La segmentació

1. Introducció

A l'hora d'establir una millora en el rendiment es podria dur a terme mitjançant una disminució dels factors que intervenen en el càlcul del mateix: CPI(Cicles per instrucció), TC(Temps de cicle) o NI (Número d'instruccions). Els tres factors no són independents però el canvi d'un d'ell afecte al rendiment. Així tenim que els cicles d'instruccions depèn de com s'organitzen les mateixes, el nombre d'instruccions està fortament del compilador i el repertori d'instruccions que disposem, i el cicles de rellotge depèn de la tecnologia hardware que utilitzem.

El temps de CPU que consta l'execució d'un conjunt d'instruccions es podria expressar com:

Tcpu=CPI x TC x NI on CPI= NC / NI NC= num. de cicles per l'execució.

La solució per millorar el rendiment sense tenir que canviar de tecnologia i suposant que el nombre d'instruccions que genera el compilador és la mateixa només passaria per reduir el CPI i això és el que intentarem veure en aquesta materia. Això es veurà mitjançant la utilització de tres tipus d'arquitectures avançades: La segmentació, la supersegmentació i els superescalars.

La segmentació és una tècnica d'implementació en la qual múltiples instruccions es solapen durant l'execució. Amb la segmentació no es millora el temps d'execució d'una instrucció però si es millora la productivitat de les instruccions, és a dir augmenta el nombre d'instruccions que executen per unitat de temps.

Les instruccions es divideixen en trossos petits que es poden executar independentment anomenats segments. Aquesta divisió ens permet que mentre una instrucció acabi d'executar un segment pugui entrar una instrucció per executar-se.
Això implicarà que totes les etapes estiguin connectades entre si i totes elles han d'estar preparades per procedir en el mateix temps i això implica que la velocitat amb què surten les instruccions ha d'ésser no pot excedir de la velocitat amb la qual entren.

El temps ideal de pas de l'execució d'un segment a un altre d'una instrucció seria 1 cicle de rellotge i per tant la duració d'un cicle de rellotge vindrà determinada pel temps necessari per a l'etapa més lenta de la segmentació.

A l'hora de dissenyar una segmentació la cosa que s'intenta és equilibrar la duració de cada etapa fent que el temps de d'interactivitat que es pugui produir entre etapes sigui el mínim.

En condicions ideals suposant que no hi ha esperes entre etapes tindrem que el temps entre instruccions segmentades seria :

$$\frac{\textit{Temps entre instruccions}_{\text{ no segmentada}}}{\textit{Número d'etapes de segmentació}}$$

En aquestes condicions la millora de velocitat amb la segmentació serà igual al nombre d'etapes. Cal tenir en compte que en la realitat és molt difícil equilibra de forma perfecte les etapes i per tant la millora no serà igual al número d'etapes.

Per exemple :

Suposem que tenim una instrucció càrrega d'una dada i per tant l'execució de la mateixa comporta les següents operacions :

-Busqueda instrucció.
-Lectura del registre.
-Accés a l' ALU.
-Busqueda de dada.
-Escritura en el registre.

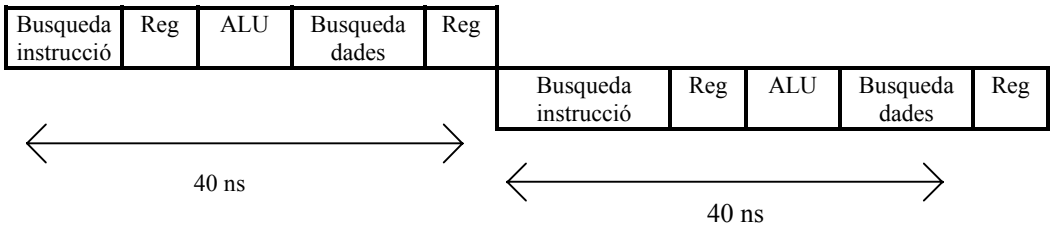
I que els temps necessaris són els següents :

-Accés a Unitat de Memòria .... : 10 ns
-ALU i sumadors .....: 10 ns
-Arxiu de registres (L/E).....: 5 ns

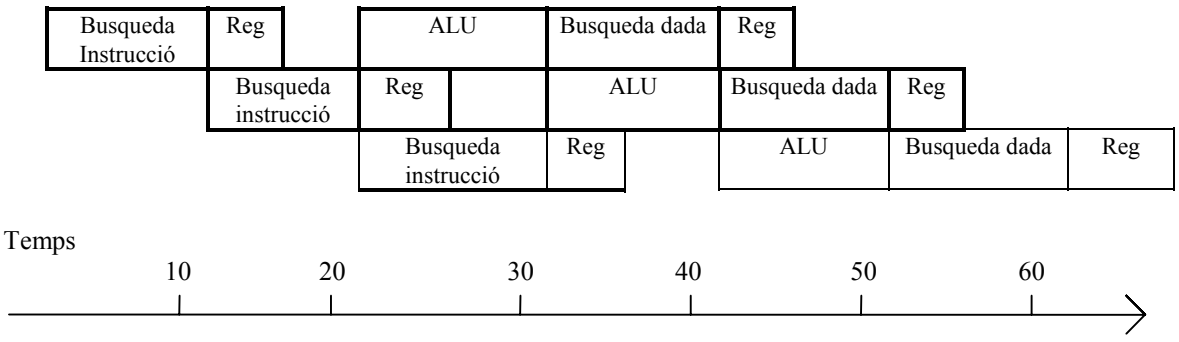
Per tant el temps d'execució serà :

Tipus instrucció	Memòria instruccions	Lectura de registre	Operació ALU	Memòria de dades	Escriptura registre	Temps total
Cargar paraula	10ns	5 ns	10 ns	10 ns	5 ns	40 ns

El diagrama de temps obtingut sense segmentació seria :



Si fessim una segmentació podria resultar :



Com es pot observar es tracta d'una segmentació en 5 etapes la qual cosa ens permetrà poder executar 5 instruccions alhora. En el nostre cas es pot observar que s'executen tres instruccions alhora. El nostre cicle de rellotge s'haurà d'adaptar a l'instrucció més lenta en aquest cas és la de 10ns, i per tant el cicle de rellotge haurà de ser de 10 ns. També cal observar que no totes les instruccions utilitzen els 10 segons. Hi han algunes que només necessiten 5 segons el reste de temps es troben en espera.

També es bo observar que si executessim 1 sola instrucció el temps d'execució amb segmentació seria 5ns més que no segmentada, però a partir d'aquí la productivitat de la segmentada millora el temps respecte a la no segmentada. La productivitat de la segmentada quan supera les 5 instruccions s'aproxima a 4 vegades la producció de la no segmentada. Per exemple :

100 instruccions no segmentada = 100 x 40ns = 4000 ns
100 instruccions segmentada= 97x10 ns + 70ns = 1040 ns

La relació és de 4000/1040 = 3,846

i quan més nombre d'instruccions més s'aproxima a 4 vegades.

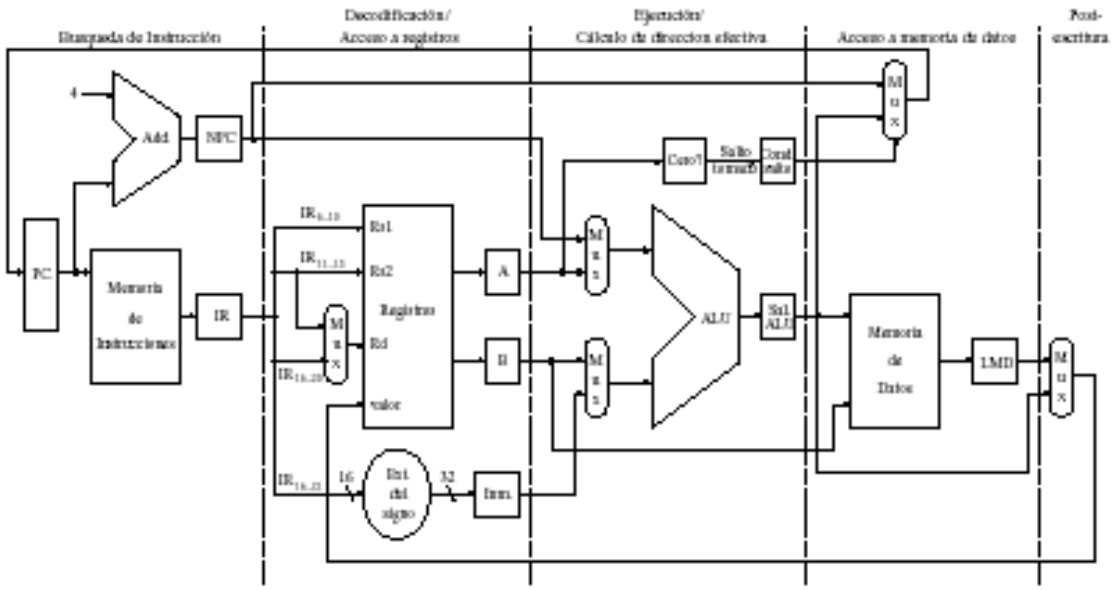
La segmentació ens permet explotar el paral.lelisme entre instruccions en un fluxe seqüencial de les mateixes.

En la segmentació s'ha de tenir en compte que a un mateix recurs només es pot accedir per part d'un segment.

2. El camí de les dades en la segmentació

Una de les coses més importants que s'han de tenir en compte a l'hora d'estudiar la segmentació és el camí que seguiran les dades. Això és degut a que la màquina sobre la que realitzem la segmentació té un nombre de recursos determinats i aquests només poden ésser utilitzats en un mateix moment per un segment.

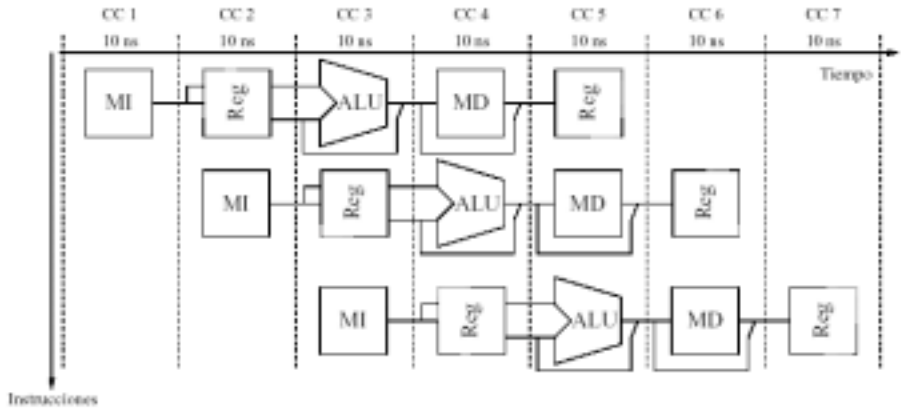
Per veure el flux de les dades en aquest apartat utilitzarem un camí de dades molt sencill per una estructura monocicle que serà el següent :



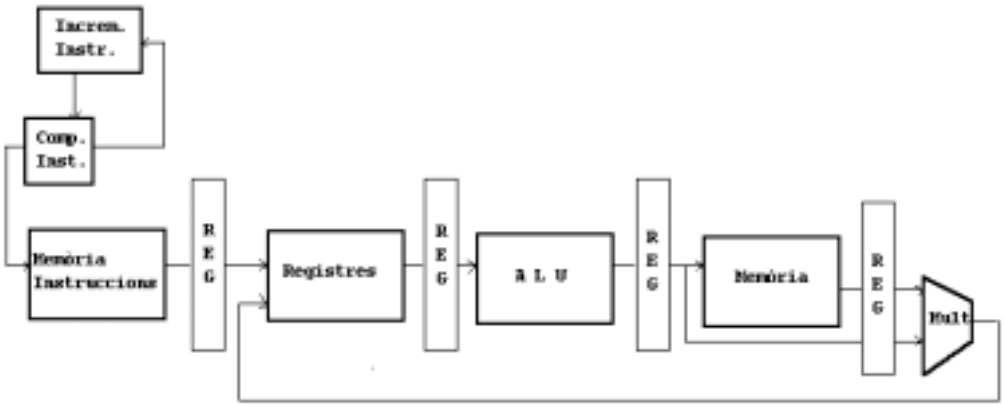
Com es pot observar l'informació passa d'esquerra a dreta en cada pas. El camí de les dades com es pot observar estaria dividit en quatre parts :

- Busqueda instrucció a on s'accediria al comptador d'instruccions per obtenir l'adreça que hem d'accedir de la memòria d'instruccions i posteriormet l'increment del comptador d'instruccions.
- Decodificació de l'instrucció i busqueda dels registres.
- Execució del càlcul.
- Accés a memòria.
- Escriptura dels resultats en els registres (aquests és l'únic pas a on el fluxe de les dades anirà de dreta a esquerra)

En aquest cas al tractar-se d'un fluxe de dades monocicle no hi ha perill de l'utilització del mateix recurs per part de la instrucció. Quan volem tractar d'una segmentació la cosa es complica i per poder observar el camí hauriem d'utilitzar una gràfica on representem en cada moment el registre utilitzat :

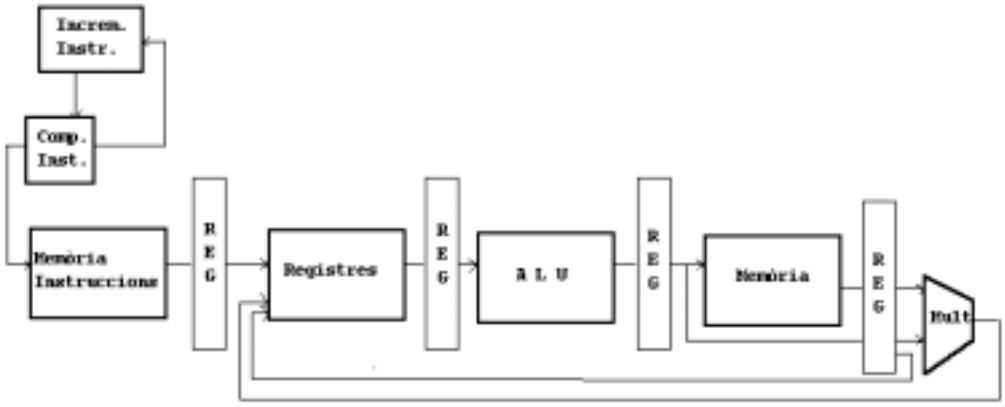


Amb aquesta gràfica podem observar en tot moment si existeixen conflictes. El primer problema que s'observa és el fet de que les dades com són el codi de l'instrucció i el valor dels registres es necessiten en tota l'execució de l'instrucció i donat a que es compartit entre tots els segments al passar d'una fase a un altre es perdrien i per tant s'haurien d'introduir registres entre cada fase que ens permetin desplaçar els valor de fase a fase.



Ara considerem les dues operacions més sencilles que es poden realitzar : la carga i l'emmagatzemament de les dades de un posició de memòria a un registre i a l'inrevés.

Per realitzar la **càrrega de les dades** de memòria a un registre s'haurà de realitzar una modificació en l'estructura que tenim fins introduïnt un camí que permeti passar dels darrers registres de segmentació l'adreça del registre de la CPU a on volem realitzar l'escriptura. L'estructura resultant serà la següent :



1) Busqueda d'instrucció .

