una crida a un procediment. Note's que es fa ús de diversos mètodes per a preservar la pila. La pila per dalt de $sp es preserva simplement assegurant-se que l'invocat no escriu per dalt de $sp. L'invocat preserva $sp posant-li exactament la mateixa quantitat d'elements que s'han llevat, i els altres registres es preserven guardant-los en la pila (si s'utilitzen) i restaurant-los d'allí. Eixes accions també garanteixen que en una càrrega de la pila, l'invocador obtindrà de volta el mateix valor que haja posat en ella mitjançant un emmagatzematge: perquè l'invocat promet preservar $sp i perquè també promet no modificar la part de la pila de l'invocador que és, l'àrea per damunt de $sp en el moment de la crida.

| Preservats | No preservats |
|---|---|
| Registres d'argument: $sa0-$a3 | Registres de retorn de valors: $v0-$v1 |
| Registres salvats: $s0-$s7 | Registres temporals: $t0-$t9 |
| Registre de punter de pila: $sp | |
| Registre de retorn d'adreça: $ra | |
| Pila per dalt del punter de pila | Pila per baix del punter de pila |

Figura 2.7.

### 2.6.3. Reserva d'espai per a noves dades.

L'última complicació és que la pila també s'utilitza per a emmagatzemar variables que són locals al procediment i que no caben als registres, com taules locals o estructures. El segment de la pila que conté els registres guardats del procediment i les variables locals s'anomena **bloc d'activació**. La figura 2.8 mostra l'estat de la pila abans, al moment i després de la crida al procediment.

Alguns programes MIPS utilitzen un punter de bloc d'activació o **frame pointer** ($fp) per a senyalitzar la primera paraula del bloc d'activació del procediment. El punter de pila podria canviar al llarg del procediment i, d'aquesta manera les referències a una variable local en memòria podrien tindre diferents desplaçaments depenent d'on estigueren en el procediment, dificultant

la seua comprensió. De manera alternativa, el punter de bloc d'activació, ofereix un registre base estable dins d'un procediment per a referències de memòria locals. Es nota que en la pila apareix un bloc d'activació, independentment de què existeixa un punter de bloc explícit o no. Fins a aquest punt s'ha evitat la necessitat de $fp, evitant canvis en $sp dins dels procediments.

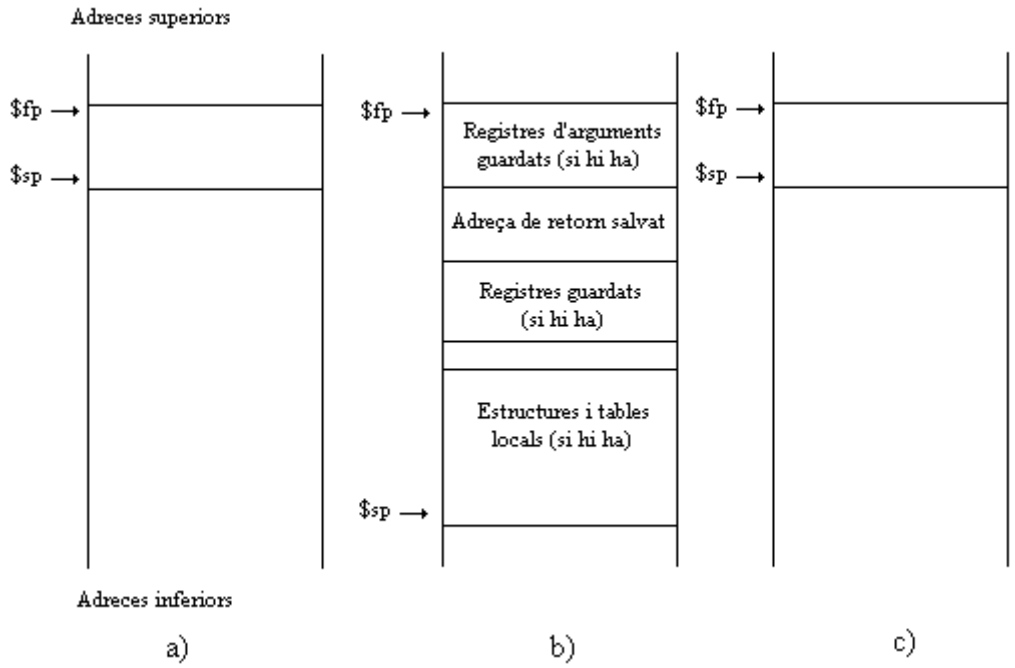


Figura 2.8. Il·lustració del espai reservat a la pila a)abans, b)al mateix temps i c) després de la crida al procediment.

La figura 2.9 resumeix les convencions dels registres per al llenguatge ensamblador del MIPS, i la figura 2.10 resumeix les parts del repertori d'instruccions MIPS descrites fins ara.

| Nom | Nombre de registre | Ús | Preservat en les cridades |
|---|---|---|---|
| $zero | 0 | Valor constant 0 | n.a. |
| $v0-$v1 | 2-3 | Valors per a resultats i avaluació d'expressions | no |
| $a0-$a3 | 4-7 | Arguments | si |
| $t0-$t7 | 8-15 | Temporals | no |
| $s0-$s7 | 16-23 | Salvats | si |
| $t8-$t9 | 24-25 | Més temporals | no |
| $gp | 28 | Punter global | si |
| $sp | 29 | Punter de pila | si |
| $fp | 30 | Punter de bloc d'activació | si |
| $ra | 31 | Adreça de retorn | si |

Figura 2.9.

**Llenguatge ensamblador del MIPS**

| Categoria | Instrucció | Exemple | Significat | Comentaris |
|---|---|---|---|---|
| Bot incondicional | jump and link | jal 2500 | $ra=PC + 4; anar a 10000 | Cridades a procediments |

**Llenguatge màquina MIPS**

| Nom | Format | | Exemple | Comentaris |
|---|---|---|---|---|
| jal | J | 3 | 2500 | jal 2500 |

Figura 2.10.

## 2.7. Més enllà dels nombres.

Els computadors s'inventaren per a devorar nombres, però tan prompte com arribà a ser viable la seua comercialització, s'utilitzaren per a processar texts. Hui en dia molts computadors utilitzen bytes de 8 bits per a representar caràcters i la representació que pràcticament tots utilitzen és l'American Standard Code for Information Interchange (ASCII). La figura 2.11 resumeix la representació ASCII.

Es pot utilitzar una sèrie d'instruccions per a extraure un byte d'una paraula, per tal motiu les instruccions **load word** i **store word** serveixen per a transferir bytes de la mateixa manera que paraules. De qualsevol manera, degut a la freqüència amb què apareix text en alguns programes, el MIPS proporciona instruccions especial per a moure bytes. La instrucció **load byte** (lb) carrega un byte de memòria i els posa en els huit bits més a la dreta d'un registre. La instrucció **store byte** (sb) llig un byte dels huit bits més a la dreta d'un registre i l'escriu a la memòria.

D'aquesta manera, se còpia un byte amb la seqüència:

```
lb $t0, 0($sp)   # Llegir byte de la font.
sb $t0, 0($gp)   #Escriure byte en el destí.
```

Els caràcters normalment se combinen en cadenes, que se composen d'un nombre variable de caràcters. Hi ha tres formes de representar una cadena:

(1) La primera posició de la cadena se reserva per a indicar la grandària de la mateixa ,
(2) Una variable complementària disposa de la grandària de la cadena (com en una estructura),
(3) L'última posició de la cadena s'indica amb un caràcter utilitzat per a indicar el final.

El llenguatge C utilitza la tercera forma, acabant les cadenes amb un byte, tenint com a valor 0 (denominat Null en ASCII). Doncs així, la cadena "Cal" se representa en el llenguatge C pels 4 bytes següents, mostrats com nombres decimals: 67, 97, 108, 0.

| V | C | V | C | V | C | V | C | V | C | V | C |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | espai | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ¡ | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | “ | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ‘ | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M. | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ¿ | 79 | O | 95 | _ | 111 | o | 127 | DEL |

Figura 2.11. Taula del codi ASCII.
On V = Valor ASCII i C = Caràcter.

## 2.8. Altres estils d'adreçament del MIPS.

Els dissenyadors de l'arquitectura MIPS proporcionen dues formes més d'accés als operands. La primera busca fer més ràpid l'accés a constants petites, i la segona busca fer més eficients els bots.

### 2.8.1. Constants o operands immediats.

Moltes vegades els programes utilitzen constants en les operacions per a, per exemple, incrementar un índex amb la finalitat d'apuntar al següent element d'una taula, contar iteracions d'un bucle, o ajustar la pila a una crida de procediments niats. De fet, en dos programes, més de la meitat de les instruccions aritmètiques té una constant com a operand: en el compilador de C gcc, el 52 % de les operacions aritmètiques utilitzen constants; en el programa de simulació de circuits Spice és el 69 %.

Utilitzant sols les instruccions que hem vist fins ara (add, sub, lw, sw, slt, j, jr, beq, bne, ..., per a utilitzar una constant hauria que carregar-la des de memòria. (Les constants tindrien que estar situades en memòria quan se carrega el programa). Per exemple, per a sumar la constant 4 al registre $sp, es podria utilitzar el codi:

```
lw $t0, AdreçaConstant($zero)  # $t0 = constant 4.
add $sp, $sp, $t0   # $sp = $sp + $t0 ($t0 == 4).
```

suposant que el AdreçaConstant és l'adreça de memòria de la constant 4.

Una alternativa que evita els accessos a memòria és oferir versions de les instruccions aritmètiques en les quals un operand és constant, amb la nova restricció de què aquesta constant es guarda dintre de la mateixa instrucció. Continuant amb la recomanació que demanava uniformitat, s'utilitza el mateix format per a eixes instruccions que per a les transferència de dades i instruccions de bot. De fet la I en el nom del format Tipus-I, ve d'immediat, el nom tradicional per a aquest tipus d'operands. El camp MIPS que conté la constant té 16 bits de grandària.

| *op* | *rs* | *rt* | *Immediat* |
|---|---|---|---|
| **8** | 29 | 29 | 4 |
| **001000** | 11101 | 11101 | 0000 0000 0000 1000 |

Figura 2.12.Exemple d'instrucció de Tipus-I, amb codificació decimal i binària.

Els operands immediats o constants també són habituals en comparacions. Com el registre $zero sempre val 0, ja se pot comparar amb 0. Per a comparar

altres valors, hi ha una versió amb immediat de la instrucció **set on less**. Per a comprovar si el registre $s2 és menor que la constant 10, se pot simplement escriure:

*slti $t0, $s2, 10  # $t0 = 1 si $s2 < 10.*

El mode de adreçament immediat il·lustra l'últim principi de disseny de circuiteria, abans esmentat:

*Principi de disseny 4: Fer ràpid el cas més comú.*

Els operands constants, apareixen amb freqüència i situar-los dins de les instruccions aritmètiques fa que s'executen molt més ràpidament.

