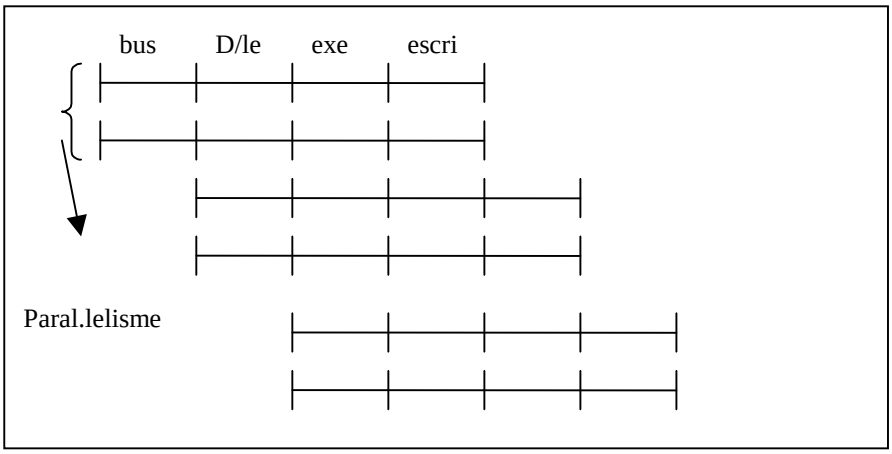
INDEX

1. DEFINICIÓ DE PROCESSADOR SUPERESCALAR

La principal característica que tenen els processadors superescalars és el fet de permetre inici i l'execució de forma paral.lela de més d'una instrucció en el mateix cicle. Per permetre això requereix que les unitats funcionals es repliquin i s'introdueixin noves vies de comunicacions entre unitats funcionals.

El grau de paral.lelisme d'un processador superescalar ens definirà el nombre d'instruccions que es busquen en cada cicle. Si un processador té grau de paral.lelisme 2 vol dir que cada cicle de rellotge si la unitat de control ho permet (no hi ha conflictes de dades ni estructurals) entraran 2 instruccions a executar-se.



Aquesta tecnologia es fonamenta en el fet de que la tecnologia avança de forma que la capacitat d'integració augmenta de forma quadràtica i per tant es poden introduir un nombre major unitats en la mateixa àrea.

Els cicles del processadors segueixen essent els mateixos però el rendiment augmenta com es pot observar:

$$T = NxCPI^{\sup es} xt_{ciclo}$$

Donat que es busquen m instruccions per cicle això vol dir que el CPI s'ha reduït segons el nombre d'instruccions que s'executen de forma paral.lela així tenim que:

$$CPI^{SUPES} = 1/m \, xCPI^{SEG}$$

Per tant si substituïm tot el resultat final serà el següent:

$$T = NxCPI^{SEG} x \frac{1}{m} xt_{ciclo}$$

Com es pot observar és molt millor que en el segmentat.

Pel que fa referència al paral.lelisme cal tenir en compte a l'existent entre les instruccions i el processadors i la relació que existeixen entre elles dos. Això es deu a que el fet de tenir més unitats funcionals gràcies a la duplicació de la mateixes no implica major grau de paral.lelisme. Les instruccions poden un grau de dependència important el qual pot fer que redueixi molt el rendiment del sistema.

Per intentar explotar el paral.lelisme entre les instruccions haurem de dotar al processadors d'una sèrie de mecanismes com són:

- Mecanismes per a detectar independència entre instruccions.
- Mecanismes que permetin executar o no parar el processador quan aquest detecte instruccions dependents.
- Dotar del nombre de unitats funcionals suficients per evitar conflictes per la manca d'elles.
- Camins amb l'ample de banda adequat per comunicar-se les unitats.

2. EXECUCIÓ D'INSTRUCCIONS EN DESORDRE (INICI I FINALITZACIÓ), (CONFLICTES RAW I WAW).

En el cas dels processadors superescalars per optimitzar el seu rendiment tant de trhougput com la utilització de les unitats funcionals que el formen la execució de les instruccions comencen en ordre però la seva finalització pot ser desordenades. Això es coneix com a planificació dinàmica aquesta la realitza el hardware reorganitzant l'execució de les instruccions en el moment de l'execució.

Quan s'inicia l'execució la instrucció es descodificarà i a partir d'aquí ja es coneixerà tant les unitats funcionals com les dades que requereix. Una vegada fet això les instruccions s'aniran executant segons tinguin les dades i les unitats disponibles. Si una instrucció no es pot executar però la següent sí que ho pot fer sense afectar en resultat de les que la precedeixen aquesta s'executarà. Per aquest motiu juntament amb els diferents temps de latència

de les operacions faran que es pugui produir un desordre en la finalització de les instruccions.

La planificació dinàmica en front de la estàtica te els següents avantatges:

1. Habilita el tractament de casos en que les dependències són desconegudes en el temps de compilació, simplificant d'aquesta manera el compilador.