   S'agafarà el contingut del comptador d'instruccions i es buscarà l'instrucció corresponent a la memòria d'instruccions. Tant el comptador d'instrucció com la instrucció es guardaran en el registres de segmentació per poder ser utilitzats, si així s'escau, en les següents etapes de la segmentació.
   Finalment s'incrementarà el contingut de comptador d'instruccions.

2) Decodificació d'instrucció i càlcul de l'adreça efectiva.

   Es decodificarà l'instrucció , es calcula en la CPU l'adreça de memòria que s'ha d'accedir i es trapassarà en el registres de segmentació per a la fase següent tota l'informació del comptador l'adreça de memòria i els valors dels registres de la CPU, així com el registre sobre el que s'ha de fer l'escriptura posteriorment. El fet de traspassar tots els registres a la fase següent no seria necessari donat que ja coneixem la instrucció però a que el cos es baix així ho farem.

3) Accés a memòria.

   Es busca en l'adreça de memòria el valor dessitjat i es guarda juntament amb l'adreça del registre de la CPU a on esvol guardar en el registre de segmentació per poder-ho utilitzar en la fase següent.

4) Escriptura en el registres de la CPU.

   Finalment haurem de passar el contingut del registre de segmentació de la lectura de la memòria així com l'adreça del registre per poder fer l'escriptura del valor en el registre corresponent.

Per realitzar **l'escriptura de les dades** es seguirant el següents passos :

1) Busqueda d'instrucció .

   Es farà el mateix procés que en la càrrega de dades és a dir s'agafarà el contingut del comptador d'instruccions i es buscarà l'instrucció corresponent a la memòria

d'instruccions. Tant el comptador d'instrucció com la instrucció es guardaran en el registres de segmentació per poder ser utilitzats, si així s'escau, en les següents etapes de la segmentació.
Finalment s'incrementarà el contingut de comptador d'instruccions.

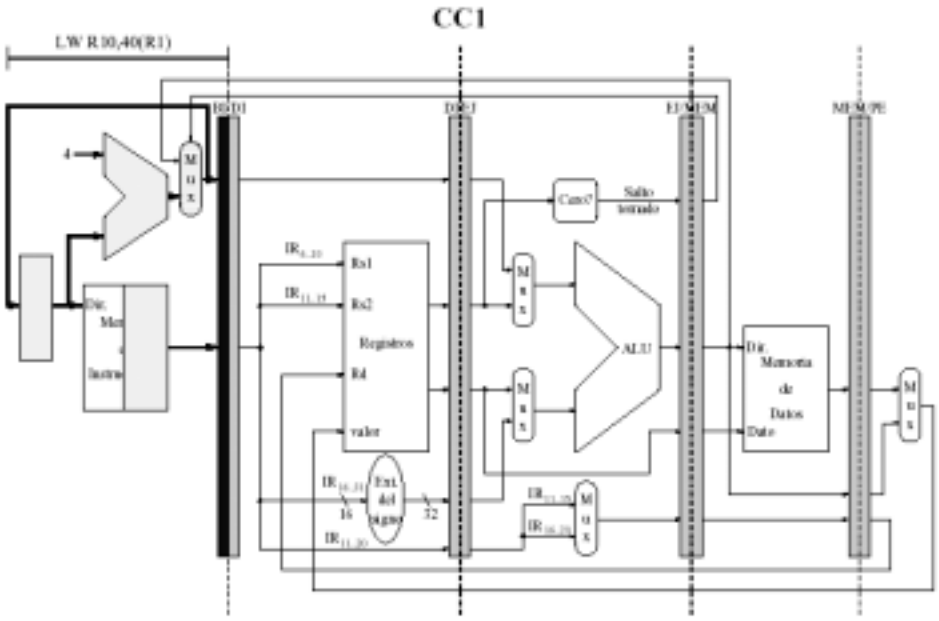
2) Decodificació d'instrucció i càlcul de l'adreça efectiva.
   Es decodificarà l'instrucció , es calcula en la CPU l'adreça de memòria que s'ha d'accedir i es trapassarà en el registres de segmentació per a la fase següent tota l'informació del comptador l'adreça de memòria i els valors dels registres de la CPU.

3) Accés a memòria.
   Es guarda en l'adreça de memòria el valor dessitjat prèviament calculat en l'anterior fase.

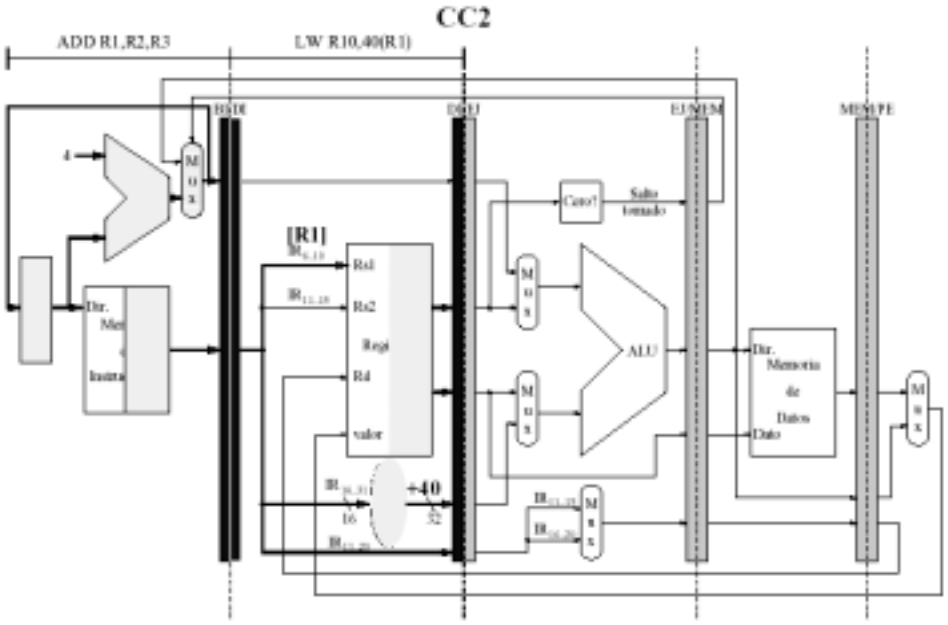
4) Escriptura en el registres de la CPU.
   En aquesta fase no es realitzarà cap operació donat que no s'ha de guardar cap valor en els registres de la CPU.

Per exemple podem veure com s'executarien les següents instruccions:

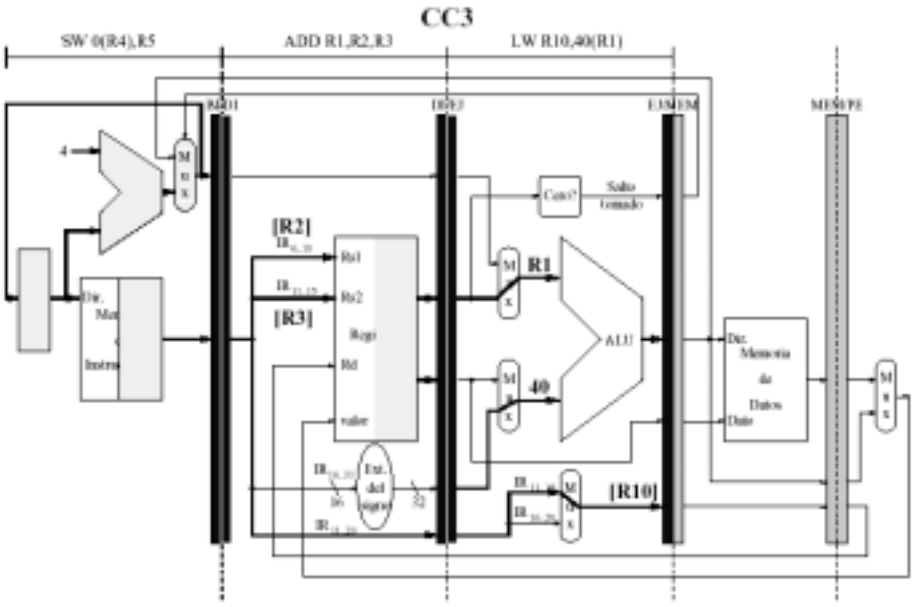
LW R10,40(R1)
ADD R1,R2,R3
SW 0(R4),R5
SUB R2,R2,R6
BEQ R1,dest
SW 40(R1),R10



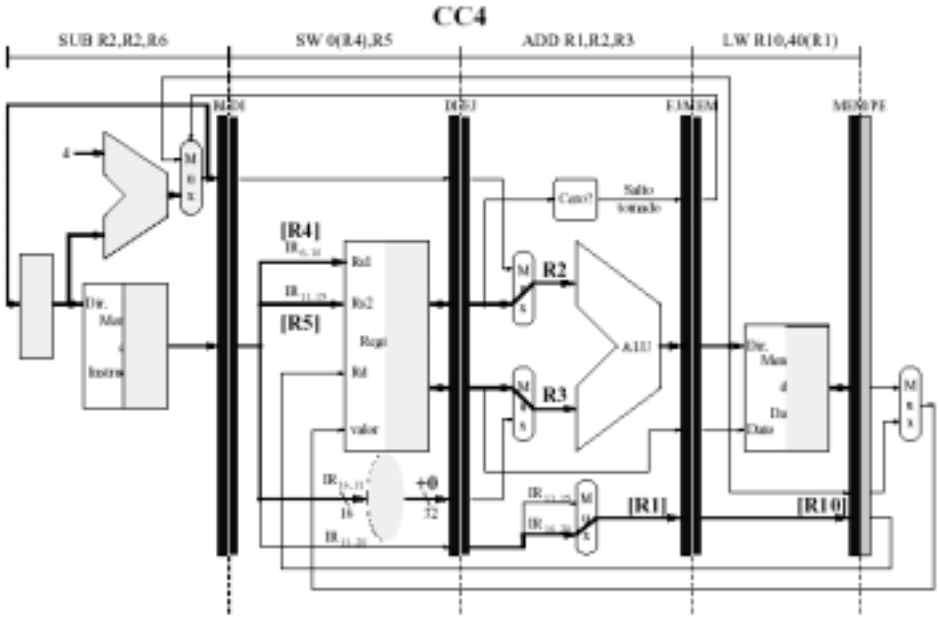
En el primer cicle com es pot observar entre la primera instrucció en el registre d'instruccions i automàticament s'autoincrementa el comptadors d'instruccions i guarda els registres corresponents en la primera banda de registres.



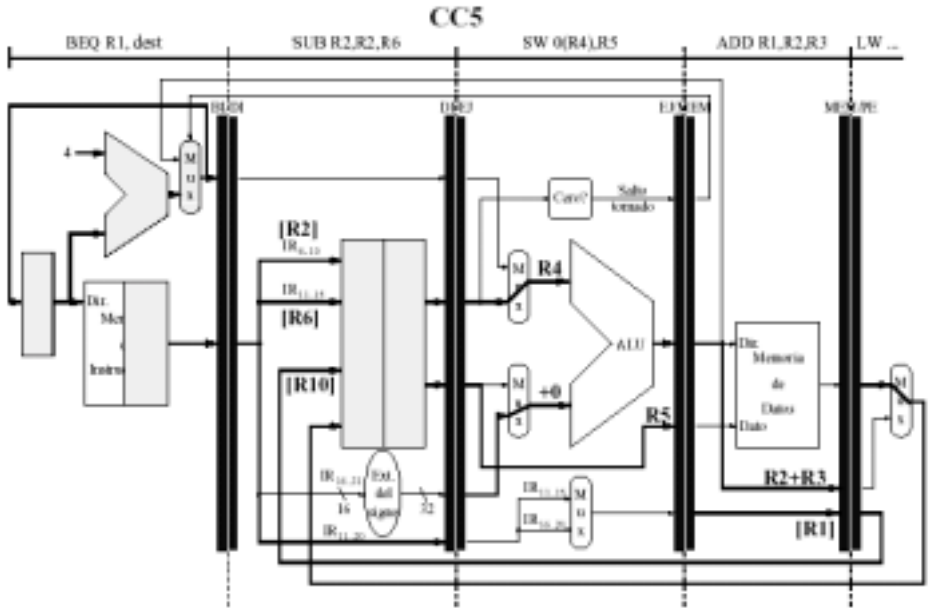
En el segon cicle de rellotge entre la segona instrucció i la primera realitza la lectura dels registres que necessita i els guarda en la segona banda de registres.



En el tercer cicle com es pot observar es realitza el càlcul de l'adreça de memòria que requereix la primera instrucció, mentre que la instrucció ADD recull el valors del registres R2,R3. La tercera es decodifica.

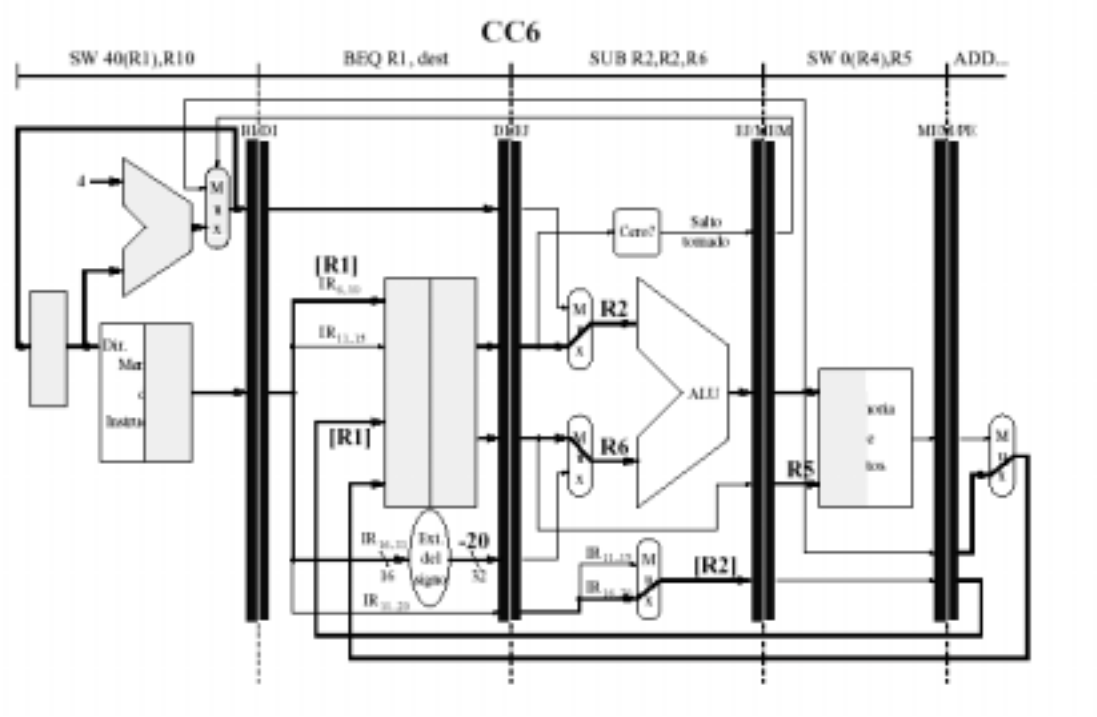


En el quart cicle de rellotge la primera instrucció realitza l'accés a memòria, la segona realitza la suma, la tercera agafa el valors dels registres que necessita i la quarta es decodifica.