Encara que les constants són habitualment xicotetes i caben en un camp de 16 bits, de vegades són més grans. El repertori d'instruccions del MIPS inclou la instrucció **load upper inmediat** (lui) específicament per a emmagatzemar els 16 bits de la part alta d'una constant en un registre, permetent a la instrucció subsegüent especificar els 16 bits més baixos de la constant. En la figura 2.12 es pot veure l'operació lui.

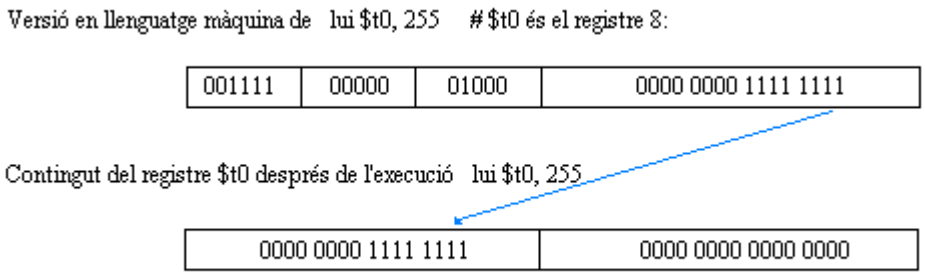


Figura 2.12.

## 2.8.2. Adreçament en bots condicionals i incondicionals.

L'adreçament simple es troba en les instruccions **jump** de MIPS. Aquestes instruccions utilitzen l'últim format d'instrucció del MIPS, anomenat tipus-J, que consisteix en 6 bits per al camp d'operació i la resta de bits per al camp d'adreça. D'aquesta manera,

*j 10000  # Anar a la localització 10000.*

s'ensambla amb aquest format, en decimal:

| 6 bits | 26 bits |
|---|---|
| 2 | 10000 |

on el valor del codi d'operació del bot incondicional és 2 i l'adreça del bot és 10000.

A diferència de la instrucció de bot incondicional, la de bot condicional deu especificar dos operands a més a més de l'adreça de bot. Doncs així,

*bne $s0, $s1, Fi  # Anar a Fi si $s0 != $s1.*

s'ensambla en esta instrucció, deixant sols 16 bits per a l'adreça de bot:

| 6 bits | 5 bits | 5bits | 16 bits |
|---|---|---|---|
| 5 | 16 | 17 | Fi |

Si les adreces del programa tingueren que cabre en aquest camp de 16 bits, significaria que cap programa podria ser més gran $2^{16}$, que és lluny de ser una opció realista hui en dia. Una alternativa seria especificar un registre que s'afegira sempre a l'adreça de bot, i per tant les instruccions de bot calcularien el següent:

Comptador de Programa = Registre + Adreça de Bot

Aquesta suma permet al programa arribar als $2^{32}$ bytes i ser capaç encara d'utilitzar bots condicionals, resolent el problema de tamany de les adreces de bot. La qüestió ara és, quin registre?

La resposta apareix al veure com s'utilitzen els bots condicionals. Els bots condicionals es troben en bucles i en sentències **if**, pel que tenen tendència de botar a instruccions properes. Per exemple, quasi la meitat de tots els bots condicionals al **gcc** i al **Spice**, van a parar a localitzacions no més enllà de 16 instruccions. Ja què el comptador de programa (PC) conté l'adreça de la instrucció actual, se pot botar dins del rang de paraules $\pm 2^{15}$ de la instrucció actual, si s'utilitza el PC com el registre a afegir a l'adreça. Quasi tots els bots i sentències **if** són molt més xicotetes que $2^{16}$ paraules, per tant el PC és l'elecció ideal.

Aquesta manera d'adreçament de bot s'anomena **mode de adreçament relatiu al PC**. Com més endavant es vora, és convenient per a la circuiteria augmentar el PC prompte per a apuntar a la següent instrucció. Així que l'adreça MIPS és realment relativa a l'adreça de la següent instrucció (PC + 4), en lloc de l'adreça actual (PC).

De la mateixa manera que moltes de les màquines recents, el MIPS utilitza el mode d'adreçaments relatiu al PC per a tots els bots condicionals, perquè el destí d'eixes instruccions és susceptible de ser prop del bot. D'altra banda, les instruccions **jump-and-link** invoquen procediments que no tenen raó de ser prop de la crida, i per tant s'utilitzen normalment altres modes d'adreçament. D'igual manera que l'arquitectura MIPS ofereisca adreces llargues per a crides a procediments, utilitzant el format tipus-J per a les instruccions **jump** i **jum-and-link**.

Com totes les instruccions MIPS tenen 4 bytes de tamany, el MIPS allarga la distància del bot mitjançant l'adreçament relatiu al PC, referit al nombre de paraules de la següent instrucció, en lloc del nombre de bytes. Doncs així, el camp de 16 bits pot botar quatre vegades més lluny, interpretant el camp com una adreça relativa de paraula, en lloc d'una adreça relativa de byte. L'adreçament relatiu de paraula és la raó per la qual cosa les versions del llenguatge màquina de **beq** i **bne**, a la figura 2.6 , té 25 en els seus camps d'adreça en lloc de 100, com en les versions del llenguatge ensamblador.

El camp de 26 bits de la instrucció **jump** és també una adreça de paraula; açò significa que representa una adreça byte de 28 bits. Donat que el PC és de 32 bits, 4 bits deuen vindre d'algun altre lloc. L'adreça MIPS **jump** reemplaça sols els 28 bits de menor pes del PC, deixant els 4 bits de major pes del PC inalterables. El carregador i el muntador deuen anar amb molt de compte per a evitar situar un programa més enllà del límits de 256 MB (64 milions d'adreces). Donat que, en cas contrari, els bots incondicionals deuen ser reemplaçats per instruccions **jump register** precedides per altres instruccions per a carregar l'adreça de 32 bits completa en un registre.

### 2.8.3. Resum dels modes d'adreçament del MIPS.

S'han vist dues noves formes d'adreçament fins ara, les múltiples formes d'adreçament genèricament es coneixen com **modes d'areçament**. Els modes d'adreçament del MIPS són els següents:

1. Mode d'adreçament registre, on l'operand és un registre.

2. Mode d'adreçament base més desplaçament, on l'operand està en una localització de memòria, on l'adreça és la suma d'un registre i una constant present en la pròpia instrucció.

3. Mode d'adreçament immediat, on l'operand és una constant que apareix en la mateixa instrucció.

4. Mode d'adreçament relatiu al PC, on l'adreça és la suma del PC i la constant d'instrucció.

5. Mode d'adreçament pseudodirecte, on l'adreça de bot són els 26 bits de la instrucció concatenats amb els bits de major pes del PC.

S'ha de fer notar que una operació simple pot utilitzar més d'un mode d'adreçament. La instrucció **add**, per exemple, utilitza tant l'adreçament immediat (**addi**) com l'adreçament registre (add). La figura 2.13 il·lustra per a cada mode d'adreçament, com se localitza l'operand corresponent.

Si bé s'ha mostrat l'arquitectura MIPS amb adreces de 32 bits, pràcticament tots els microprocessadors (inclòs el MIPS) tenen extensions d'adreces de 64 bits. Aquestes extensions van nàixer com a resposta a les necessitats de programes més amplis. El procés d'extensió del repertori d'instruccions, permet ampliar les arquitectures de manera que es mantinga la compatibilitat dels programes, amb la següent generació de l'arquitectura.

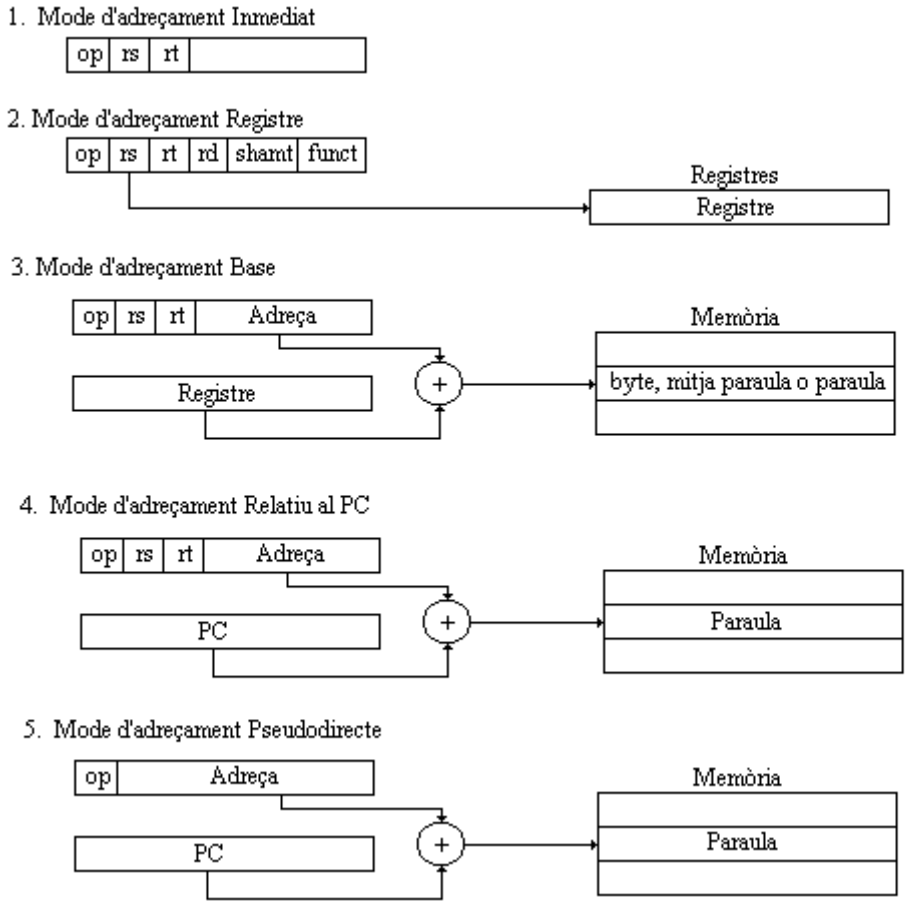


Figura 2.13. Modes d'adreçament del MIPS.