2. Permet que un programa compilat per un determinat processador segmentat pugui ser executat per un altre processador segmentat totalment diferent.

Com a desavantatge hi ha la complexitat del maquinari.

Els problemes que ens pot comportar l'execució desordenada d'instruccions afectarien als salts dintre les instruccions i la dependència de les dades.

Com ja havíem previament en el tema de la segmentació es poden produir en l'execució dependència entre dades, que en el cas dels processadors superescalars augmenta la proporció degut al major nombre d'instruccions que s'executen alhora. Les tres possibles dependències que haurem de considerar són les següents:

1. **WAW :** Dues instruccions que escriguin en el mateix registre i pel fet de l'execució desordenada que la darrera instrucció escrigui en el registre abans que la primera.

   Per exemple:

   ```
   DIVF  F0,F2,F4
   ADDF F10,F0,F8
   SUBF F10,F8,F10
   ```

   Si s'executa abans SUBF que ADDF el registre F10 contindria un valor incorrecte.

2. **WAR:** La instrucció escriu el valor en un registre abans que l'anterior hagi llegit d'aquest registre.

Per exemple:

```
DIVF  F0,F2,F4
ADDF F5,F10,F8
SUBF F10,F8,F10
```

Si s'executa abans SUBF que ADDF el registre F10 contindria un valor incorrecte.

3. **RAW:** La instrucció desordenada llegeix el contingut d'un registre abans que l'intrucció que li precedeix escrigui el valor que li pertoca en el mateix

   Per exemple:

```
DIVF  F0,F2,F4
ADDF F6,F10,F8
SUBF F10,F6,F10
```

   Si s'executa abans SUBF que ADDF el registre F6 contindria un valor incorrecte quan SUBF fa la lectura.

Els conflictes de dades es produeixen quan, en temps d'execució, alguna dependència entre instruccions fa que no poguem executar-ne alguna fins que es resolgui aquesta dependència.

3. TÈCNIQUES DE PLANIFICACIÓ: SCOREBOARD I TOMASULO

Per intentar solucionar els conflictes de dependència de dades veurem dos algorismes de planificació dinàmica per reduir el seu cost: El Scoreboard i el Tomasulo.

En tots els mètodes que utilitzarem haurem de tenir en compte una paràmetre molt important que ens pot afectar en la solució del mateix la latència d'entrada i sortida de les instruccions en una unitat funcional.

La latència d'entrada és el temps que ha de passar entre dues instruccions que volen entrar en una mateixa unitat funcional. És a dir, si una multiplicació en una unitat funcional de coma flotant és de 2, vol dir que cada dos cicles de rellotge poden entrar 2 instruccions. El fet de mesurar la

latència d'entrada es deu a que gràcies a mètodes de pipeline se'ns permet tenir més d'una instrucció executant-se en la mateixa unitat funcional.

La latència de sortida és el temps que tarda una instrucció en acabar la seva operació en la unitat funcional, és a dir, si una instrucció de multiplicació té latència 4, vol dir que requereix 4 cicles de rellotge per poder obtenir el resultat.

Sempre la latència d'entrada serà menor o igual que el de la sortida.

Un altre concepte important és el número de vies que indica el nombre de instruccions que podem fer la seva fase emetre alhora, és a dir, si el nombre de vies és dues ens diu que podem emetre dos instruccions en el mateix moment en cada cicle de rellotge.

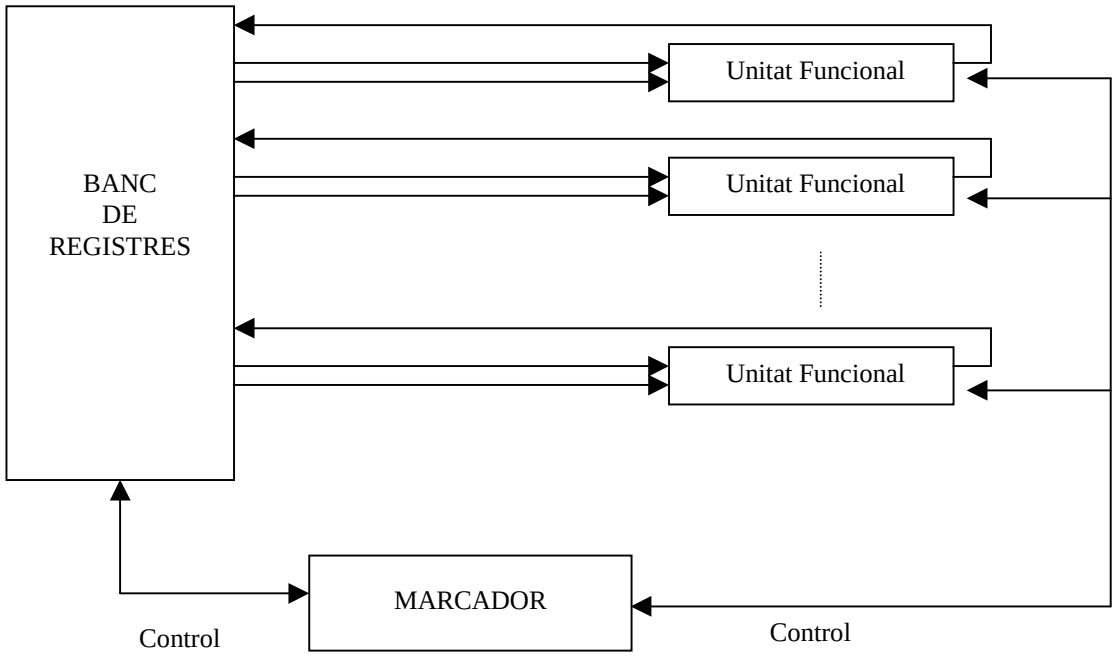
3.1. SCOREBOARD.