En el cinquè cicle de rellotge la primera instrucció (LW) guarda el contingut llegit a memòria en el registre corresponent. La instrucció segona es troba en l'etapa de memòria però no realitza cap accés donat que no ho necessita. La tercera instrucció realitza el càlcul de l'adreça que necessita per

a fer l'escriptura a memòria. La instrucció quarta recull el valors del registres i entra la cinquena instrucció.



Com es pot observar les instruccions anirien entrant cicle per cicle i que el fet de que una instrucció es trobi en una etapa no implica que tingui de realitzar la tasca que es fa en l'etapa, és a dir, que si no ha d'escriure cap valor en un registre no ho realitzarà i el temps que dura la etapa el passarà sense realitzar cap tasca.

3. Formes de representació gràfiques de la segmentació.

La representació gràfica de la segmentació ajuda a la comprensió de la mateixa, donat que hi han moltes instruccions que es executen simultàniament en cada cicle de rellotge. Fonamentalment podiem dividir les representacions en 2 tipus :

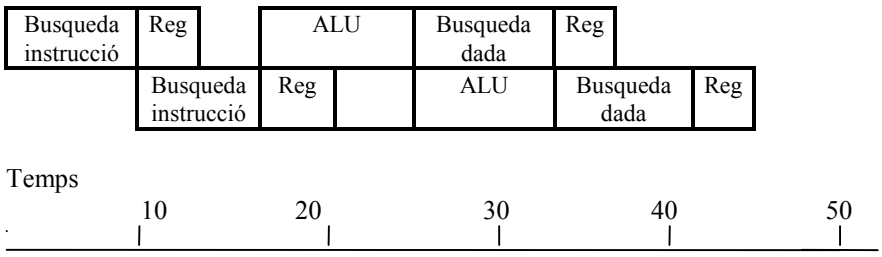
- Representacions de multiples cicles de rellotge.

  Ens permet observar en una visió general de les situacions que es presenten en la segmentació.
  En aquest tipus de diagrames el temps avança d'esquerra a dreta mentres que les instruccions ho fan d'a baix cap a munt.

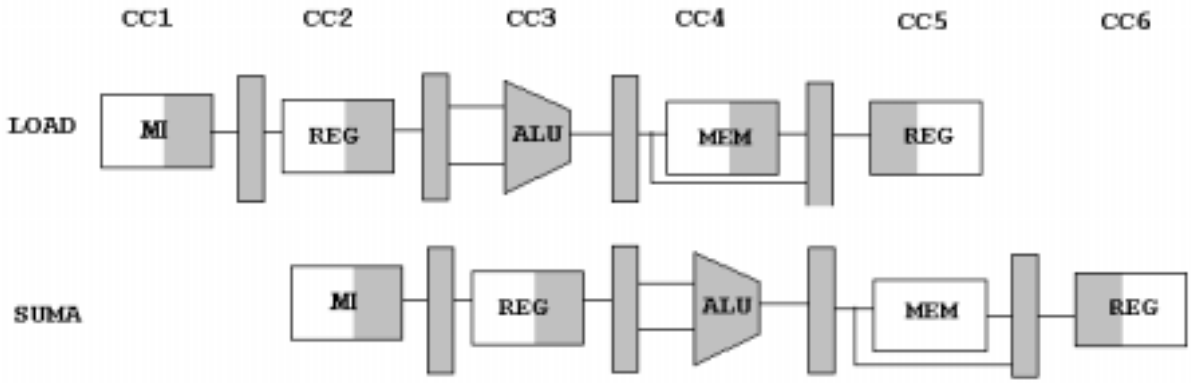
Un exemple seria el següent :

Si considerem les següents instruccions :

Carregar A ,1000 (on A és un registre i $1000 és una posició de memòria)
Sumar A,B,C (sumar els registres B i C i guardar el resultat a A)



D'una forma més acurada es pot representar com :



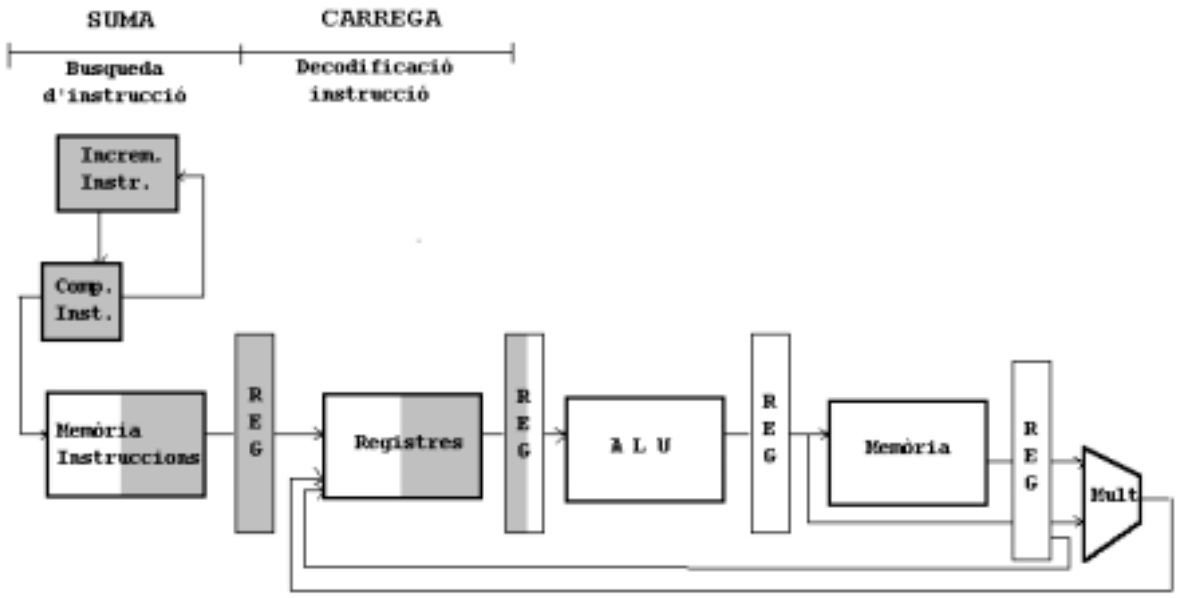
A on podem observar que el cas de memòries i registres es sobreja la meitat esquerra si és una escriptura i la dreta si és una lectura. En aquest exemple es pot observar com el cas de la memòria en la suma no intervé .

- Representació d'un cicle de rellotge.

Aquesta representació ens permet veure un estat complet del camí de dades en un cicle de rellotge. Cal tenir en compte que per tenir tot el cicle complet que realitza una instrucció serà necessari tants gràfics com cicles composin l'execució de l'instrucció.

Cal esmentar que les dues versions són equivalents i que es pot passar d'una versió a un altre. Per passar de la versió monocicle a la multicicle només cal retallar en el cicle de temps dessitjat la multicicle. En el cas de passar d'una monocicle a una multicicle no és tant fàcil donat a que hem de obtenir la representació de cada cicle i després realitzar una solapament de totes les representacions.

Com es pot observar en el següent exemple el sentit d'execució de les instruccions seran d'esquerra a dreta. A part es ressaltaran en el cas dels registres i la memòria la part dreta en el cas d'una lectura i l'esquerra en el cas d'una escriptura.



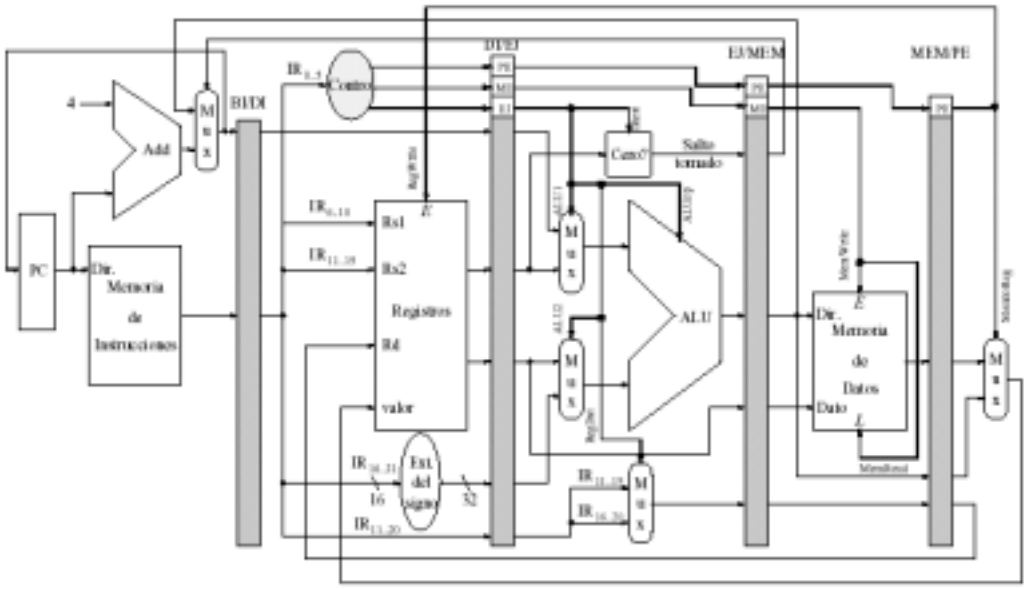
4. El control en la segmentació.

Un dels aspectes importants en l'execució d'un sistema segmentat és el fet del control dels recursos que s'han d'executar. Així trobem que en diferents estats d'execució d´un segment i segons el tipus d'instrucció que s'executa , un recurs podrà ésser o no utilitzat i sobre ell es podran realitzar diferent tipus d'operacions. Per exemple hi han instruccions que en el mateix segment necessitarà un accés a memòria i un altra no i dintre de les que realitzen una operació a memòria alguns faran lectures i altres faran escriptures.

Per controlar aquest aspecte haurem de crear una estructura de control. Fonamentalment utilitzarà senyals de control que aniran directament sobre els recursos que ho requereixen i a part es necessitarà una sèrie d'elements que s'encarreguin d'enviar aquestes senyals quan així es requereixi.

Cal considerar que hi han recursos que no necessitaran aquestes senyals de control donat a que sempre s'executen de la mateixa forma independentment de la instrucció.

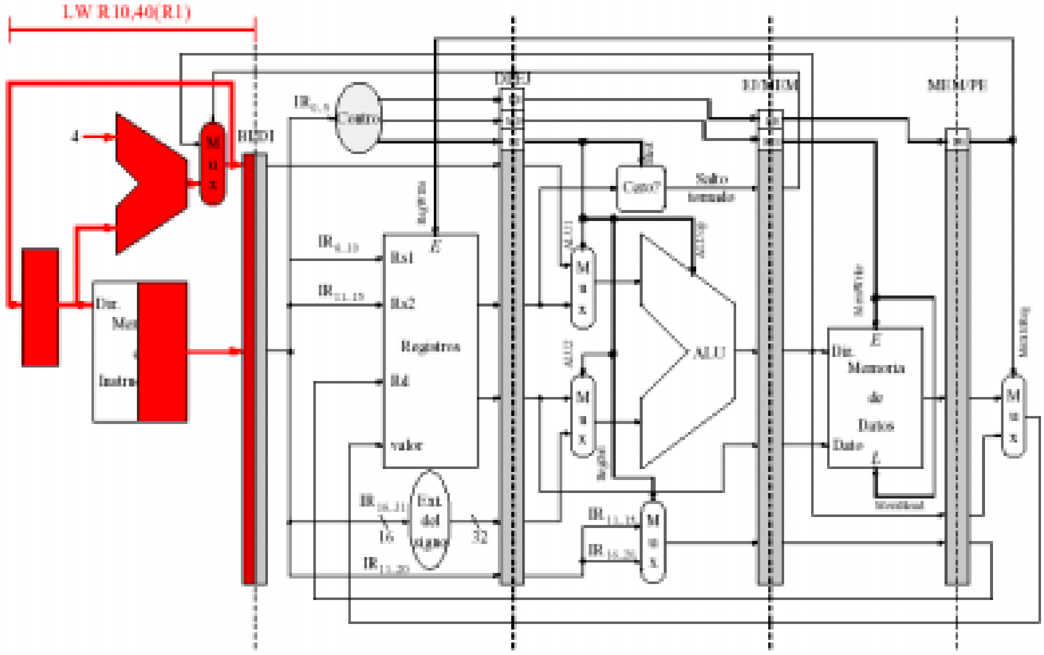
En el nostres cas podem observar que les senyals de control es requeriran fonamentalment en els registres, la memòria i en algunes parts de la CPU.



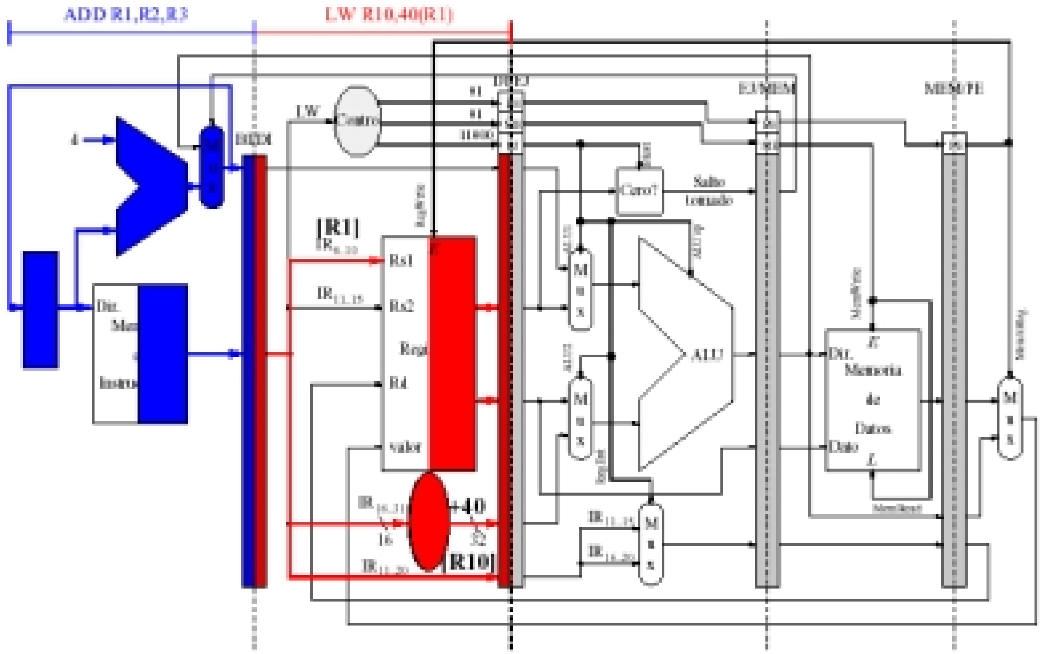
Com es pot observar els senyals de control es poden generar després de la fase de codificació de la instrucció donat que en les fases anteriors no es requereixen cap senyal de control. Ja que cada instrucció requerirà uns senyals diferents a altres l'estructura que els controlaria estaria lligat a la descodificació de la instrucció .Els senyals es passaran a través dels registres de segmentació de fase a fase de forma que cada fase utilitzarà aquells que requereixi.