## 2.8.4. Descodificació del llenguatge màquina.

De vegades, ú es veu forçat a aplicar tècniques d'enginyeria al llenguatge màquina per a reconstruir el llenguatge ensamblador original. Un exemple és quan es mira un bolcat de memòria. La figura 2.14 mostra la codificació MIPS dels camps per al llenguatge màquina del MIPS. Aquesta figura es pot utilitzar per a traduir a mà entre el llenguatge ensamblador i el llenguatge màquina.

En la figura 2.15, es troba el llenguatge ensamblador de les instruccions del MIPS que encara queden per mostrar.

| op (31:26) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 28-26 / 31-29 | 0(000) | 1(001) | 2(010) | 3(011) | 4(100) | 5(101) | 6(110) | 7(111) |
| 0(000) | Format-R | Bltz/gez | jump | jal | beq | bne | blez | bgtz |
| 1(001) | addi | addiu | slt | situ | andi | ori | xori | lui |
| 2(010) | TLB | FlPt | | | | | | |
| 3(011) | | | | | | | | |
| 4(100) | lb | lh | lwl | lw | lbu | lhu | lwr | |
| 5(101) | sb | sh | swl | sw | | | swr | |
| 6(110) | lwc0 | lwcl | | | | | | |
| 7(111) | swc0 | swcl | | | | | | |

| op(31:26)=010000(TLB), rs(25:21) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 23-21 / 25-24 | **0(000)** | **1(001)** | **2(010)** | **3(011)** | **4(100)** | **5(101)** | **6(110)** | **7(111)** |
| **0(00)** | mfco | | cfc0 | | mtc0 | | ctc0 | |
| **1(01)** | | | | | | | | |
| **2(10)** | | | | | | | | |
| **3(11)** | | | | | | | | |

| op (31:26)=000000 (Format-R), funct (5:0) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2-0 / 5-3 | **0(000)** | **1(001)** | **2(010)** | **3(011)** | **4(100)** | **5(101)** | **6(110)** | **7(111)** |
| **0(000)** | sll | | srl | sra | sllv | | srlv | srav |
| **1(001)** | jr | jalr | | | syscall | break | | |
| **2(010)** | mfhi | mthi | mflo | mtlo | | | | |
| **3(011)** | mult | multu | div | divu | | | | |
| **4(100)** | add | addu | sub | subu | and | or | xor | nor |
| **5(101)** | | | slt | sltu | | | | |
| **6(110)** | | | | | | | | |
| **7(111)** | | | | | | | | |

Figura 2.14. Codificació de les instruccions MIPS.

Respecte a la figura 2.14, com es pot observar, aquesta notació proporciona el valor d'un camp per fila i per columna. Per exemple, a la part

superior de la figura **load word** es troba a la fila 4 ($100_2$ per als bits 31-29 de la instrucció) i a la columna 3 ($011_2$ per als bits 28-26 de la instrucció), per tant, el valor corresponent del camp op (bit 31-26) és $100011_2$. El subratllat significa que el camp s'utilitza en una altra part. Per exemple, **Format-R** a la fila 0 y columna 0 (op=$000000_2$) es defineix a la part inferior de la figura. Doncs així, **substract** a la fila 4 i a la columna de la secció inferior, significa que el camp **funct** (bit 5-0) de la instrucció, és $100010_2$, i el camp op (bits 31-26) és $0000000_2$.

| Categoria | Instrucció | Exemple | Significat | Comentaris |
|---|---|---|---|---|
| Transferència | load byte | lb $s1, 100($s2) | $s1=Mem[$s2+100] | Byte de mem. a registre |
| | store byte | sb $s1, 100($s2) | Mem[$s2+100] = $s1 | Byte de registre. a memòria |
| | load upper immediate | lui $s1, 100 | $s1=100*$2^{16}$ | Carrega constant en els 16 bits MSB |
| Bot condicional | set less than immediate | slti $s1,$s2,100 | si ($s2 < 100) $s1=1 sinó $s1=0 | Compara menor que amb constant |

Figura 2.15.

Al present projecte les operacions possibles han sigut sis, les quals són: add, sub, lw, sw, beq i j, perquè reflecteixen els tres tipus de formats de l'arquitectura MIPS (R, I i J).

Una vegada dit açò ens aproparem al funcionament de l'arquitectura i de les seues parts, arribant a la construcció d'un sistema de control que faja possible l'execució de les instruccions abans esmentades.

## 2.9. Construcció d'un camí de dades.

Una manera raonable de començar el disseny d'un camí de dades és examinar els components principals requerits per a executar tipus d'instrucció del MIPS. Primer es considereren els elements del camí de dades que necessita cadascuna de les instruccions i a partir d'ells es construeixen les diferents parts d'ell mateix. Una vegada que s'ha determinat els elements necessaris, s'estudiaran les senyals de control.

El primer element que se necessita és un lloc on guardar les instruccions dels programes. Pel tant, s'utilitza una unitat de memòria (un element d'estat) per a guardar i subministrar les instruccions a partir d'una adreça, tal com apareix a la figura 2.16. També es necessita un altre element d'estat per a guardar l'adreça de la instrucció, que rep el nombre de comptador de programa (PC) i es mostra

en la mateixa figura. Finalment, es necessita un sumador encarregat d'incrementar el PC per a què anote a l'adreça de la següent instrucció. Aquest sumador, que és combinatori, es pot construir a partir d'una ALU on les seues línies de control sempre especifiquen una operació de suma. El dibuix corresponent al d'una ALU amb l'etiqueta sumador (com es veu a la figura abans esmentada), per a indicar que és un sumador permanent i que no pot realitzar cap altra funció pròpia d'una ALU.

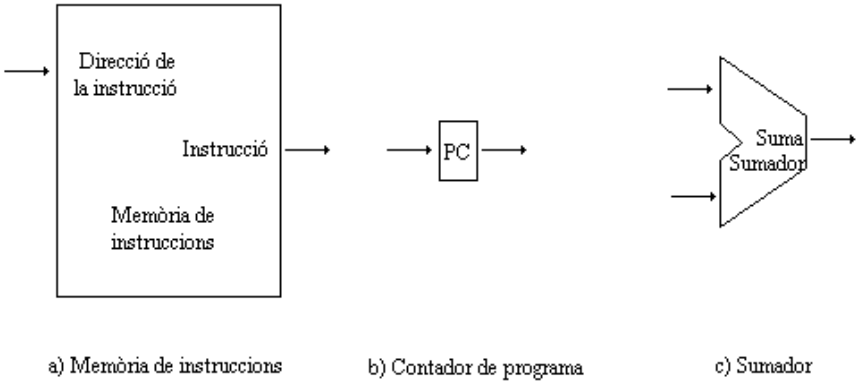


Figura 2.16.

Resumint, per a executar qualsevol instrucció, es deu començar per carregar la instrucció des de la memòria. Per a després poder executar la pròxima instrucció, es deu incrementar el comptador de programa per a què apunte 4 bytes més enllà. El camí de dades per a aquest pas, es mostra a la figura 2.17 i utilitza els tres elements de la figura 2.16.

Considerem ara les instruccions de Tipus-R, ja esmentades anteriorment. Totes elles lligen dos registres, operen amb l'ALU els continguts d'aquests registres, i escriuen el resultat. Aquestes instruccions s'anomenen de Tipus-R o **aritmètico-lògiques** (ja què realitze operacions aritmètiques o lògiques). En aquesta classe d'instruccions s'inclouen les instruccions **add**, **sub** i **slt**, introduïdes anteriorment, així com **and** i **or**. Hem de recordar també que l'exemple típic d'aquest tipus d'instrucció és:

*add $t1,$t2,$t3*

que llig $t2 i $t3 i escriu en $t1.

Els registres de 32 bits del processador s'agrupen en una estructura banc de registres. Un banc de registres és una col·lecció de registres, on en qualsevol es pot llegir o escriure especificant el seu nombre. El banc de registres conté l'estat dels registres de la màquina. A més a més, se necessita una ALU per a operar amb els valors llegits dels registres.

Tenint en compte que les instruccions de tipus R tenen tres operands registre, es necessita poder llegir dos dades del banc i escriure una en ell per a cadascuna de les instruccions. Per a cada registre que es llig, es necessita una entrada en el banc, on s'especifique el nombre de registre que es vol llegir així com una eixida del banc on es dipositarà l'esmentat valor. Per a escriure un valor es necessiten dues entrades, una per a especificar el **nombre de registre** on escriure i una altra per a subministrar la **dada**. El banc de registres, sempre retorna en les seues eixides els continguts dels registres, els identificadors dels quals, estan a les entrades dels registres a llegir. Les escriptures es controlen mitjançant una senyal de control d'escriptura, la qual deu estar activa per a què una escriptura es duga a terme en el flanc del rellotge. Resumint, es necessiten un total de quatre entrades (tres per als nombres dels registres i una altra per a les dades) i dues eixides (ambdues de dades), com es pot veure a la figura 2.18. Les entrades dels identificadors de registres són de 5 bits, per a especificar un dels 32 ($2^5$) mentre que els bussos de l'entrada i de les dues eixides de dades són de 32 bits.

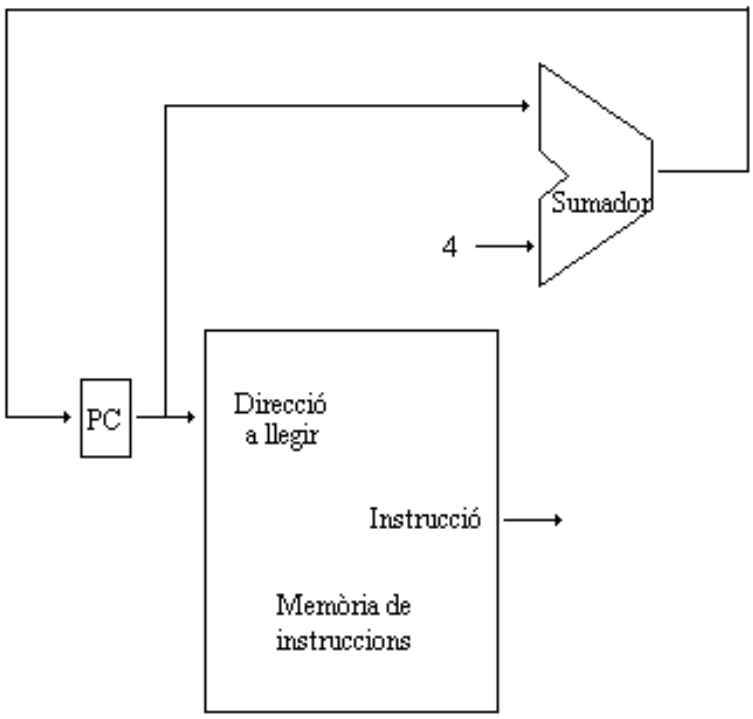


Figura 2.17.

L'ALU de la figura 2.18 es controla mitjançant una senyal de tres bits. L'ALU pren dues entrades de 32 bits i produeix un resultat de 32 bits.

A la figura 2.19 es mostra el camí de dades per a aquestes instruccions de tipus R, que utilitzen el banc de registre i l'ALU de la figura 2.18. Ja què els

nombres dels registres estan en camps de la instrucció, aquesta, que ve de la figura 2.17, es connecta a les entrades de nombre de registre del banc de registres.

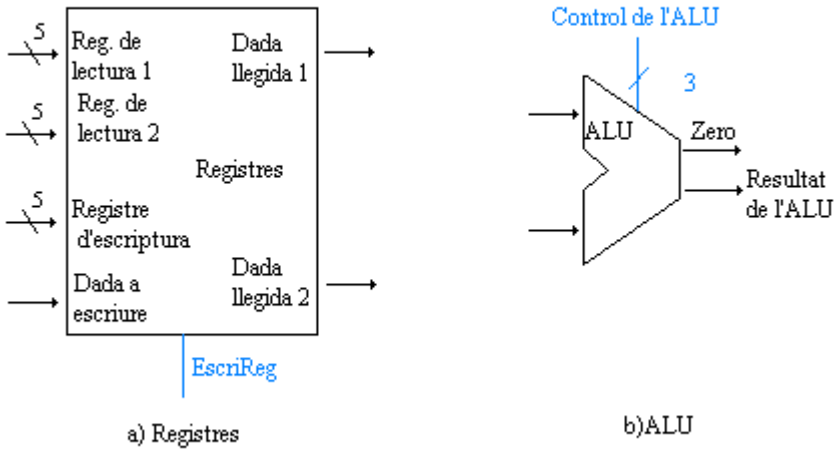


Figura 2.18.

Considerem ara les instruccions del MIPS de càrrega i emmagatzematge de paraules, les quals tenen el següent format:

lw $t1,despl($t2)
sw $t1,despl($t2)

Aquestes instruccions calculen l'adreça de memòria afegint al registre base ($t2) el camp de desplaçament (positiu o negatiu) de 16 bits contingut en la instrucció. Si la instrucció es un **store**, el valor a emmagatzemar, es deu llegir del banc de registres, on resideix $t1. En cas de **load**, el valor llegit de memòria, es deu escriure al registre especificat per $t1 al banc de registre. Resumint, es necessiten tant el banc de registres com l'ALU, mostrats a la figura 2.18.

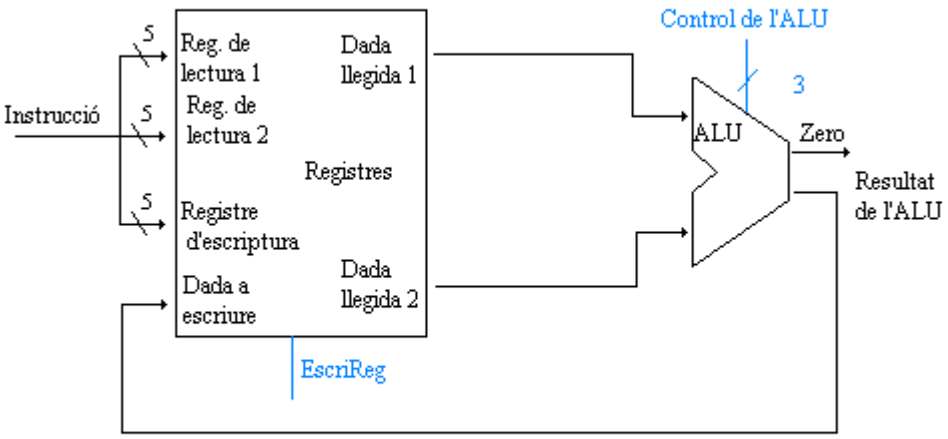


Figura 2.19.

A més a més es necessita una unitat per a extendre el signe del camp de desplaçament de 16 bits de la instrucció a un valor amb signe de 32 bits, i una unitat de memòria de dades per a llegir i escriure. La memòria de dades s'escriu amb instruccions **store**; pel que té senyals de control escriptura i de lectura, una entrada d'adreça i una altra de dades a escriure en memòria. La figura 2.20, mostra aquests dos elements.

A la figura 2.21, s'observa com es combinen aquests elements per a construir el camí de dades per a instruccions de **load** i **store**, suposant que ja s'ha obtés la instrucció. Els identificadors de registre per al banc, estan en els camps de la instrucció, així com el valor del desplaçament, el qual després d'extendre el signe, es converteix en el segon operand de l'ALU.

La instrucció **beq** té tres operands, dos registres que es comparen en igualtat, i un desplaçament de 16 bits, utilitzat per a calcular l'adreça destí del bot relativa a l'adreça de la instrucció. El seu format,

beq $t1,$t2,despl

Per a realitzar aquesta instrucció s'ha de calcular l'adreça destí del bot sumant el camp de desplaçament amb el signe extés al PC.

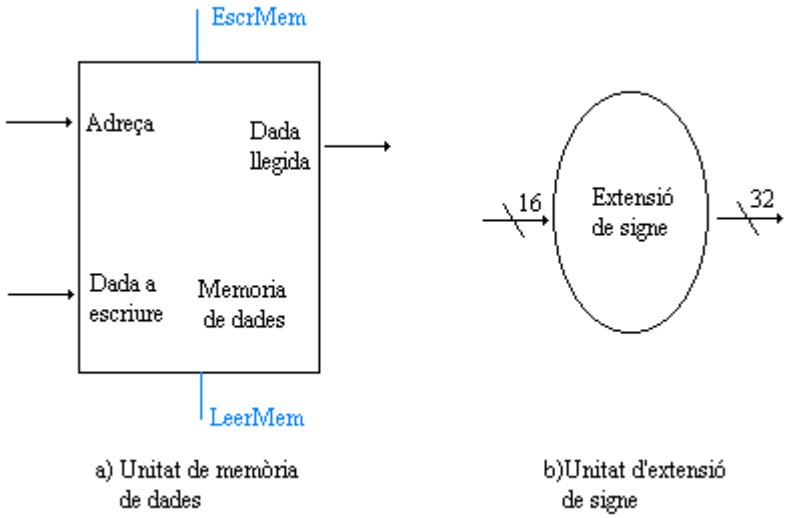


Figura 2.20.

Hi ha dos detalls en la definició de les instruccions de bot condicional als quals es deu prestar atenció:

- L'arquitectura del repertori d'instruccions especifica que és l'adreça de la següent instrucció en ordre seqüencial la que s'utilitza com a base per al càlcul de l'adreça destí. Ja què es calcula el PC + 4 (l'adreça de la pròxima instrucció) en el camí de dades de la càrrega d'instruccions, és fàcil utilitzar aquest valor com a base per al càlcul de l'adreça destí del bot.

- L'arquitectura també imposa que el camp de desplaçament es desplace cap a l'esquerra 2 bits per a què aquest corresponga a una paraula; de manera que s'incremente el rang efectiu de l'esmentat camp en un factor 4.

Per a tractar l'última complicació es necessita desplaçar dos posicions el camp de desplaçament.

A més a més de calcular l'adreça destí del bot, també es deu determinar si la pròxima instrucció a executar és la que segueix seqüencialment o la situada en l'adreça destí del bot. Quan la condició s'acompleix (és a dir, els operands són iguals), l'adreça calculada passa a ser el nou PC, i es diu que el bot condicional s'ha complit. Si els operands són diferents, el PC incrementat deuria reemplaçar al PC actual (igual que per a qualsevol altra instrucció); en aquest cas es diu que el bot no s'ha complit.

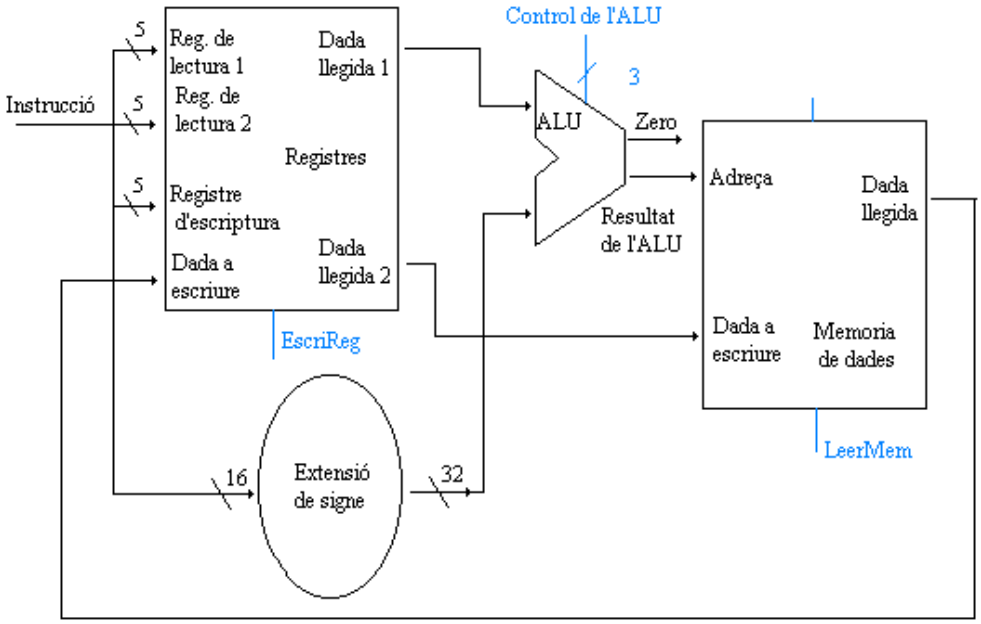


Figura 2.21.

Resumint, el camí de dades pe a bots condicionals, deu efectuar dos operacions:

- Calcular l'adreça destí del bot,
- Comparar el contingut dels registres (els bots condicionals també requereixen que es modifique la part de càrrega d'instruccions).

La figura 2.22 mostra el camí de dades dels bots condicionals. Per a calcular l'adreça destí del bot, el camí de dades inclou una unitat d'extensió de signe, com en la figura 2.20, i un sumador. Per a la comparació es necessita utilitzat el banc de registres mostrat a la figura 2.18 a la fi d'obtindre els dos operands registres (encara que no serà necessari escriure en ells) i l'ALU per a realitzar l'esmentada operació. Ja què aquesta ALU proporciona una senyal d'eixida que indica si el resultat era 0, se li pot enviar els dos operands amb la senyal de control activada de manera que realitze una resta. Si la senyal **Zero** a l'eixida de l'ALU està activa, aleshores els dos valors són iguals. Encara que l'eixida Zero sempre indica si el resultat és 0, sols s'utilitza per a realitzar el test d'igualtat en bots condicionals. En el capítol dedicat a la realització de l'arquitectura, s'estudiarà de manera més exacta com es connecten les senyals de control de l'ALU per a utilitzar-la al camí de dades.