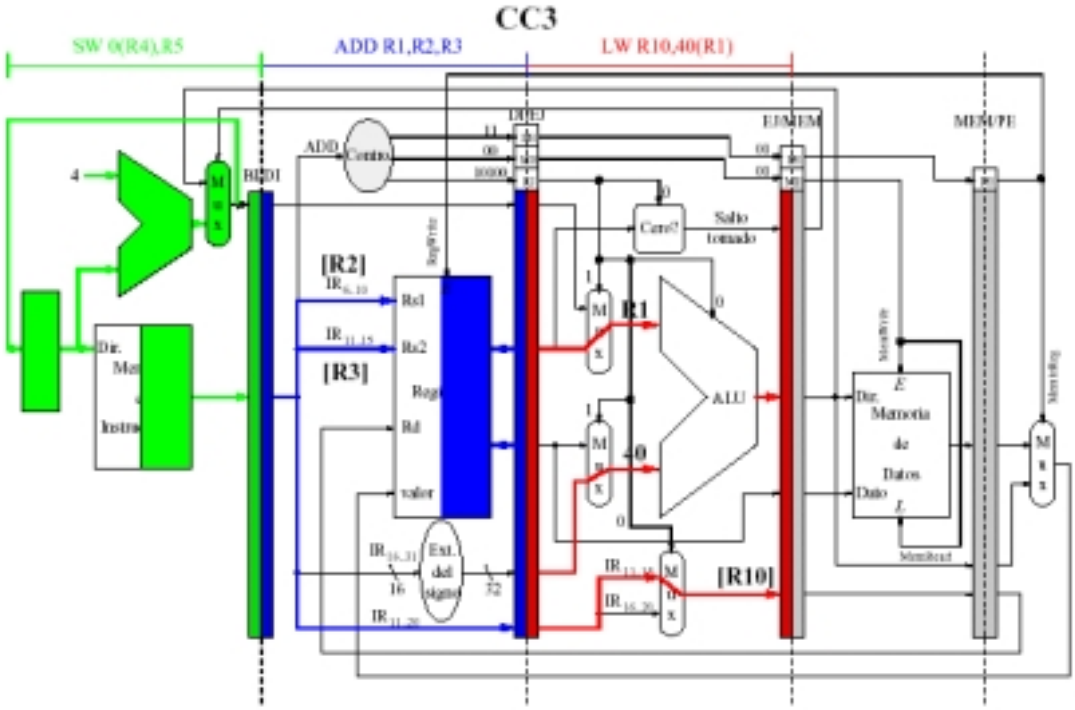
Si agaféssim l'exemple anterior veuríem com va evolucionant els senyals de control:

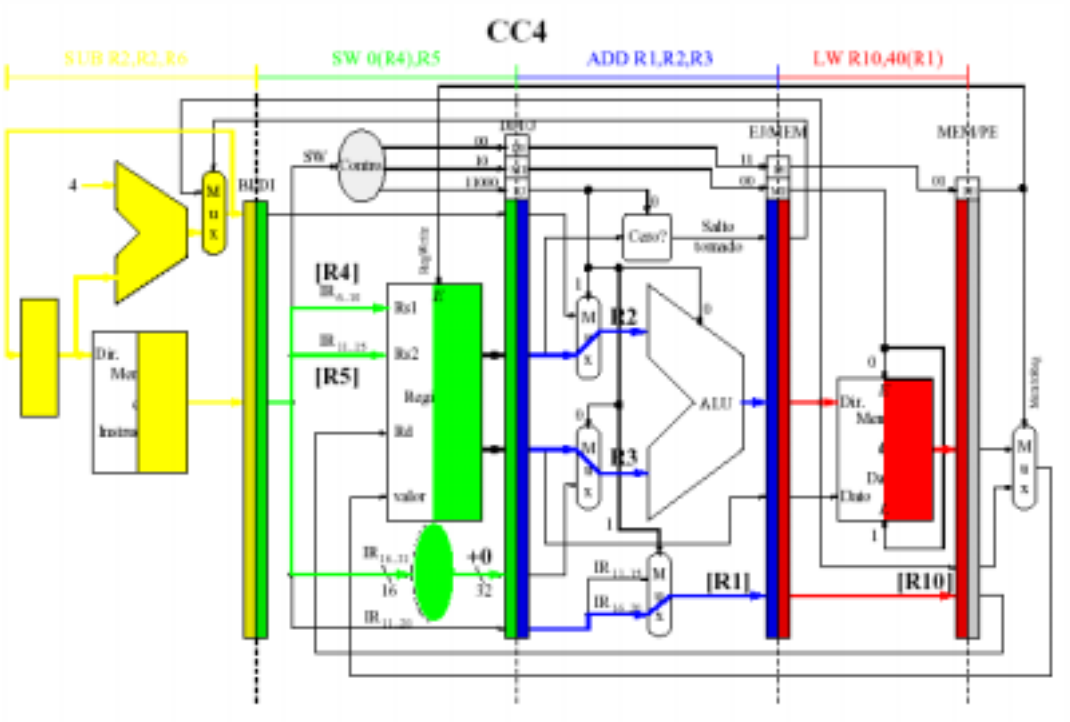
CC1

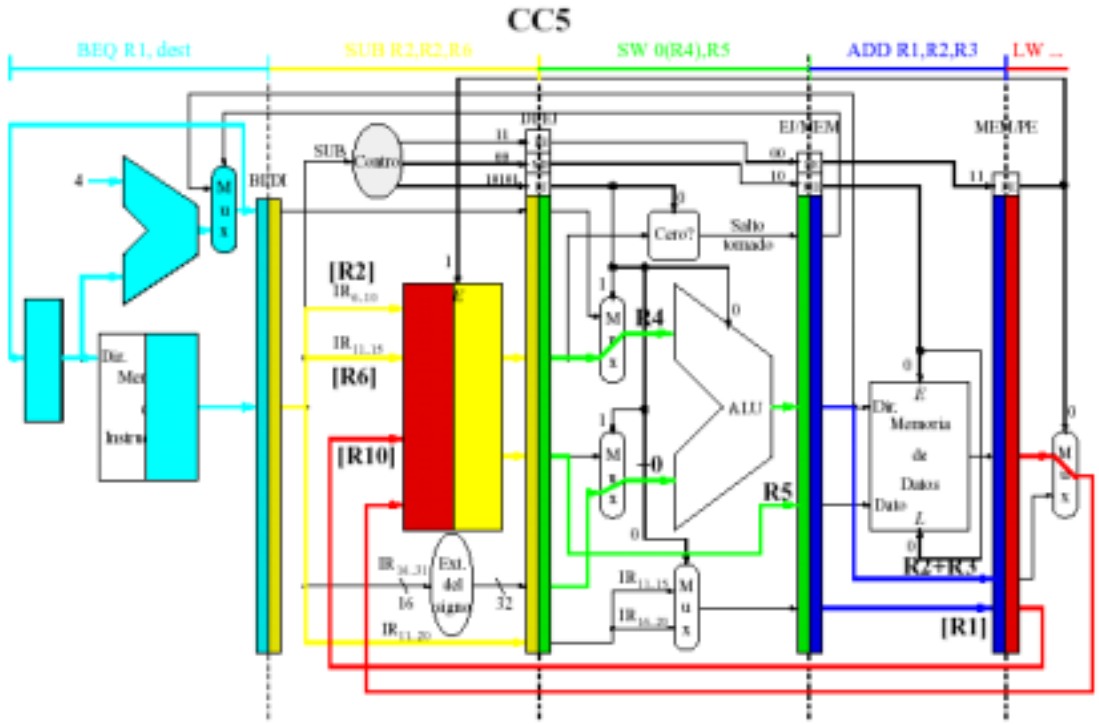

LW R10,40(R1)
Control
ALU
Memoria de Datos
Registros

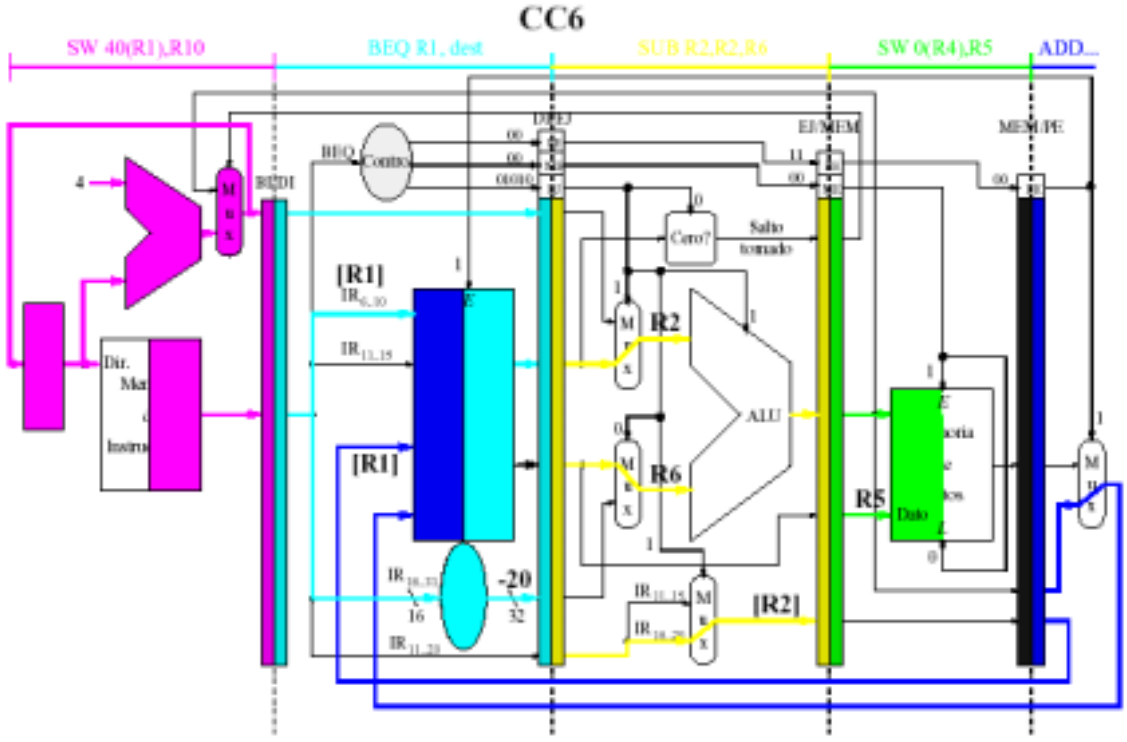
CC2


ADD R1,R2,R3
LW R10,40(R1)
Control
[R1]
+40
[R10]
ALU
Memoria de Datos


CC3
SW 0(R4),R5
ADD R1,R2,R3
LW R10,40(R1)
[R2]
[R3]
[R1]
[R10]
ALU
+0


CC4
SUB R2,R2,R6
SW 0(R4),R5
ADD R1,R2,R3
LW R10,40(R1)
[R4]
[R5]
[R2]
[R3]
[R1]
[R10]
ALU
+0


CC5
BEQ R1, dest
SUB R2,R2,R6
SW 0(R4),R5
ADD R1,R2,R3
LW ...
DEC
EJ/MEM
MEM/PE
SUB
Contro
BIDI
Cero?
Salto tomado
[R2]
[R6]
[R10]
R4
R5
ALU
Dir. Memoria de Datos
Dato
R2+R3
[R1]
Ext. del signo
16
32
4


CC6
SW 40(R1),R10
BEQ R1, dest
SUB R2,R2,R6
SW 0(R4),R5
ADD...
DEC
EJ/MEM
MEM/PE
BEQ
Contro
BIDI
Cero?
Salto tomado
[R1]
[R1]
R2
R6
R5
ALU
Dato
[R2]
-20
16
32
4

5. Risc estructurals.

Un dels problemes que se'ns pot presentar és el fet dels conflictes que es poden produir en l'execució de les instruccions. Els recursos que disposem a nivell d'arquitectura ens limiten a l'hora d'executar les instruccions.

Per exemple pot donar-se el cas que dues instruccions vulguin accedir a la Alu en el mateix moment per fer una operació de coma flotant (això sol succeir força sovint en segmentacions multicicles). Un altre problema pot ser el fet de voler accedir a algún recurs i no hi hagi cap camí disponible perquè aquest està essent fet servir en aquest moment.

Per evitar aquest problema s'hauran de realitzar modificacions a nivells estructurals per adaptar-se al model que estem fent servir. Per exemple :

- Introduir dos camins per a accedir a memòria: un per a obtenir la instrucció que necessitem i un altre per a accedir a les dades i d'aquesta forma evitar els conflictes que es produeixen quan es volen llegir una instrucció i a una dada.

- Permetre dos accessos de lectura a registres i un d'escriptura (en un mateix cicle). Amb això aconseguirem llegir dos operant alhora en un mateix cicle.

- Establir un sumador propi pel ***PC*** i així evitarem que s'utilitzi l'ALU per calcular el comptador de la següent instrucció a executar.

Cal tenir en compte que la magnitut de perdua de rendiment que es pot produir provocat per aquests conflictes és bastant considerable. Per exemple:

Si només disposessim d'un sol camí a memòria això ens fa perdre en cada colisió d'instruccions 1 cicle de rellotge. Si un estudi de la carga del programa ens indica que el 30% d'instruccions són ***LOAD*** aplicant la llei d'Amdhal observariem que:

Si suposem que la CPI^{ideal} de la segmentació fos de 1.2

$$\text{Millora}_{seg} = \frac{Cpi^{ideal} \times n \text{ etapes}}{Cpi^{ideal} + \text{cicles perduts}}$$

$$\text{Perdua de rendiment} = \frac{\text{Millora}^{\text{ideal}}}{\text{Millora}^{\text{real}}} = \frac{\frac{1.2 \times n \text{ etapes}}{1.2}}{\frac{1.2 \times n \text{ etapes}}{1.2 + 0.3 \times 1}} = 1.25$$

Com es pot observar en aquest exemple la perdua del rendiment és d'un 25% respecte al rendiment que es considera ideal.

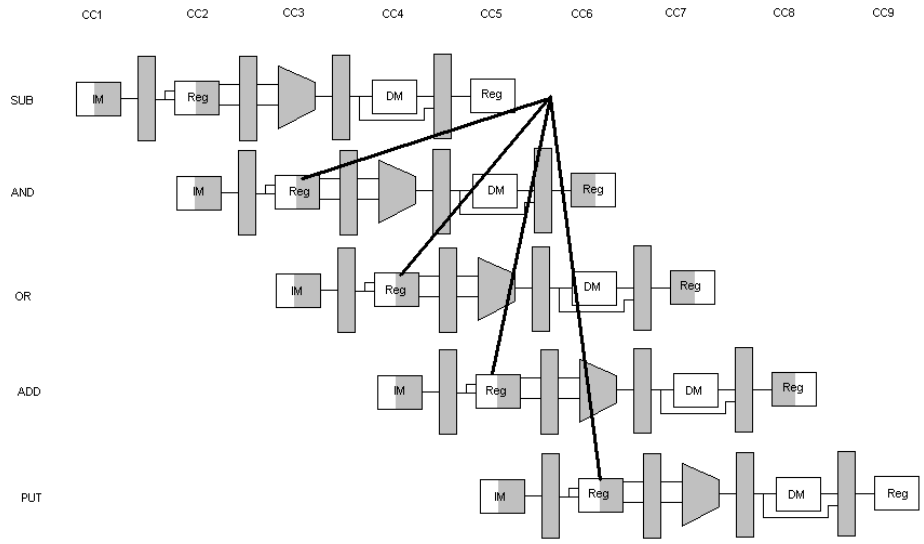
6. Risc en la dependència de dades.

Un dels problemes més grans que ens podem trobar en la segmentació és la dependència de les dades entre instruccions. Aquest es produeixen quan degut a la segmentació l'ordre de lectura als operants i el d'escripura dels resultats es modifica. Per exemple si tenim la següent seqüència:

```
SUB r2,r1,r3
AND r4,r2,r5
OR r5,r7,r2
ADD r8,r2,r2
MOVE r10,r2
```

veiem que la primera instrucció realitza una reste entre els registres r1 i el r3 i el resultat el possa en r2. En la següent instrucció es realitza una operació AND entre el registre r2 i el r5 i el guarda en el r4.

El problema principal es troba en el fet que la segona operació necessita el valor del r2 de la primera instrucció i aquest no el té fins el cicle 5, i quan va a fer la lectura la segona instrucció és en el cicle 2, per tant el valor no serà correcte. En aquest cas hi ha una dependència entre les dades de la primera i la segona instrucció.



El mateix ens passa amb les instruccions 3, 4 i 5. Per solucionar aquest problema s'hauria d'implementar alguna tècnica que eviti l'execució de l'instrucció fins que el valor del registre fos el correcte.

Quan s'està executant una instrucció les dades poden treballar de dues formes:

- Com operant d'una instrucció.
- Com a resultat d'una instrucció.

Per exemple si tenim la següent instrucció *"A= B+C"* tenim que *"B"* i *"C"* actuen com a operants i *"A"* actua com a resultat.

A partir d'ara farem servir la següent nomenclatura: "R= Resultat", "O=Operant" , "i=instrucció en curs", "j=instruccions que li precedeixen".

Tenint en compte aquesta diferenciació de les dades podem extreure la següent classificació de dependència de dades:

11. $O(i) \cap O(j) \neq \emptyset$ que entre els operants de succesives instruccions hi hagi alguna coincidència.
    Per exemple:

    A=B+C
    D=B*E

    Aquest cas no es considera perquè no produeix cap error donat que les dades no queden modificades ja que només es llegeix i no s'escriu.

11. $O(i) \cap R(j) \neq \emptyset$ quan algun dels operants de la instrucció en curs serà modificat posteriorment per una altra instrucció. Per exemple:

A=B+C
B=D+E

La dependència es produiria si la instrucció en curs llegeix l'operant després que la instrucció que li precedeix escrivís el seu valor. Aquest error també se'l coneix com WAR (escriure abans de llegir). En el cas de la segmentació aquest error no es produeix donat que quan es comença l'execució d'una instrucció es llegeixen tots els operants de la mateixa i es copien en els registre de desacoplament que hi ha en les unitats funcionals.

11. $R(i) \cap O(j) \neq \varnothing$ quan el resultat de la instrucció en curs es necessita com a operant d'una instrucció que li precedeix. Per exemple:

A=B+C
B=A*D

Aquest error es pot produir fàcilment en la segmentació donat que abans que una instrucció obté el resultat poden haver començat l'execució vàries instruccions. L'única solució disponible en aquest cas és parar l'execució de la instrucció que provoca la dependència juntament amb les que li precedeixen fins que l'operant no estigui disponible. També es coneix aquesta dependència com *RAW* (llegir abans d'escriure). En el cas de la segmentació aquest problema es sol solucionar amb la parada de l'execució de la instrucció en curs.

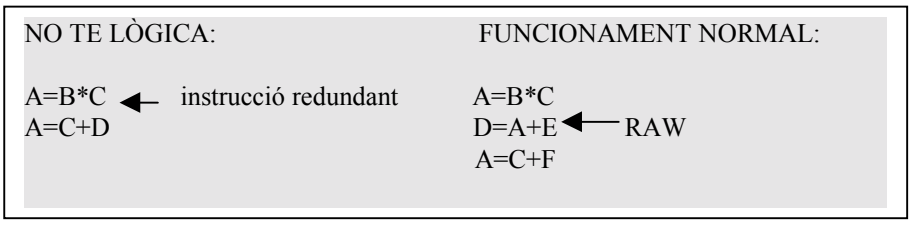
11. $R(i) \cap R(j) \neq \varnothing$ quan el resultat de la variable de la instrucció en curs s'escriu amb posterioritat al resultat de la mateixa variable que es produeix en una instrucció que li precedeix. Per exemple:

A=B*C
A=B+C

En el cas de la segmentació monocicle no té raó d'ésser donat que totes les instruccions acaben en ordre, però en el cas del multicicle es podria si la instrucció en curs té una durada superior en cicles que la que produeix el conflicte (per exemple una operació de coma flotant i una operació sencera).

Cal tenir en compte que aquest tipus d'error en el cas de la segmentació no s'haurien de produir si el compilador fos bo. Això es deu a que no té lògica que després d'una escriptura en un registre com a mínim es farà un lectura del mateix abans de fer una escriptura en ell. Si això no es produeix es podria dir que la primera instrucció és redundant i no necessària donat a que el seu resultat no es fa servir per a res. En el cas de que després de fer l'escriptura es fes una lectura del registre el problema WAW mai es

produiria perquè abans es produiria el RAW cosa que bloquejaria l'execució de les instruccions. Per exemple:



Com veurem després hi han diferents tècniques implementades sobre el compilador que ens faciliten la dependència de les dades. Una d'ella correspondria al fet d'introduir instruccions nop (no operació) entre instruccions per sincronitzar la seva execució :

```
SUB r2,r1,r3
NOP
NOP
NOP
AND r4,r2,r5
OR r5,r7,r2
ADD r8,r2,r2
MOVE r10,r2
```

Amb això conseguirem que quan fem un accés al registre el valor sigui el correcte. Altres compiladors introdueixen instruccions independents entre mitges per optimitzar l'execució de les instruccions.

7. Mecanismes de control de dependència de dades.

Com hem parlat anteriorment un dels mecanismes de control per solucionar el problema de dependència de dades seria la detecció del problema abans de que aquest es produeixi. Per fer això haurem d'estudiar els possibles estats en que es poden produir els conflictes esmentats.

Tots els casos de dependència de dades es troben quan una instrucció requereix gravar el contingut d'una operació en un registre i les instruccions següents volen operar amb el contingut d'aquest registre abans de que aquest sigui escrit. Per tant els conflictes es poden produir en qualsevol dels diferents estats de segmentació que es pot trobar la instrucció d'escriptura és a dir:

Accés a registre 1 i una altre instrucció de escriptura del registre 1 es troba en:

- Accés a ALU.
- Accés a memòria.
- Escriptura de registre.

O Accés a registre 2 i una altre instrucció de escriptura del registre 2 es troba en :

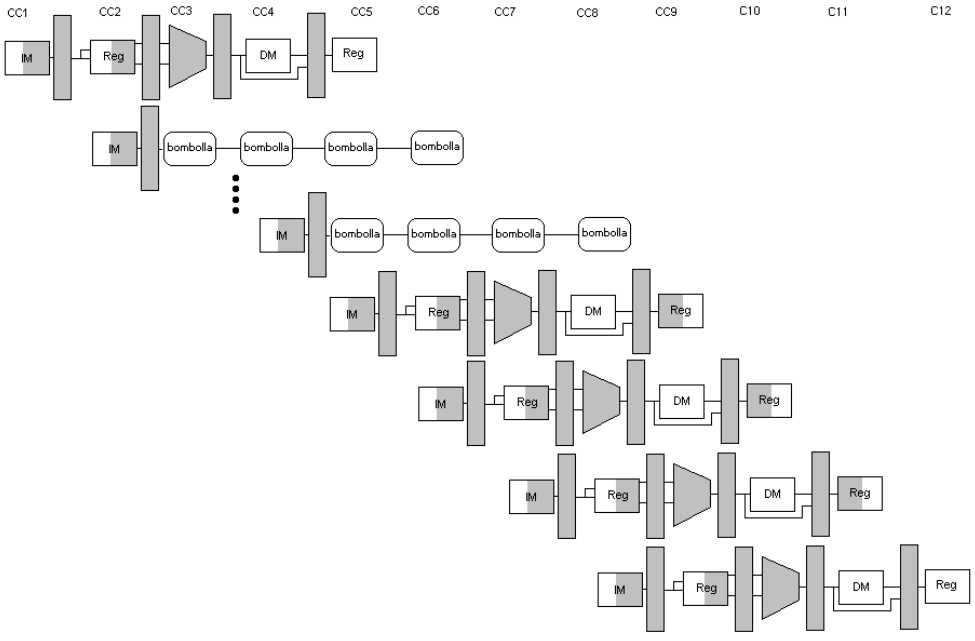
- Accés a ALU.
- Accés a memòria.
- Escriptura de registre.

Una vegada marcat aquests possibles estats en que es poden donar un conflicte cal establir un mecanisme per parar l'execució de la instrucció fins que el registre en qüestió estigui disponible.

Per fer això haurem de establir un dispositiu que permeti avançar la busqueda de la següent instrucció per evitar modificar el valor de la instrucció llegida fins que aquesta sigui operant. Això es pot fer no permeten avançar el comptador d'instruccions fent que cada cop vagi llegint la mateixa instrucció.

A part s'haurà de implementar un dispositiu que faci que els continguts dels registres de segmentació (adreces de memòria, descodificació d'instruccions, senyals de control, …) siguin inoperant en les fases en que la instrucció sigui inoperant, per evitar que es realitzin operacions incorrectes. Això es podria fer posant a zero els valors.

Com es pot observar en aquesta gràfica de l'execució resultant hi hauran una sèrie de cicles on no es realitzarà cap operació en espera d'obtenir el valor del registre desitjat:



7.1.Reducció de la dependència mitjançant anticipació "Curtcircuïts"

Amb el mètode que hem utilitzat fins ara per resoldre la dependència de les dades (introducció d'esperes) produïa una baixada del rendiment del procés donat a la pèrdua dels cicles en espera que el valor de l'operació fos gravat en el registre.

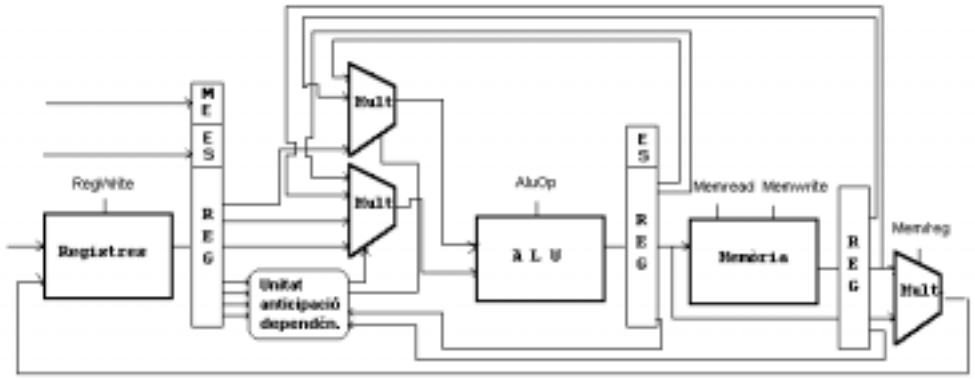
Per obtenir un millor rendiment utilitzarem la tècnica de l'anticipació que consisteix en no esperar a que el resultat es trobi gravat en el registre , sinó que mentre es produeixen les operacions es puguin passar els valors resultant cap els registres de segmentació de la fase d'execució i cap els registres de segmentació de la fase de instrucció si es produeix una dependència de dades. El mateix haurem d'establir per la fase de memòria donat que també es pot establir una dependència entre una instrucció que es trobi en la fase de memòria i una altre que es trobi en la fase d'instrucció.

Per poder implementar aquest mètode haurem de realitzar unes modificacions en la unitat de control per adaptar-la. En l'entrada de les dades en el registres de segmentació de la fase d'instrucció s'haurà d'introduir uns multiplexors que permetin escollir entre les dades provinents dels registres de la CPU , les dades provinents dels registres de segmentació de la fase d'execució i les dels registres de segmentació de la fase de memòria. Quan es produeixi una dependència de dades el multiplexor permetrà l'entrada dels valors provinents dels registres provinents de la fase d'execució o de memòria segons sigui la dependència.

S'ha de tenir en compte que els cas de produir-se una dependència de dades d'una instrucció amb altres instruccions que es troben executant-se, sempre tindrà més preferència la que es troba en una fase més propera a ell. Per exemple :

1) A= B + C
2) A= A + D
3) B= A + E sempre te més preferència el valor de A de la instrucció 2)

La unitat de control resultant podria ser la següent :

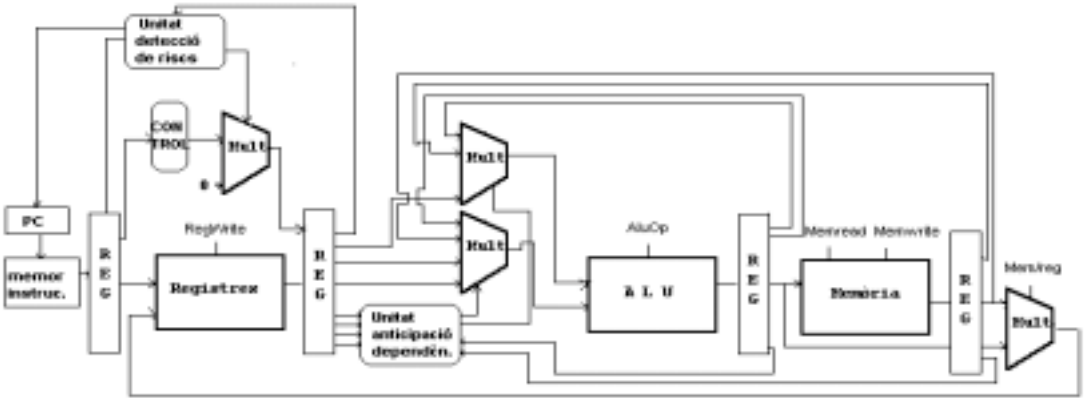


El problema que ens podem trobar amb l'anticipació es troba en les instruccions de carregar en un registre el valor corresponent a una adreça de memòria .

Això es deu a que l'informació del registre no es tindrà fins que estiguem en la fase de memòria. Si la instrucció que té una dependència de les dades es troba en la fase d'instrucció mentres que la instrucció de la que es té dependència no es troba en la fase de memòria s'haurà de

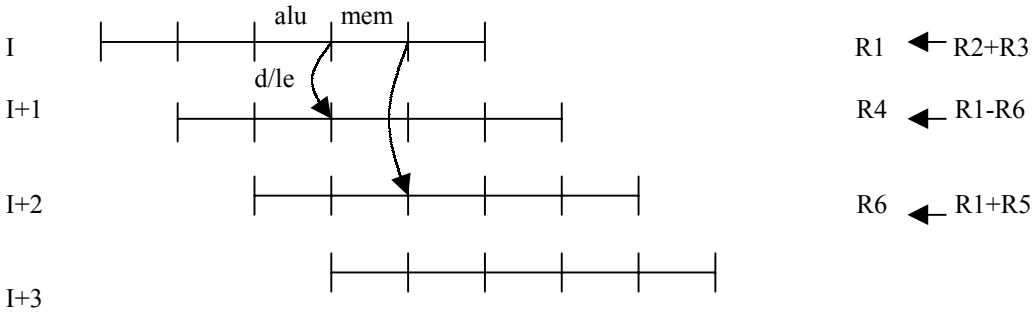
generar un cicle d'espera. D'aquesta forma evitarem que es produeixi una incongruència de les dades.