La instrucció de bot incondicional reemplaça els 28 bits de menor pes del PC amb els 26 bits de menor pes de la instrucció desplaçats dos bits cap a

l'esquerra. Aquest desplaçament es realitza simplement concatenant 00 amb aquests 26 bits.

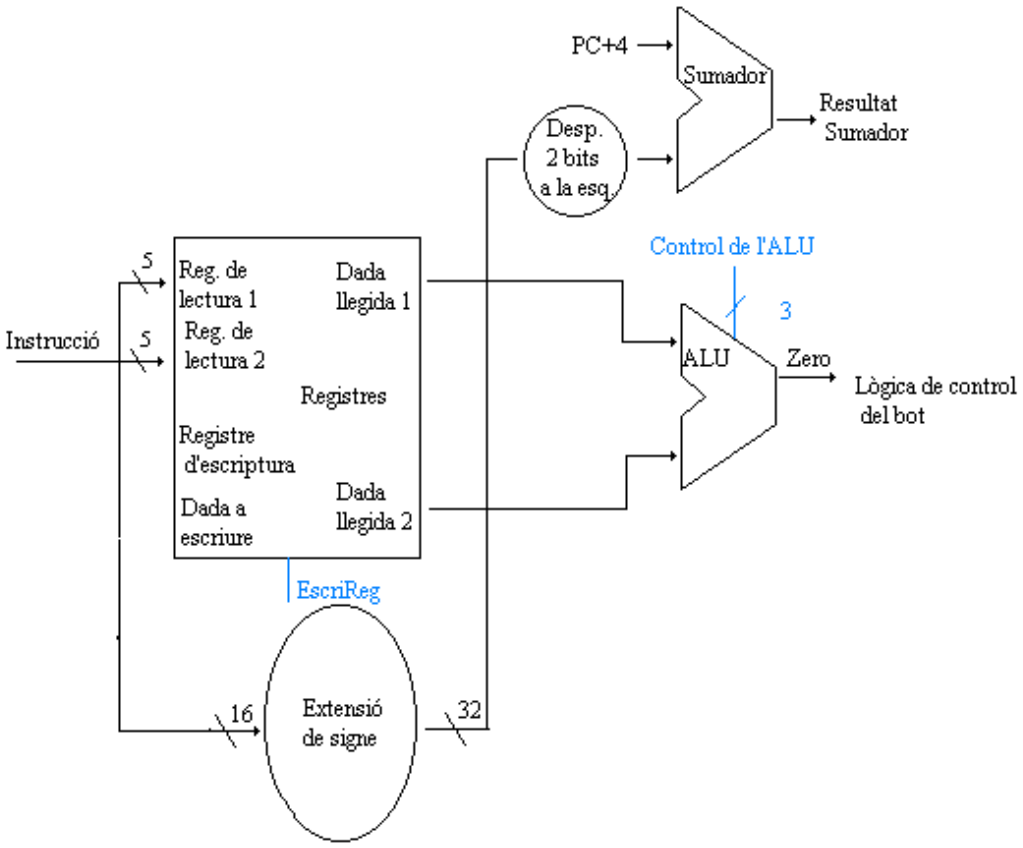


Figura 2.22.

Una vegada examinats els camins de dades necessaris segons el tipus d'instruccions, aquests poden combinar-se en un camí de dades únic i afegir el control per a completar la realització. Els camins de dades de les figures 2.17, 2.19, 2.21 i 2.22 són els blocs bàsics per a dues realitzacions diferents.

Al repertori d'instruccions del MIPS, els bots condicionals es retarden, la qual cosa significa que la següent instrucció immediata després del bot s'executa sempre, independentment que si la condició de bot s'acompleix o no. Quan la condició és falsa, l'execució es comporta com un bot normal. Quan és certa, un bot retardat, s'executa primer la instrucció immediata posterior al bot, i posteriorment bota a l'adreça destí. El motiu dels bots retardats surt pel mode en que les afecta la segmentació. Per a simplificar, s'ignoraran els bots retardats i es realitzarà una instrucció **beq** no retardada.

## 2.10. Construcció d'un camí de dades senzill.

La construcció d'un camí de dades a partir de les peces que s'han vist a les figures 2.17, 2.19, 2.21 i 2.22 es ben senzill. El més fàcil dels dissenys intentarà

executar totes les instruccions en un sol cicle (màquina unicicle). Açò significa que cap element del camí de dades pot utilitzar-se més d'una vegada per instrucció, de manera que qualsevol recurs que es necessite més d'una vegada deurà aparèixer dues vegades. Per tant, la memòria d'instruccions ha d'estar separada de la memòria de dades. Encara que es necessite duplicar algunes de les unitats funcionals, molts d'aquestos elements poden compartir-se en els diferents fluxos d'instruccions quan els camins de dades individuals de les figures anteriors es combinen.

Per a compartir un element del camí de dades entre dues classes d'instruccions diferents, es requereix que l'esmentat element dispose de múltiples entrades, així com d'una senyal de control que seleccione l'adequada en cada instant. Aquesta selecció es realitza normalment mitjançant un dispositiu anomenat **multiplexor** encara que el seu nom més correcte deuria ser **selector de dades**.

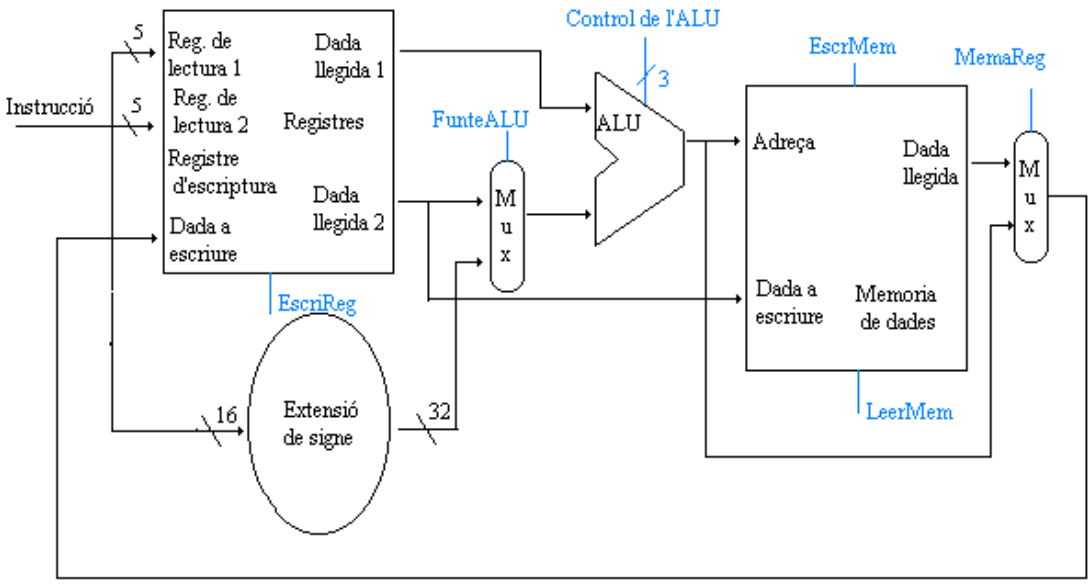


Figura 2.23.

El camí de dades de les instruccions aritmètico-lògiques (o tipus R) que apareix a la figura 2.19, així com el de les instruccions d'accés a la memòria de la figura 2.21, són molt pareguts, sent les principals diferències les següents:

- La segona entrada de l'ALU és, o bé un registre en cas d'una instrucció aritmètico-lògica, o bé els bits de menor pes d'una instrucció de memòria amb el signe extés.

- El valor emmagatzemat al registre destí, o bé prové de l'ALU (per a instruccions de tipus R) o de la memòria (en cas de loads).

Per a combinar ambdós camins de dades i utilitzar un únic banc de registres i una sola ALU, la segona entrada d'aquesta, a de suportar dos tipus de dades diferents, a més de dos possibles camins per a la dada a emmagatzemar al banc de registres. D'aquesta manera, es col·loca un multiplexor en l'entrada de l'ALU i un segon a l'entrada de dades del banc de registres. La figura 2.23 mostra aquest nou camí de dades.

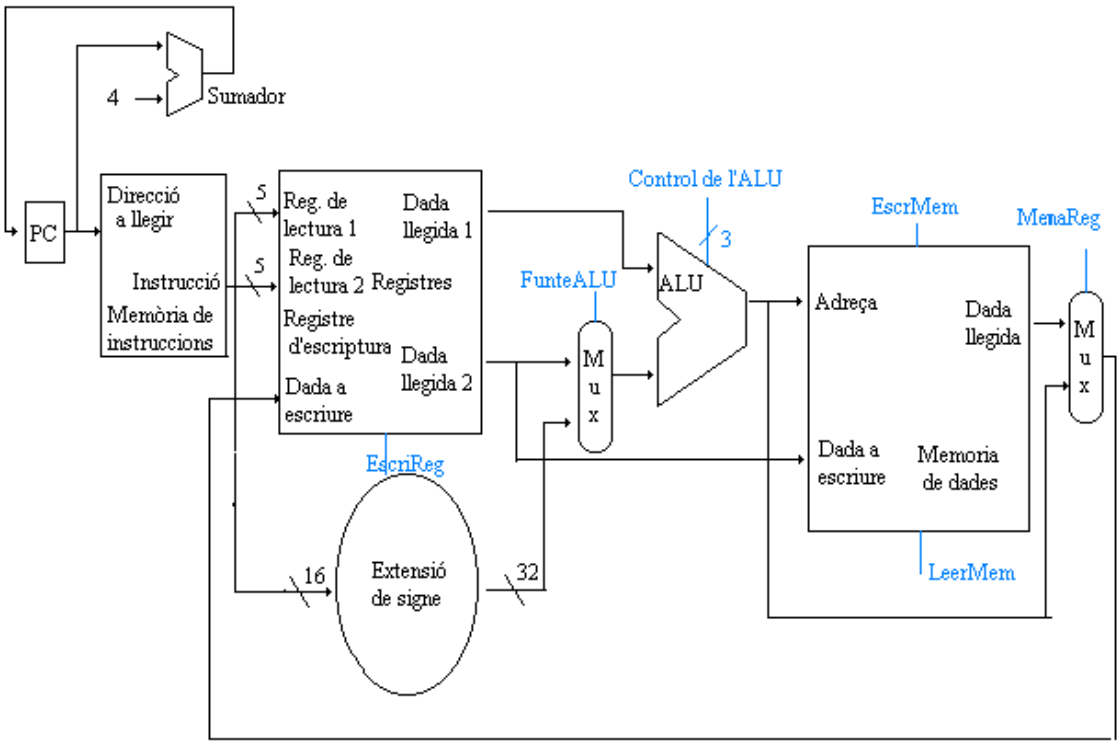


Figura 2.24.