Per arreglar aquest problema hauriem d'establir una modificació en la unitat de control per prevenir aquest problema de forma que pari l'execució fins tenir el valor correcte.



Com es pot observar en la figura s'ha establert una unitat de detecció de risc que comprova si es dona la condició de ser una instrucci'o que dependència amb una instrucció de lectura de memòria. En aquest cas introdueix valor 0 en els senyals de control i no permet avançar l'instrucció parant així l'instrucció fins que el valor sigui llegit de memòria.

Un exemple de execució seria el següent:

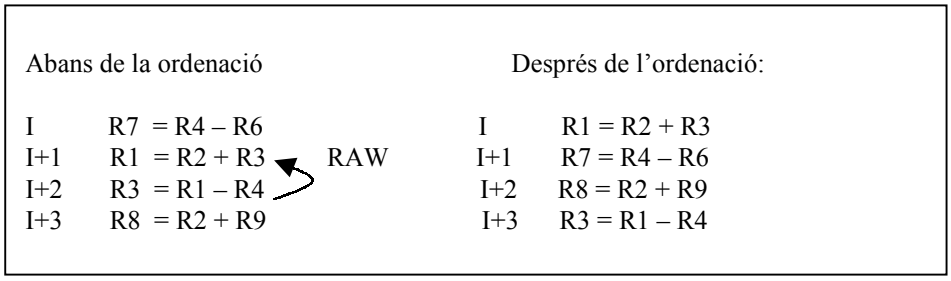


Es fa un curtcircuït de la sortida de l'Alu cap a l'entrada de registres (instruccions i , i+1) i un de la sortida de la memòria cap a l'entrada de registres (instruccions i, i+2).

7.2.Reducció de la dependència mitjançant reordenació d'instruccions.

Aquest mètode consisteix en que el compilador ha d'ésser el suficientment bo per detectar els possibles casos de dependència de dades i el solucioni mitjançant una reordenació de les instruccions. Per fer això hauria de fer un estudi previ de les instruccions que pot canviar de lloc sense que això modifiqui la lògica del programa. Si el compil.lador pot obtenir una combinació òptima aconseguirà eliminar la possibilitat d'introduir NOP que parin l'execució del programa i per tant optimitzant el rendiment.

Un exemple d'aquest mètode seria el següent:



8. Seqüenciament de les instruccions.

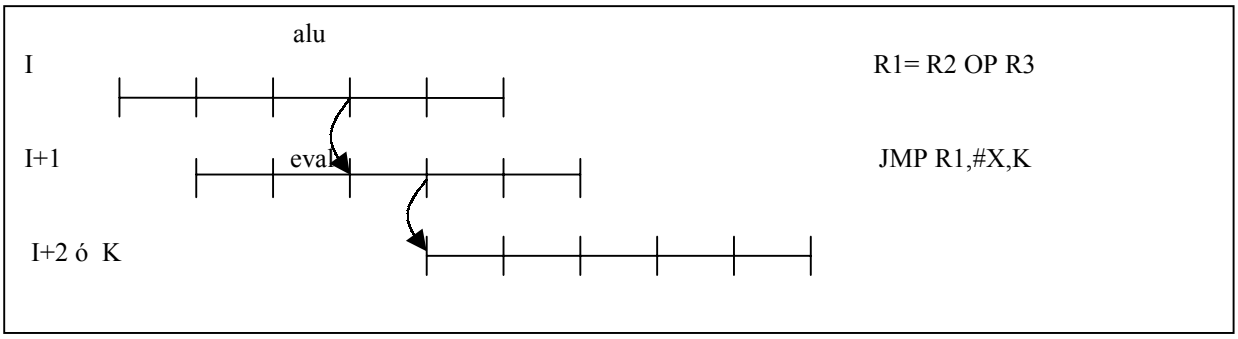
Quan parlem de la seqüenciament de les instruccions podem parlar de dos tipus :*l'implícit* que seria el que té tot programa per defecte és a dir que de cop que es llegeix una instrucció en el cicle de recerca el comptador d'instruccions augmenta en una unitat (PC=PC+1) , i *l'explícit* provocat per la modificació del comptador d'instruccions per part d'una instrucció de salt.

El fet de la introducció del seqüenciament explícit ens comportarà una sèrie de modificacions tant a nivell de maquinari com d'unitat de control.

El primer és la necessitat d'un sumador pel càlcul del valor del comptador nou (PC=PC+X) i a quin cicle de la instrucció es posa. Normalment es posa en el cicle de d/le (decodificació/ lectura d'operants).

Una altre element nou seria el fet d'evaluar la condició que produeix el salt. El fet d'obtenir més aviat o més tard aquesta evaluació ens pot perjudicar de forma important el rendiment del sistema.

Fins que el resultat de l'evaluació de la condició no es conegués s'hauria de Parar l'execució de les instruccions.Aquest mètode provocarà la pèrdua de cicles en l'espera de la condició i per tant es perd rendiment en el programa tot i fent servir els curtcicuïts que disposem.



Les perdues de rendiment que es poden produir són força importants com es pot observar en aquest exemple:

Si suposem que el programa que s'executa té aproximadament un 30% de salts i en cada salt es perd un cicle de rellotge això és indicaria que:

$$\text{Millora}_{seg} = \frac{\text{Cpi}^{ideal} \text{ x n etapes}}{\text{Cpi}^{ideal} + \text{cicles perduts}}$$

$$\text{Perdua de rendiment} = \frac{\text{Millora}^{ideal}}{\text{Millora}^{real}} = \frac{\frac{1 \text{ x n etapes}}{1.2}}{\frac{1 \text{ x n etapes}}{1 + 0.3 \text{ x } 1}} = 1.3$$

Això representa que es perd al voltant de un 30% de rendiment degut als salts

9. Tècniques per reduir el risc de salts d'instruccions.

Per poder optimitzar el temps de pèrdua de salt es pot fer mitjançant el mètode de retardar el salt. Aquest mètode es pot implementar de dues formes:

1) ***Delay****: Retard de salt sense anul·lació.*

La màquina haurà de continuar la seqüència de les instruccions fins que no s'avaluï el salt per a conèixer si el salt s'ha de produir o no. Això implicarà que per poder tracta un salt s'haurà d'introduir instruccions que no afectin a la lògica d'execució del procés.

Per al correcte funcionament d'aquest mètode és molt important conèixer els cicles de latència que requereix la màquina per avaluar la condició de salt, és a dir, el temps que passa des de l'inici d'execució de la instrucció fins que l'avaluació de salt és disponible. Cal tenir en compte que aquest valor està fortament lligat a l'arquitectura que disposa el sistema i sobretot els possibles curtcircuïts que disposi entre els diferents recursos (avançar en resultat sense esperar a la finalització de l'execució).

Per exemple si disposem una arquitectura segmentada amb el següent pipeline:

a)

Fetch	Desc/lect PC+X,Aval	ALU	Mem	Escriptura

Tindrà una latència de dos cicles per conèixer el valor del salt.

Si el pipeline fos el següent:

b)

Fetch	Desc/lect PC+X	ALU Aval	Mem	Escriptura

Tindrà una latència de tres cicles per conèixer el valor del salt.

Aquestes tècnica com es pot observar recau tot el pes sobre el compil.lador donat que és el que haurà d'optimitzar el codi pel correcte funcionament del sistema.

Hi ha dues formes d'implementar aquest model:

- La primera consistiria en introduir sentències NOP que no realitzaran cap tasca.

  En el cas de que hi hagi 2 cicles de latència per a conèixer el valor d'una condició caldrà per tant introduir dos NOP davant del salt. Com es pot observar aquesta tècnica no pararà l'execució del programa però el rendiment del mateix disminueix donat que les instruccions NOP no realitzen cap tasca.

Per exemple si tenim el següent codi agafant un pipeline del tipus a):

```
Abans:

        add r1, r2, r3
        add r5,r2,r4
        complt r7,r5,r4
        jmp r7,salt
       mult r4,r1,r5
salt: load r1, (r3)
```

```
Després amb delays:

        add r1, r2, r3
        add r5,r2,r4
        complt r7,r5,r4
        jmp r7,salt
        nop
        nop
       mult r4,r1,r5
salt: load r1, (r3)
```

- La segona seria moure instruccions que no afectin l'execució del programa e introduir-les darrera la instrucció de salt de forma que gastin els cicles que es necessiten per avaluar la condició de salt. Aquest mètode aconsegueix millorar el rendiment en front l'anterior donat que intenta omplenar els cicles que es perden de latència introduint instruccions entremig.

  Per implementar aquest mètode requerirem dividir el programa amb blocs bàsics.

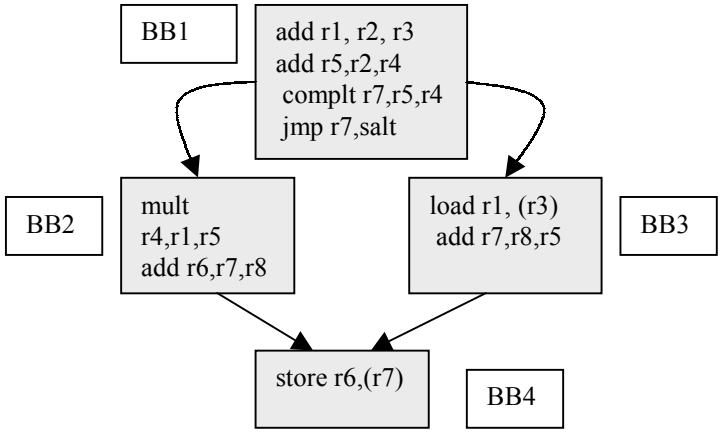
  Un Bloc bàsic és un conjunt d'instruccions que s'executen seqüencialment que té per primera instrucció la primera instrucció del programa o la primera instrucció després d'una instrucció de salt o la primera instrucció que marca una instrucció de salt, i que té per darrera instrucció la darrera instrucció abans d'un salt o la darrera instrucció del salt.

  Per exemple:

Si tenim el següent programa:

```
        add r1, r2, r3
        add r5,r2,r4
        complt r7,r5,r4
        jmp r7,salt
       mult r4,r1,r5
       add r6,r7,r8
       br fi
salt: load r1, (r3)
        add r7,r8,r5
fi:     store r6,(r7)
```

La divisió en blocs seria la següents:



Una vegada organitzat tot en Blocs bàsics el compilador per realitzar delays utilitzarà només les instruccions que pertanyin al mateix bloc bàsic, sempre i quan la modificació no afecti a la lògica del programa. En el nostre cas la resposta seria:

```
      add r5,r2,r4
      complt r7,r5,r4
      jmp r7,salt
      add r1, r2, r3
      nop
      mult r4,r1,r5
      add r6,r7,r8
      br fi
salt: load r1, (r3)
       add r7,r8,r5
fi:    store r6,(r7)
```

Com es pot observar en aquest exemple degut a que la primera instrucció no es pot moure donat a que es modificaria la lògica de l'execució s'ha d'incloure un NOP, és a dir que es poden establir mètodes mixtes.

Un exercici que ens compararia els dos mètodes seria el següent:

En un segmentat amb el pipeline de la figura, que utilitza un Delay-load i un Delay-barch s'executa el següent codi:

Fetch	Desc./lect PC+X,Aval	ALU	Mem	Escipt.

```
1       const    r1, base_a          ;
2       load     r2, (r1)            ;
3       add      r3, r2, 8           ;
4       load     r4, (r3)            ;
5       const    r5, base_b          ;
6       load     r6, (r5)            ;
7       add      r7, r6, 8           ;
8       load     r8, (r7)            ;
9       cmplt    r9, r8, r4          ;
10      jmp      r9, else            ;
11      store    r8, (r3)            ;
12      br       fi                  ;
13 else:         store               r4, (r7)     ;
14   fi:
```

Contesta:

a) Col·loca NOP als llocs a on faci falta.
b) Quants cicles costa l'execució del programa segons l'apartat a)?
c) Optimitza el codi a fi de reduir el temps d'execució. Quants cicles costa ara?

Resposta:

a) NOPs

```
1       const    r1, base_a          ;
2       load     r2, (r1)            ;
        NOP                          ; 1
3       add      r3, r2, 8           ;
4       load     r4, (r3)            ;
        NOP                          ; 2
5       const    r5, base_b          ;
6       load     r6, (r5)            ;
        NOP                          ; 3
7       add      r7, r6, 8           ;
8       load     r8, (r7)            ;
        NOP                          ; 4
9       cmplt    r9, r8, r4          ;
10      jmp      r9, else            ;
        NOP                          ; 5
11      store    r8, (r3)            ;
12      br       fi                  ;
        NOP                          ; 6
13 else:         store               r4, (r7)     ;
14   fi:
```

b) Teòricament, tardaria en mitjana 19 cicles (13 instruccions + 6 NOPS = 19 instruccions a 1 instrucció per cicle = 19 cicles)
En realitat tarda 23 cicles (19 cicles + 4 d'inicialització)

c) Si es fa un anàlisi de dependències podem reordenar les instruccions per poder emplenar els delays. Un possible resultat és:

```
1       const    r1, base_a          ;
5       const    r5, base_b          ;
2       load     r2, (r1)            ;
6       load     r6, (r5)            ;
3       add      r3, r2, 8           ;
7       add      r7, r6, 8           ;
8       load     r8, (r7)            ;
4       load     r4, (r3)            ;
9       cmplt    r9, r8, r4          ;
10      jmp      r9, else            ;
        NOP                          ; no hi ha op. independents
11      store    r8, (r3)            ;
12      br       fi                  ;
        NOP                          ; no hi ha op. independents
13 else:         store               r4, (r7)     ;
14   fi:
```

Teòricament, tardaria en mitjana 15 cicles (13 instruccions + 2 NOPS = 15 instruccions a 1 instrucció per cicle = 15 cicles)
En realitat tarda 19 cicles (15 cicles + 4 d'inicialització)

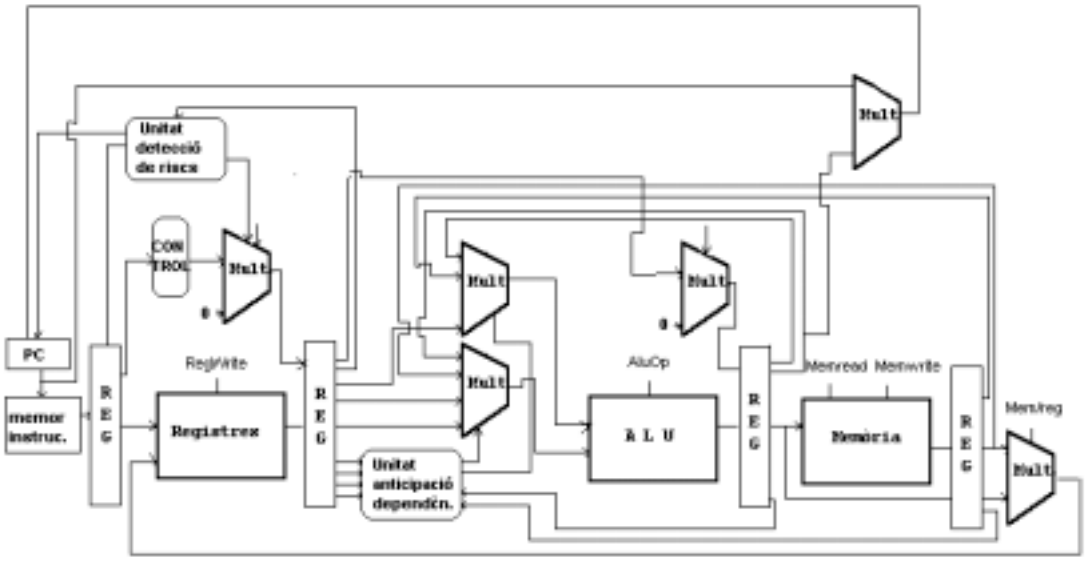
2) *Fordwarding: Retard de salt amb anul·lació.*

Suposar que no es produirà el salt i continuar l'execució seqüencial.
Aquest mètode és més òptim donat que la majoria de salts en la realitat no es solen produir i per tant es guanya en rendiment.
Les instruccions es continuaran executant de forma seqüencial l'única cosa que hem de tenir en compte és el fet de que en el cas de que es produeixi el salt les instruccions que s'han

executat s'han d'anul·lar. Una de les formes d'anul.lar seria no escrivint els resultats de les instruccions que s'han d'eliminar.

Per anul·lar les instruccions només haurem d'incloure un dispositiu de control que ens permeti ficar a zero els valor de les senyals de control dels registres de segmentació de les fases de instrucció, registre i execució.

A part també haurem de permetre modificar el registre d'instrucció amb el valor de la instrucció del salt i modificar el valor del comptador d'instruccions.



Com es pot observar s'han introduït 2 multiplexors entre els registres de segmentació (1 en la fase d'execució i l'altre en la fase de registre) que juntament amb el que existeix en la fase d'instrucció ens permet ficar a zero el valor dels senyals de control i així podrem eliminar els continguts de les instruccions que hem executat i no serveixen.

També s'ha introduït un altre multiplexor en la fase d'execució que serà l'encarregat de passar al comptador d'instruccions el valor de la següent instrucció a executar, és a dir, la consecutiva o la que correspon al salt.

Amb aquesta estructura solucionem els problemes que ens podem trobar a l'hora d'executar qualsevol programa en un sistema segmentat, només ens quedarà resoldre el problema que es pot produir en cas d'una excepció.

3) *Especulatiu: Especulació del possible resultat del salt amb anul·lació.*

Aquest mètode és una variant de l'anterior però en aquest cas no necessàriament es segueix la seqüència de l'execució sinó que el salt es realitzarà o no responent unes estadístiques. Aquest mètode el tractarem quan parlarem dels supersegmentats.

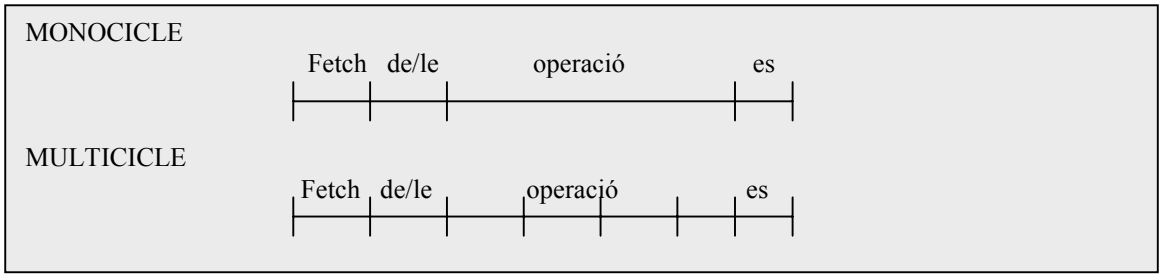
10.Sistemes segmentats multicicles

Fins ara els sistemes segmentats que hem estat veien tenen com a característiques que totes les etapes d'execució del pipeline tarden un cicle d'execució i per tant el cicle d'execució s'adapta a la mesura del cicle més llarg.

Per intentar optimitzar l'execució dels sistemes segmentats apareixen els sistemes multicicles que permeten segmentar les etapes. Això té dues aventatges:

- Reduir el tamany de les etapes.
- Permetre l'execució concurrent dins de una mateixa etapa d'instruccions diferents que es trobin en diferents segments de l'etapa.

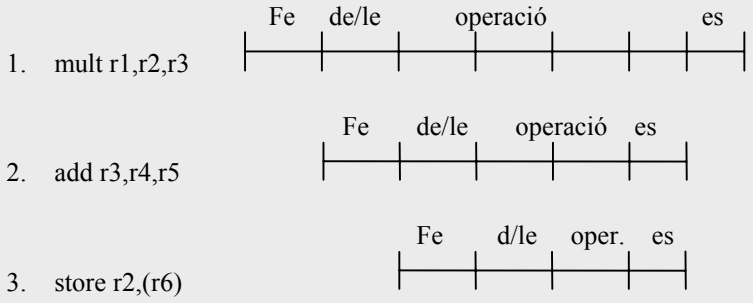
Aquests dos aventatges permeten augmentar d'una forma important la velocitat d'execució dels programes tan perquè les etapes ràpides no desaprofiten tant de temps al ser més petit el tamany de cicle i perquè les etapes de major durada (ex. ALU) es poden trossejar en bocins més petits que es poden executar en paral.lel.



En contrapartida aquests sistema es reporta alguns possibles problemes que s'han de resoldre que serien:

- Les instruccions poden acabar en desordre: Poden haver-hi instruccions que tenen un nombre de cicles inferiors a altres que fan que acabin abans per exemple:

Si suposem que tenim tres pipelines: un per operacions senceres de amb dos cicles d'operacions; un per operacions amb coma flotant amb 4 cicles d'operacions; i un de lectures/escriptures i salts amb un 1 cicle per operacions.

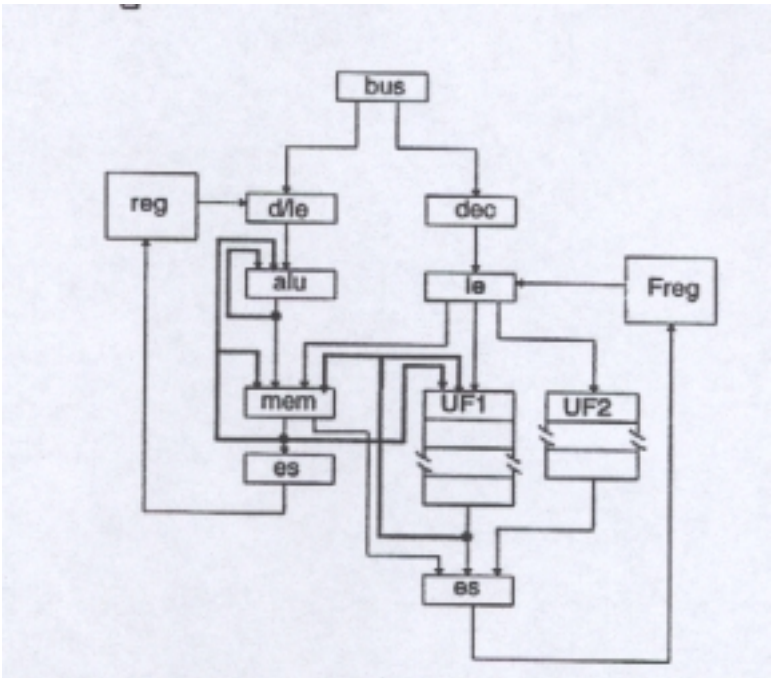


Com es pot observar en l'execució la instrucció 2 i 3 acaben abans que la instrucció 1.

- L’aparició del risc en la dependència de dades WAW provocat pel fet que les instruccions acabin en desordre. Com ja havíem parlat amb anterioritat això es produirà quan el compil.lador amb el que treballem no ho faci d’una forma correcta. Per solucionar aquest problema es pot parar l’execució de la instrucció o eliminar l’escriptura més antiga.

- Cal tenir en compte el fet de que a l’haver un nombre més gran d’instruccions en execució això farà augmentar el nombre de dependències a nivell estructural. Per intentar evitar el màxim possible aquestes s’hauran d’establir el màxim possible de curtcircuits en l’arquitectura.

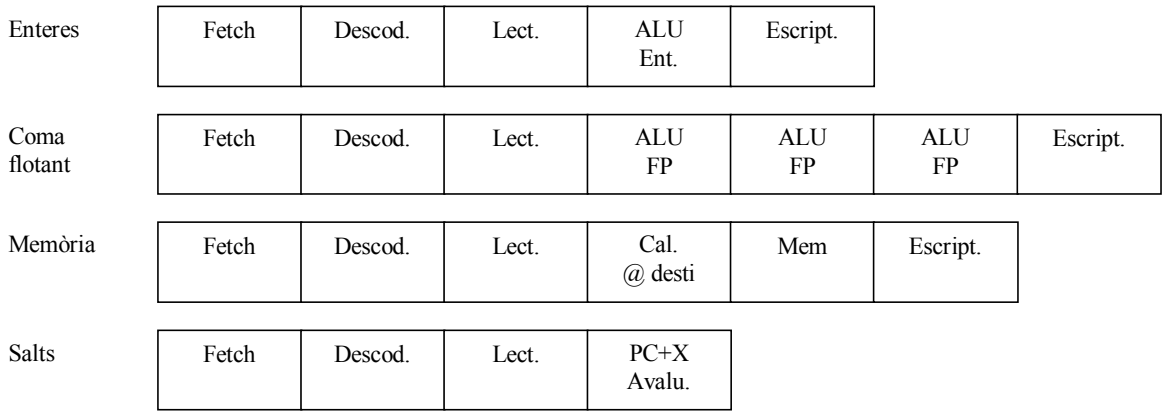
Un exemple d’arquitectura seria el següent:



11.Exercicis

PROBLEMA 1

Donat un processador segmentat amb pipelines diferents segons el tipus de funció, i amb un únic banc de registres compartit entre operacions enteres i de coma flotant:



Contesta:

b) Quins conflictes estructurals es poden produir, i quin hardware fa falta per evitar-los.

b) Analitza els possibles conflictes de control? Proposa almenys dos solucions.

b) Quins conflictes de dades es poden produir? Proposa almenys dos solucions.

SOLUCIÓ:

a) Simultàniament es poden executar les següents etapes:

- Accés a memòria:
  - Fetch
  - Mem

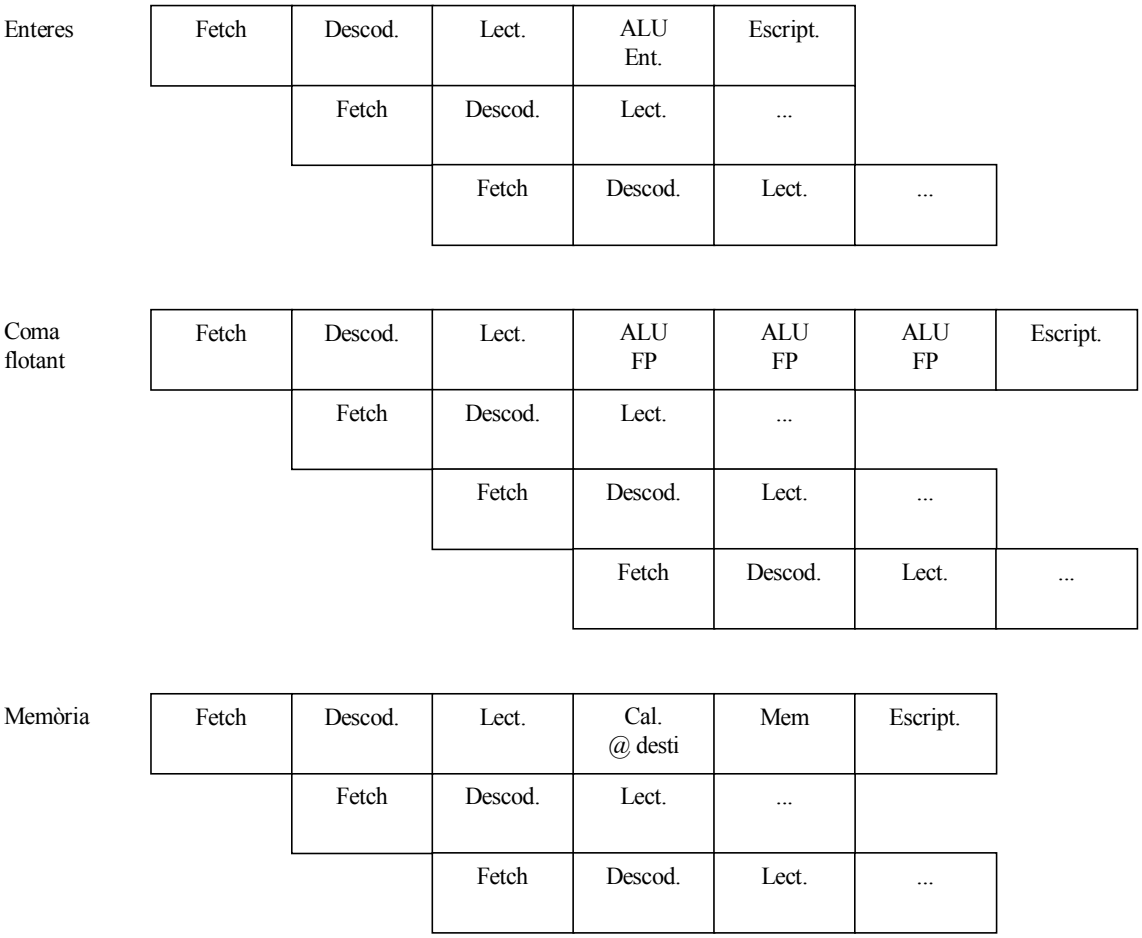
  Per tant fan falta dos camins d'accés a memòria.
- Càlcul:
  - ALU Ent.
  - ALU FP.
  - Cal. @ destí
  - PC+X
  - Avaluació

  Per tant feria falta un element per cada acció anterior.
- Banc de Registres:
  - Escriptura d'enters (1 registres)
  - Escriptura de FP. (1 registre)
  - Escriptura de memòria (1 registre)
  - Lectura (2 registres)

  Per tant fan falta 3 camins d'escriptura i dos de lectura al banc de registres

b) Al executar-se una instrucció de salt hi ha un retard de tres cicles per saber quina instrucció ha de ser la pròxima. Per evitar aquesta situació es pot fer:
   - Parar el processador surant aquest 3 cicles.
   - Utilitzar 3 delay-slots.
   - Utilitzar tècniques de predicció (dinàmiques o estàtiques) i anticipació del càlcul del PC (BTB) combinat amb execució especulativa.

c) Es poden produir conflictes degut a les dependències RAW i WAW (finalització en desordre).