La part del camí de dades encarregada de la búsqueda d'instruccions (figura 2.17), pot afegir-se fàcilment a aquest nou camí de dades. A la figura 2.24 pot veure's el resultat obtés. Aquest nou camí de dades té memòries separades per a dades i per a instruccions, així com un sumador independent de l'ALU, el qual es va a utilitzar per a incrementar el PC mentre que l'ALU executa la instrucció en el mateix cicle del rellotge.

Ara es poden combinar la resta de les peces per a obtenir un camí de dades únic per a l'arquitectura MIPS, afegint la part encarregada de l'execució de les instruccions de bot de la figura 2.22. La figura 2.25 mostra aquest camí de dades obtés al ajuntar les diferents parts. Les instruccions de bot utilitzen l'ALU per a la comparació dels registres font, de manera que deu posar-se el sumador de la figura 2.22 per a calcular l'adreça de destí del bot. També és necessari un nou multiplexor per a escollir entre seguir en seqüència (PC + 4) i l'adreça destí del bot per a escriure aquesta nova adreça al PC.

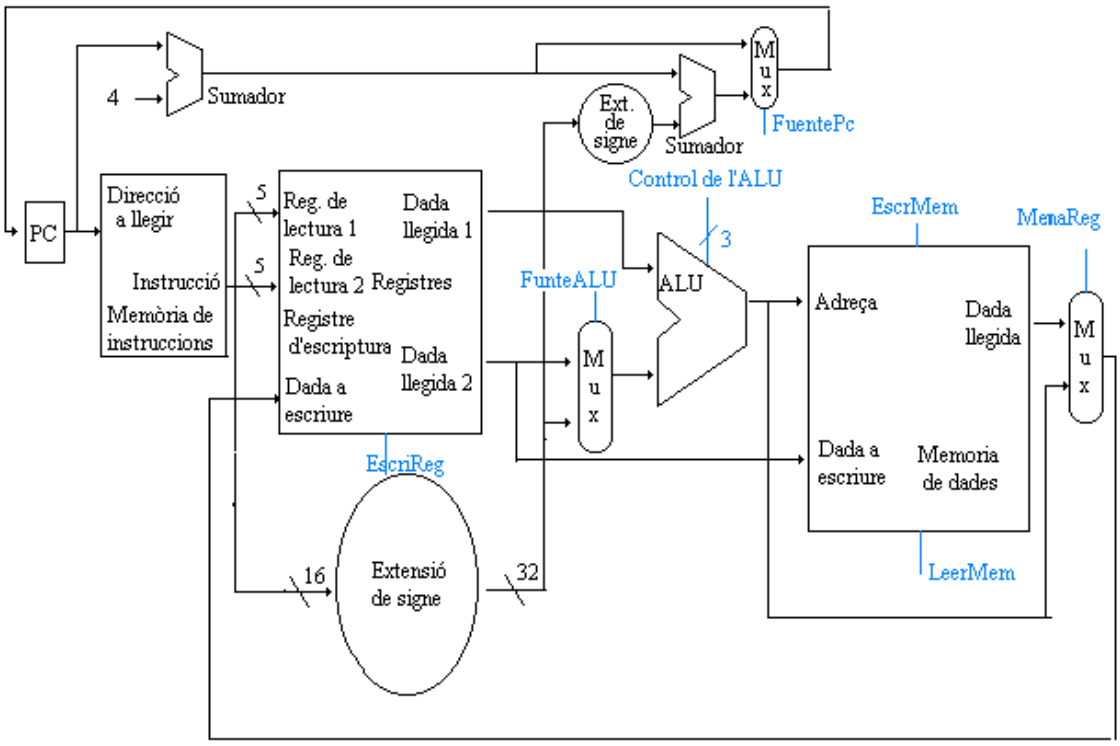


Figura 2.25.

Una vegada completat aquest senzill camí de dades, es pot afegir la unitat de control. Aquesta deu ser capaç de generar les senyals d'escriptura per a cadascun dels elements d'estat i les de control per a l'ALU i per a cadascun dels multiplexors a partir de les entrades. Ja què el control de l'ALU és diferent de la resta en major o menor mesura, resultaria útil dissenyar-lo com a pas previ al disseny de la unitat de control.

### 2.10.1. El control de l'ALU.

L'ALU té tres senyals de control com a entrades, encara que sols s'utilitzen cinc de les vuit combinacions possibles. La taula següent mostra les cinc possibles combinacions.

| ALU control input | Funció |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | suma |
| 110 | resta |
| 111 | activar si menor que |

Depenent del tipus d'instrucció a executar, l'ALU deu realitzar una d'aquestes cinc operacions. Les instruccions d'accés a la memòria utilitzen l'ALU per a calcular l'adreça de la memòria mitjançant una suma. Per a les instruccions de tipus R, l'ALU ha d'executar una de les cinc operacions (**AND**, **OR**, **add**, **sub** i **slt**) en funció del valor dels 6 bits de menor pes de la instrucció, els quals composen el codi de funció, mentre que per a les instruccions de **botar si igual**, l'ALU deu realitzar una resta.

| Codid'operació de la instrucció | ALUOp | Operació | Camp de funció | Acció desitjada de l'ALU | Entrada del control de l'ALU |
|---|---|---|---|---|---|
| lw | 00 | Load word | XXXXXXX | suma | 010 |
| sw | 00 | Store word | XXXXXXX | suma | 010 |
| Branch equal | 01 | Branch equal | XXXXXXX | resta | 110 |
| R-type | 10 | Suma | 100000 | suma | 010 |
| R-type | 10 | Resta | 100010 | resta | 110 |
| R-type | 10 | AND | 100100 | AND | 000 |
| R-type | 10 | OR | 100101 | OR | 001 |
| R-type | 10 | Activar si menor que | 101010 | Activar si menor que | 111 |

Figura 2.26.

Mitjançant una xicoteta unitat de control que té com a entrades el codi de funció de la instrucció i dos bits de control addicionals que reben el nom de ALUOp, es pot generar els tres bits que configuren les senyals de control de l'ALU. Aquests bits indiquen si l'operació a realitzar tindria que ser o bé una suma (00) per a accessos a la memòria, o bé una resta (01) per a bots condicionals, o bé si l'operació a realitzar està codificada al codi de funció (10). L'eixida de la unitat de control de l'ALU és una senyal de tres bits, que controla l'ALU codificant una de les cinc combinacions mostrades a la taula d'abans.

A la figura 2.26 es pot observar el conjunt de combinacions de les senyals d'entrada formades pels dos bits que conformen la senyal d'ALUOp i els 6 bits del codi de funció. Finalment, també es mostra la relació existent entre els codis d'operació de les instruccions i el camp ALUOp. Posteriorment, es veurà com es genera la unitat de control principal els bits de la senyal ALUOp.

Aquesta tècnica basada en l'ús de múltiples nivells de descodificació (és a dir, la unitat de control principal genera els bits d'ALUOp, que s'utilitzaran com a entrada de la unitat de control encarregada de generar les senyals per a l'ALU) és molt comú. El fet d'utilitzar múltiples nivells pot reduir el tamany de la unitat principal. De la mateixa manera, l'ús de múltiples unitats de control més xicotetes pot, inclús, incrementar la velocitat de l'esmentada unitat de control. Totes aquestes optimitzacions són importants ja què, amb freqüència, la unitat de control es troba al camí crític.

Existeixen diferents maneres d'establir la correspondència entre els dos bits del camp de ALUOp i els 6 bits de codi de funció amb els 3 bits que conformen l'operació a realitzar per l'ALU. Com a conseqüència que sols una xicoteta part dels 64 valors possibles del codi de funció són importants i que els esmentats bits únicament s'utilitzen quan ALUOp val $10_2$, podria utilitzar-se una menuda part de lògica que reconeguera aquest subconjunt de valors possibles i donara com a resultat els valors correctes dels bits de control de l'ALU.

Com a pas previ al disseny de la lògica combinatòria, és útil construir una taula de veritat per a aquelles combinacions d'interés dels codis de funció i dels bits d'ALUOp, tal com s'ha realitzat a la figura 2.27. Aquesta taula mostra com depén d'ambdós camps les senyals de control de l'ALU. Encara que la taula de veritat és molt gran ($2^8 = 256$ entrades) i tenint en compte que per a moltes de les esmentades combinacions, els valors de l'eixida no tenen importància, únicament es donen els valors de les eixides per a aquelles entrades de la taula on el control de l'ALU deu tindre un valor específic.

| ALUOp | | Camp de la funció | | | | | | Operació |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp 2 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 010 |
| X | 1 | X | X | X | X | X | X | 110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 110 |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 000 |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 111 |

Figura 2.27.

Tenint en compte que en molts casos alguns valors de les entrades no són importants, es pretén que siguen indeterminats. Un terme d'aquest tipus (representat a la taula mitjançant una X en la columna d'entrada corresponent) indica que a l'eixida és independent d'aquesta entrada. Per exemple, quan el camp ALUOp val 00, cas de la primera fila de la taula de la figura 2.27, la senyal de control de l'ALU sempre serà 010 independentment del codi de funció. És a dir, en aquest cas el codi de la funció es considera indeterminat en aquesta fila de la taula de veritat.

## 2.10.2. Disseny de la unitat de control principal.

Una vegada que s'ha descrit el disseny de la unitat de control de l'ALU, la qual té com a entrades els codis de funció i una senyal de 2 bits, passem a centrar-nos en la resta del control. Per a iniciar aquest procés s'han d'identificar els camps de les instruccions i les línies de control necessàries per a la construcció del camí de dades mostrat a la figura 2.25. Per a entendre millor com

es connecten els diferents camps de les instruccions al camí de dades, seria útil revisar els formats dels tres tipus d'instruccions:

- Tipus R (aritmètico-lògica),
- Els bots,
- Les operacions de **load** i **store**.

Aquests formats es troben a la figura 2.28.

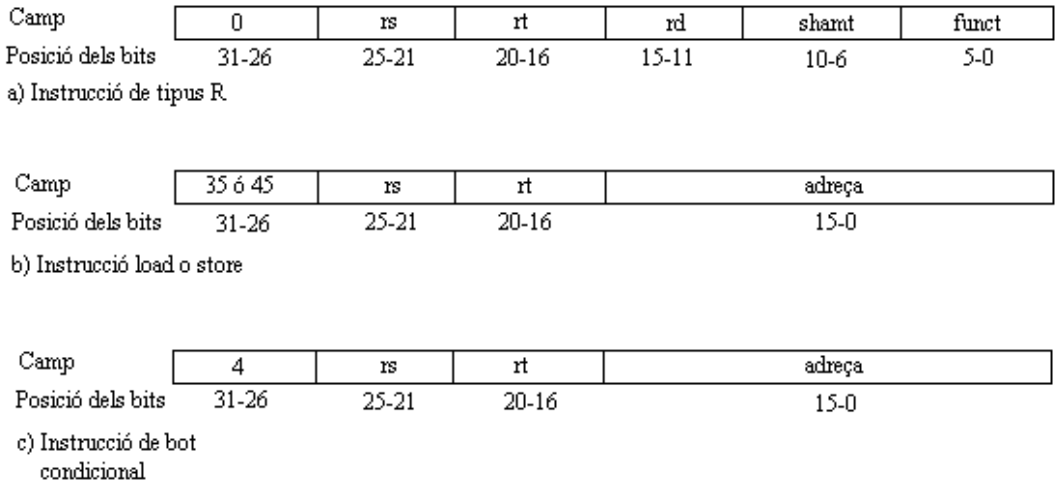


Figura2.28.

Existeixen moltes observacions importants a realitzar d'aquest format de les instruccions i que sempre s'assumiran com a certes.

- El camp del tipus d'operació estarà sempre dintre dels bits 31-26. Es referenciarà aquest camp com Op[5-0].

- Els dos registres de lectura són sempre especificats als camps rs i rt en les posicions 25-21 i 20-16. Aquest fet s'acompleix per a les instruccions de tipus R, **beq** i **store**.

- El registre base per a les instruccions d'accés a memòria es troba sempre en **rs** (bits 25-21).

- El desplaçament relatiu de 16 bits per a **botar si igual** (beq), loads i stores, és sempre a la posició 15-0.

- El registre destí pot estar en dos llocs. Per a instruccions de **load**, la seua posició és 20-16 (**rt**), mentre que en les instruccions artimètico-

lògiques, es troba als bits 15-11 (**rd**). Açò implica tindre que afegir un multiplexor per a seleccionar quin d'aquests dos camps de la instrucció va a utilitzar-se per a indicar el registre on escriure.

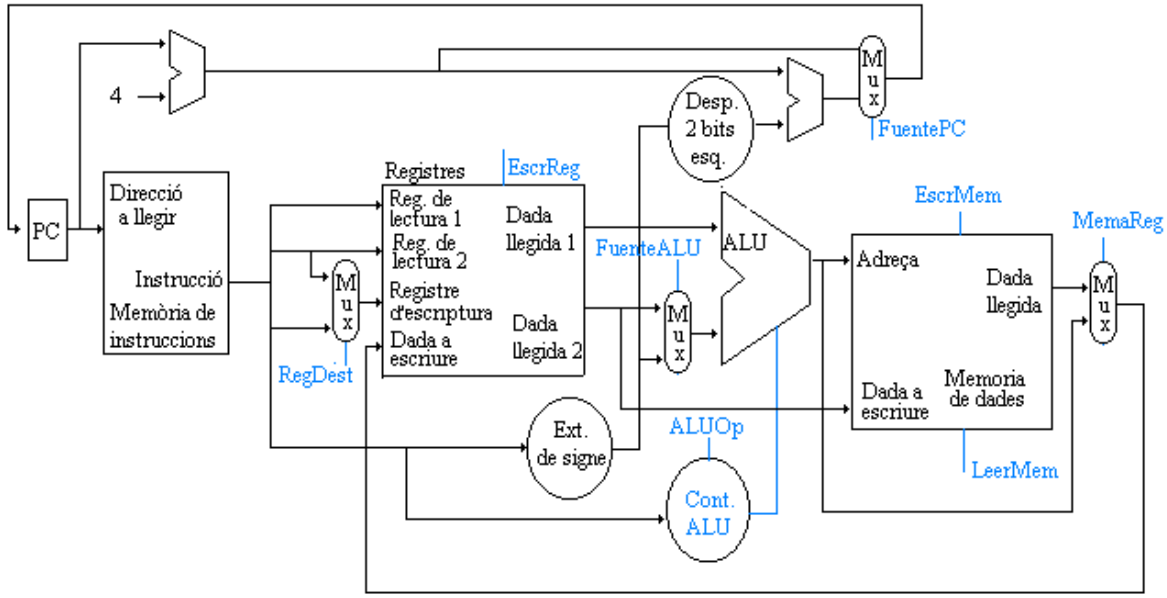


Figura 2.29.

Mitjançant aquesta informació es pot afegir les etiquetes de les diferents parts de les instruccions i el multiplexor addicional (per al registre destí) al nostre senzill camí de dades. La figura 2.29 mostra aquests afegits, que inclouen la unitat de control de l'ALU les senyals d'escriptura dels elements d'estat. La senyal de lectura de la memòria de dades i les senyals de control dels multiplexors. Encara que aquests últims únicament tenen dues entrades, sols requereixen una única línia de control cadascú. La figura 2.29 mostra les set senyals de control d'1 bit a les que hi ha que afegir la senyal d'ALUOp (de 2 bits). El funcionament d'aquesta senyal ja s'ha explicat i ara seria útil detallar-lo informalment per a la resta de les senyals abans de determinar com van a treballar durant l'execució de les instruccions. La figura 2.30 descriu la funcionalitat d'aquestes línies de control.

Una vegada definides les funcions de cadascuna de les senyals de control, es pot passar a vore com activar-les. La unitat de control pot activar totes les senyal dels codis d'operació de les instruccions exceptuant una d'elles (la senyal de FuentePC). Aquesta línia de control deu activar-se si la instrucció és de tipus bot condicional (decisió que pot prendre la unitat de control) i l'eixida de Zero de l'ALU que s'utilitza en les comparacions està a 1. Per a generar la senyal de

FuentePC es necessita aplicar una AND a una senyal anomenada SaltoCond, que ve de la unitat de control amb la senyal de Zero procedent de l'ALU.

| Senyal de Control | Efecte quan no és activa | Efecte quan no és activa |
|---|---|---|
| RegDest | L'identificador del registre destí ve determinat pel camp rd (bits 15-11). | L'identificador del registre destí ve determinat pel camp rt (bits 20-16). |
| EscrReg | Ningú. | El registre destí s'actualitza amb el valor a escriure. |
| FuenteALU | El segon operand de l'ALU són els 16 bits de menor pes de la instrucció amb el signe extés. | El segon operand de l'ALU prové del segon registre llegit del banc de registres. |
| FuentePC | El PC és reemplaçat per l'eixida del sumador, que calcula l'adreça destí del bot. | El PC és reemplaçat pel seu valor anterior més quatre. |
| LeerMem | Ningú. | El valor de la posició de la memòria designada per l'adreça es col·loca a l'eixida de la lectura. |
| EscrMem | Ningú. | El valor de la posició de la memòria designada per l'adreça es reemplaça pel valor de l'entrada de dades. |
| MemaReg | El valor d'entrada del banc de registres prové de l'ALU | El valor d'entrada del banc de registres prové de la memòria. |

Figura 2.30.

Aquestes nou senyals de control (les set de la figura 2.30 més les dues de la senyal ALUOp), poden activar-se en funció de les sis senyals d'entrada de la unitat de control, les quals pertanyen al codi d'operació. El camí de dades amb la unitat i les senyals de control es mostra a la figura 2.32.

| Instrucció | RegDest | FuenteALU | MemaReg | EscReg |
|---|---|---|---|---|
| Format-R | 1 | 0 | 0 | 1 |
| lw | 0 | 1 | 1 | 1 |
| sw | X | 1 | X | 0 |
| beq | X | 0 | X | 0 |

| LeerMem | EscrMem | SaltoCond | ALUOp1 | ALUOp0 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |

Figura 2.31.

Abans d'intentar escriure el conjunt d'equacions de la taula de veritat de la unitat de control seria útil definir-la informalment. Com que l'activació de les línies de control únicament depén del codi d'operació, es defineix si cadascuna de les senyals de control deuria valdre 0,1 o bé indeterminada (X), per a cadascun dels codis d'operació. La figura 2.31 defineix com deuen activar-se les senyals de control per a cada codi d'operació; informació que prové directament de les figures 2.26, 2.30 i 2.32.

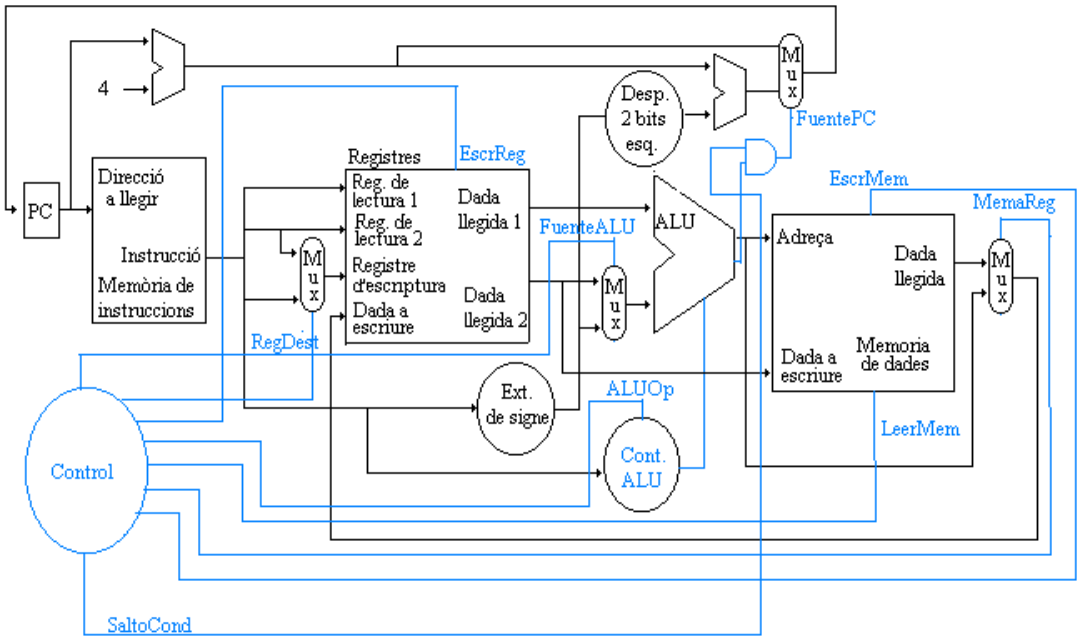


Figura 2.32.