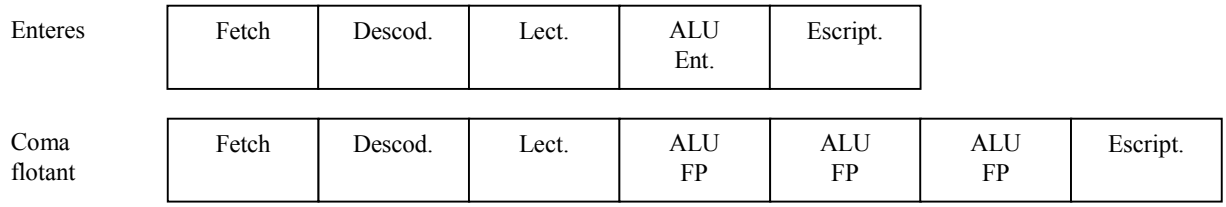
- RAW: Es poden produir com a conseqüència de la combinació de una operació aritmètica, una de CF o una d'accés a memòria:
  - Aritmètica: En aquest els conflictes es poden solucionar: parant el processador fins que desapareix-hi el conflicte o amb forwardings
  - Coma Flotant i memòria: En aquests dos casos hi han tres possibilitats: parant el processador, o parant el processador més forwardings, o aplicar l'idea dels delays junt amb els forwardings.

- WAW: Degut a la finalització en desordre. Es pot solucionar: parant el processador i ordenant les escriptures per a totes les instruccions (això és equivalent a fer els quatre

pipelines amb el mateix nombre d'etapes), En cas de conflicte ordenant les escriptures, o aplicar Tomasulo.

PROBLEMA 2

Donat un processador segmentat amb pipelines diferents segons el tipus de funció, i amb un únic banc de registres compartit entre operacions enteres i de coma flotant:



Si tenim la següent seqüència de codi:

```
1       add     r1, r2, r3        ; suma entera
2       fpmul   r3, r1, r4        ; Multiplicació en CF
3       subb    r3, r3, r5        ; resta aritmètica
4       fpdiv   r1, r2, r3        ; Divisió en CF
```

Contesta:

a) Quines dependències hi ha i quins conflictes de dades es poden produir? Proposa almenys dos solucions.
b) Quants cicles tarda en executar-se (un cop solucionats els conflictes) ?

SOLUCIÓ:

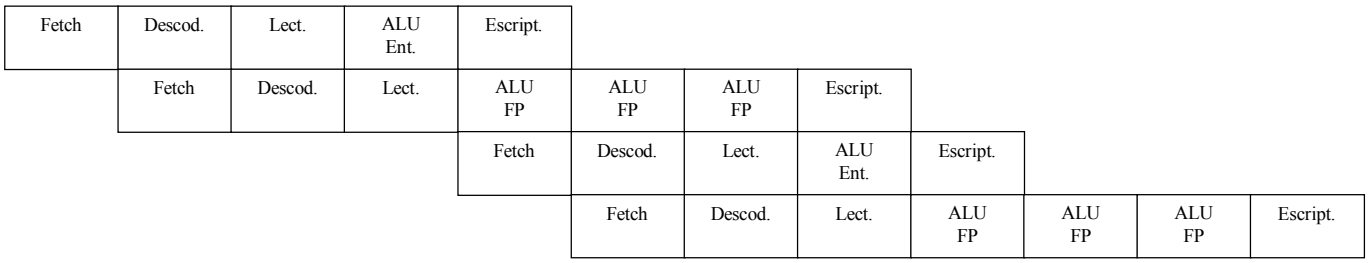
a) Tenim les següents dependències:
   - RAW:
     - entre la 1 i la 2 per r1
     - entre la 2 i la 3 per r3
     - entre la 3 i la 4 per r3
   - WAW:
     - entre la 2 i la 3 per r3
     - entre la 1 i la 4 per r1
   - WAR:
     - entre la 1 i la 2 per r3
     - entre la 1 i la 3 per r3
     - entre la 2 i la 4 per r1

   Els conflictes que es poden produir són RAW i WAW:

- RAW:
  - entre la 1 i la 2 per r1. Parar l'execució de 2 o fer forwardings
  - entre la 2 i la 3 per r3. Parar l'execució de 3 o fer dos delays+forwarding
  - entre la 3 i la 4 per r3. Parar l'execució de 4 o fer forwardings

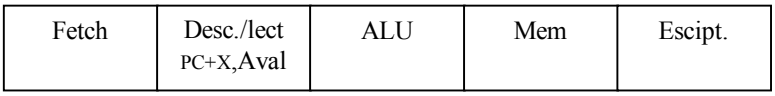
- WAW:
  - entre la 2 i la 3 per r3. La dependència RAW entre 2 i 3 fa que no es produeixi el conflicte, i finalitzin en ordre. No obstant, si no hi hagués la dependència RAW s'hauria de retardar l'escriptura de la 3, o afegir més etapes al pipeline enter a fi que les escriptures es facin sempre en ordre.
  - entre la 1 i la 4 per r1. No produeix cap conflicte, ja que finalitzen en ordre.

b) Tarda 12 cicles:



PROBLEMA 3

En un segmentat amb el pipeline de la figura, que utilitza un Delay-load i un Delay-barch s'executa el següent codi:




```
1       const   r1, base_a      ;
2       load    r2, (r1)        ;
3       add     r3, r2, 8       ;
4       load    r4, (r3)        ;
5       const   r5, base_b      ;
6       load    r6, (r5)        ;
7       add     r7, r6, 8       ;
8       load    r8, (r7)        ;
9       cmplt   r9, r8, r4      ;
10      jmp     r9, else        ;
11      store   r8, (r3)        ;
12      br      fi              ;
13 else:        store           r4, (r7)   ;
14   fi:
```

Contesta:

a) Col·loca NOP als llocs a on faci falta.
b) Quants cicles costa l'execució del programa segons l'apartat a)?
c) Optimitza el codi a fi de reduir el temps d'execució. Quants cicles costa ara?
d) Té sentit aplicar List scheduling?

SOLUCIÓ:

a) NOPs

```
1       const   r1, base_a      ;
2       load    r2, (r1)        ;
        NOP                     ; 1
3       add     r3, r2, 8       ;
4       load    r4, (r3)        ;
        NOP                     ; 2
5       const   r5, base_b      ;
6       load    r6, (r5)        ;
        NOP                     ; 3
7       add     r7, r6, 8       ;
8       load    r8, (r7)        ;
        NOP                     ; 4
9       cmplt   r9, r8, r4      ;
10      jmp     r9, else        ;
        NOP                     ; 5
11      store   r8, (r3)        ;
12      br      fi              ;
        NOP                     ; 6
13 else:        store           r4, (r7)   ;
14   fi:
```

b) Teòricament, tardaria en mitjana 19 cicles (13 instruccions + 6 NOPS = 19 instruccions a 1 instrucció per cicle = 19 cicles)
En realitat tarda 23 cicles (19 cicles + 4 d'inicialització)

c) Si es fa un anàlisi de dependències podem reordenar les instruccions per poder emplenar els delays. Un possible resultat és:

```
1       const   r1, base_a       ;
5       const   r5, base_b       ;
2       load    r2, (r1)         ;
6       load    r6, (r5)         ;
3       add     r3, r2, 8        ;
7       add     r7, r6, 8        ;
8       load    r8, (r7)         ;
4       load    r4, (r3)         ;
9       cmplt   r9, r8, r4       ;
10      jmp     r9, else         ;
        NOP                      ; no hi ha op. independents
11      store   r8, (r3)         ;
12      br      fi               ;
        NOP                      ; no hi ha op. independents
13 else:        store            r4, (r7)   ;
14   fi:
```

Teòricament, tardaria en mitjana 15 cicles (13 instruccions + 2 NOPS = 15 instruccions a 1 instrucció per cicle = 15 cicles)
En realitat tarda 19 cicles (15 cicles + 4 d'inicialització)

d) No té sentit ja que el processador és un segmentat i no podem explotar el paral·lelisme.

PROBLEMA 4

Donat un processador segmentat amb el següent pipeline. La etapa d'execució pot ser de durada variable en funció de l'operació (vegeu latències de sortida) i amb tantes unitats funcionals com ens facin falta.

Fetch 1	Fetch 2	Descod.	Lect.	ALU	...	ALU	Escript.

Si tenim la següent seqüència de codi:

```
1       r1 <- r2 op r3           ; latència de sortida 4 cicles
2       r4 <- r1 op r5           ; latència de sortida 2 cicles
3       r1 <- r5 op r7           ; latència de sortida 1 cicle
4       r8 <- r1 op r4           ; latència de sortida 2 cicles
```

Contesta:

a) Quines dependències entre instruccions hi ha?

b) Si suposem un processador amb execució de les instruccions amb inici i finalització amb ordre, quines dependències es converteixen en conflictes? Com ho solucionaries? Dibuixa l'evolució del pipeline.

c) Si suposem un processador amb execució de les instruccions amb inici amb ordre i finalització sense ordre, quines dependències es converteixen en conflictes? Com ho solucionaries? Dibuixa l'evolució del pipeline.

SOLUCIÓ:

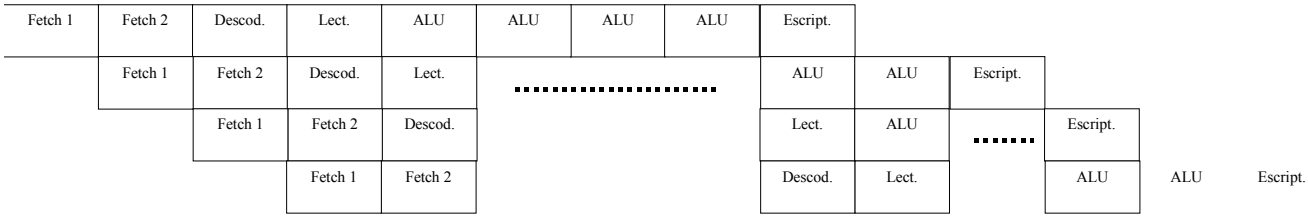
a)

RAW: 2<-1 per r1
4<-2 per r4
4<-3 per r1

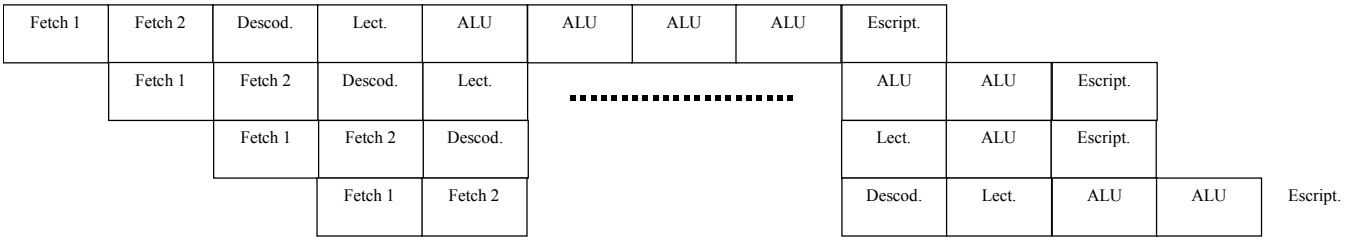
WAW: 1 i 3 per r1

WAR: 2 i 3 per r1

b) Només són conflictes les dependències RAW. Es solucionaria parant el processador i/o fent un forwarding.



c) Són conflicte les dependències RAW i WAW. Les dependències RAW es solucionarien de la mateixa manera que en l'apartat anterior. Les dependències WAW en aquest exemple queden solucionades automàticament al solucionar les RAW, no obstant, si fes falta es pararia l'escriptura més nova, o s'invalidaria la més antiga.



PROBLEMA 5

El processador segmentat de la figura utilitza un Delay-barch pels conflictes de control, i pels conflictes de dades utilitza forwardings i, en el cas de no ser suficient, para el processador:

Fetch	Desc./lect $PC+X$,Aval	ALU	Mem	Escipt.

Si volem executar el següent codi:

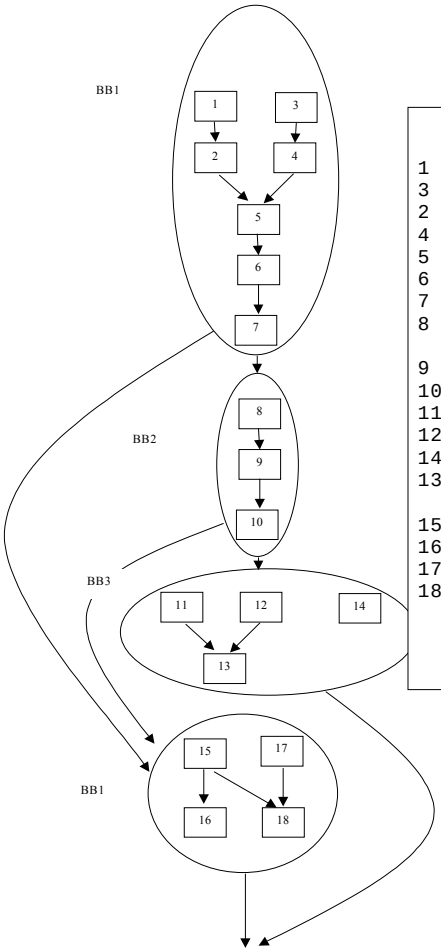
```
1          load  r1 <- base_a(r9)
2          sll   r0 <- 2,r1
3          load  r2 <- base_b(r9)
4          sra   r3 <- 2,r2
5          and   r4 <- r3,r0
6          cmp   r5 <- r4,100
7          bg    r5,LL2
8          load  r6 <- base_c(r9)
9          cmp   r7 <- r6,0
10         be    r7,LL2
11         load  r8 <- base_c(r9)
12         const      r10 <- 27
13         store      r10 -> [r8]
14         br    LL3
     LL2:
15         const      r11 <- 100
16         store      r11 -> base_c(r9)
17         const      r12 <- 38
18         store      r12 -> [r11]
     LL3:
```

Contesta:

a) Modifica el codi a fi que es pugui executar correctament en aquest processador. Fes–ho de la manera més òptima possible.

b) Quin grau de paral·lelisme en mitjana té aquest programa (fes-ho pel codi original i dividit per Blocs Bàsics), i quin grau de paral·lelisme té el processador?

a)




```
            CODI REORDENAT

1       load    r1 <- base_a(r9)
3       load    r2 <- base_b(r9)
2       sll     r0 <- 2,r1
4       sra     r3 <- 2,r2
5       and     r4 <- r3,r0
6       cmp     r5 <- r4,100
7       bg  r5,LL2  ; delay innocuo
8       load    r6 <- base_c(r9)
        (pararà un cicle)
9       cmp     r7 <- r6,0
10      be  r7,LL2  ; delay innocuo
11      load    r8 <- base_c(r9)
12      const   r10 <- 27
14      br  LL3     ; salt reordenat
13      store   r10 -> [r8]
    LL2:
15      const   r11 <- 100
16      store   r11 -> base_c(r9)
17      const   r12 <- 38
18      store   r12 -> [r11]
    LL3:
```

b) Grau de paral·lelisme del processador = 1 instrucció en paral·lel
Grau de paral·lelisme del BB1 = 7/5 (instruccions/cicle)

BB2 = 1
BB3 = 4/2
BB4 = 4/2