I per últim, cal esmentar que per donar-li un poc de complexitat al simulador, s'ha afegit la instrucció jump, la qual té un cert semblant amb la instrucció de bot condicional, però es calcula l'adreça destí d'una manera diferent, i a més, és incondicional. Com als bots, els dos bits de menor pes de l'adreça del bot són sempre $00_{dos}$. Els següents 26 bits de menor pes de l'adreça es troben en el camp immediat de la instrucció, tal com es mostra a la figura 2.33. Els 4 bits de major pes de l'adreça que deuria reemplaçar el PC venen del PC de la instrucció al qual s'ha sumat el valor 4. Així es podria realitzar un **jump** emmagatzemat al registre de PC la concatenació de:

- Els 4 bits de major pes d l'actual PC + 4 (són els bits 31-28 de l'adreça de la següent instrucció en ordre seqüencial).

- Els 26 bits corresponents al camp immediat de la instrucció jump.

- Els bits $00_{dos}$.

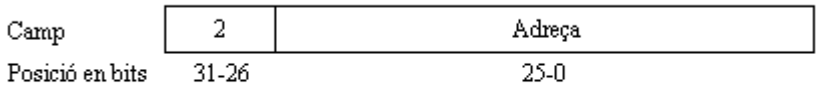


Figura 2.33

La figura 2.34, de la pàgina següent, mostra les parts afegides al control per a d'aquest tipus de instruccions respecte a la figura 2.31. Es necessita un nou multiplexor per a seleccionar l'origen del nou PC: l'adreça de la següent instrucció en ordre seqüencial (PC + 4), l'adreça del bot condicional o la d'una instrucció jump. També es necessita una nova senyal de control per a aquest multiplexor. Aquesta senyal, anomenada SaltoIncond, únicament s'activa quan la instrucció és un jump, és a dir, quan el codi d'operació és 2.

A la figura 2.35 podem observar el control per a una màquina unicicle.

| Entrada o eixida | Senyal | Format-R | lw | sw | beq | jump |
|---|---|---|---|---|---|---|
| Entrades | Op5 | 0 | 1 | 1 | 0 | 0 |
| | Op4 | 0 | 0 | 0 | 0 | 0 |
| | Op3 | 0 | 0 | 1 | 0 | 0 |
| | Op2 | 0 | 0 | 0 | 1 | 0 |
| | Op1 | 0 | 1 | 1 | 0 | 1 |
| | Op0 | 0 | 1 | 1 | 0 | 0 |
| Eixides | RegDest | 1 | 0 | X | X | 0 |
| | FuenteALU | 0 | 1 | 1 | 0 | 0 |
| | MemaReg | 0 | 1 | X | X | 0 |
| | EscrReg | 1 | 1 | 0 | 0 | 0 |
| | LeerMem | 0 | 1 | 0 | 0 | 0 |
| | EscrMem | 0 | 0 | 1 | 0 | 0 |
| | SaltoCond | 0 | 0 | 0 | 1 | 0 |
| | ALUOp | 1 | 0 | 0 | 0 | 0 |
| | ALUOp | 0 | 0 | 0 | 1 | 0 |
| | SaltoIncond | 0 | 0 | 0 | 0 | 1 |

Figura 2.35.

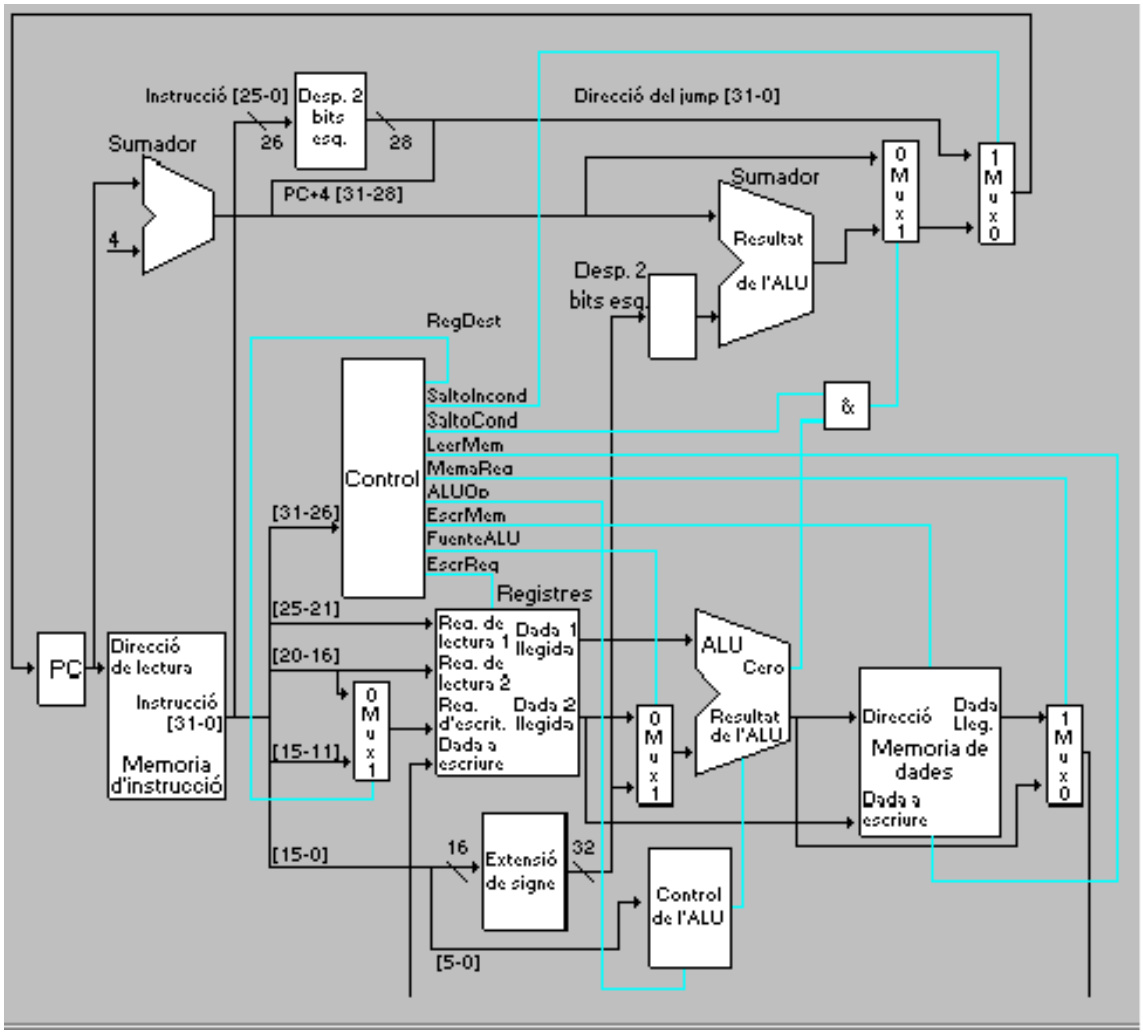


Figura 2.34.

## 2.11. Realització Multicicle.

Com ja s'ha exposat al capítol anterior, la realització multicicle suposa la "partició" de l'execució total de la instrucció en diferents passos. La diferència que hi ha entre aquests nous passos és que ara realment la instrucció s'executa per *hardware*, és a dir que si físicament muntem l'arquitectura multicicle, la instrucció comptarà amb una sèrie de passos que si bé són pràcticament iguals als passos que el programa d'aquest projecte, res tenen a veure, ja què, com ja s'ha dit abans, a la realització unicicle la instrucció es realitza en tan sols un pas.

Degut a aquest motiu, els nombre d'elements de la realització multicicle és menor que el cas de la realització unicicle, com ja s'havia comentat a la introducció. Açò no lleva, però, que la complexitat a l'hora del muntatge físic siga més difícil, així com també la pròpia realització per *software*.

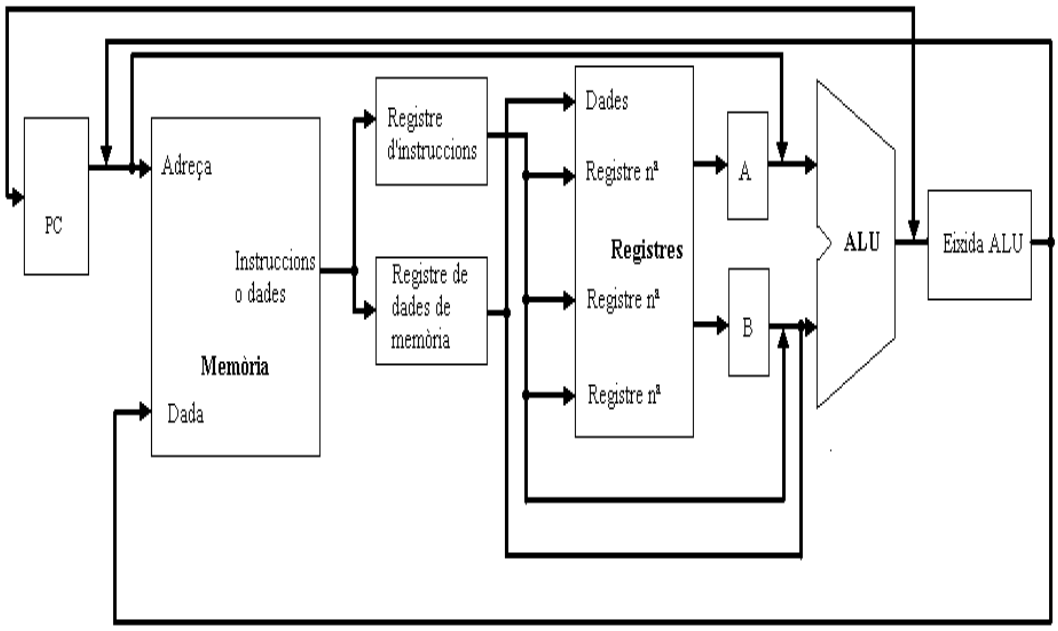


Figura 2.35.

Com ja s'ha dit abans a la introducció, hi ha unes diferències entre ambdues tipus de realitzacions, tal com es veu a les figures 2.35 i :2.25:

- La unitat de memòria és única i s'utilitza tant per a instruccions com per a dades.

- Hi ha una sola ALU en compte d'una ALU i dos sumadors.

- S'han afegit un o més registres després de cada unitat funcional per a emmagatzemar l'eixida d'aquestes unitats fins que aquest valor siga utilitzat en el següent cicle.

Com que la realització ha sigut la de tipus unicicle, no cal extendre més el tipus de realització multicicle.