L'objectiu d'aquest sistema de planificació és el de permetre l'execució en desordre de les instruccions. Per a que això sigui possible, es defineix un *pipeline* d'execució per a totes les instruccions format per les següents etapes: *emissió, lectura d'operants, execució i escriptura del resultat*.

Totes les instruccions són emeses (descodificades i assignades) en ordre, però a l'arribar a l'etapa de lectura poden provocar-se detencions si els operants no estan encara disponibles. En aquest cas, només es deté la instrucció que no disposa dels operants, mentre que les que puguin fer la lectura continuaran l'execució.

Per a que aquest algorisme provoqui una millora de rendiment necessitarem més d'una unitat funcional de cada tipus o segmentar-les. Suposarem que tenim unitats funcionals replicades. Però s'ha de tenir en compte que el nombre de bussos que comuniquen els registres amb les unitats funcionals són limitats i per tant el marcador haurà de vetllar per a que no es sobrepassin el nombre de bussos que tenim.

Per a controlar els possibles errors que es produeixen en aquest mètode s'utilitza una estructura bàsica anomenada "marcador" la funció de la qual és controlar l'execució de les instruccions intentant mantenir la velocitat d'execució en un cicle per segon. El marcador serà el responsable d'emetre i executar instruccions incloent la detecció de riscs.



El marcador està format per tres parts:

1) Taula d'estat de es instruccions on s'indica en quina de les etapes es troba la instrucció.

	1	2	3	4	5	6	7	8	9
MULT S1,S2,S3	F	D	L	X	X	X			
MULT S5,S1,S3	F	D							
CMP S6,S1,0		F	D						
BNE S6, LL1		F	D						
ADD S7,S2,S3			F	D					
ADD S3,S4,S5			F	D					
ADD S2,S4,S2									
LL1: SUB S3,S2,S3									
MULT S4,S4,S1									

2) Taula d'estats de la unitat funcional on s'indica l'estat de les unitats funcionals. Cada unitat tindrà els següents camps:
   - Ocupat: indica si el número d'instrucció que l'està ocupant.
   - Op: Operació a realitzar en la unitat.
   - Fi: Registre destí.
   - Fj,Fk: Registres fonts.

- Qj,Qk: Número d'unitats que produeixen els registres fonts Fj i Fk (si un registre prové de la unitat +1,+2,...).
- Rj,Rk: Indica si els operants estan disponibles o no.

NOM	OCUP	OPER	FI	FJ	QJ	FK	QK	RJ	RK
+1	1	+	R1	R2	SEN1	R3		NO	SI
+2									
+3									
*1									
*2									
SEN1									
SEN2									

3) Estat dels registre indicant quina unitat funcional escriurà en el registre si una instrucció activa té el registre com a destí. En cas que el registre ja tingui el valor es posarà en Vi, mentre que si el valor depèn de el resultat d'una unitat funcional es possarà en Qi indicant de quina unitat funcional prové.

	S1	S2	S3	S4	S5	S6	S7
Vi		(*1)					
Qi	(+1)						

Vegem amb més detall la funció de cada etapa:

- **Fetch** :

  Fase de búsqueda de la instrucció següent a decodificar. Aquesta fase s'anirà realitzant sempre que no es produeixi una detenció de l'emissió d'instruccions.

- **Emissió** (issue):

  Una instrucció s'emet si hi ha alguna unitat funcional lliure que la pugui executar i no hi ha cap altra instrucció ja emesa que escrigui sobre el mateix registre que aquesta. Amb això estem detectant una dependència WAW que, degut a la finalització en desordre pot provocar un conflicte i una incoherència en el banc de registres.

En cas que la instrucció no es pugui emetre per alguna de les causes descrites, no s'emetran ni ella ni les següents fins que desapareixi l'impediment.

- **Lectura d'operants**:

Es podran llegir els operants d'una instrucció quan aquests estiguin en el banc de registres. Per tant, no podrà haver-hi cap instrucció activa que hagi d'escriure algun dels registres que han de ser llegits. Tots els operants d'una instrucció es llegeixen de cop.

Amb aquestes precaucions estem solucionant els conflictes RAW.

- **Execució**:

Una unitat funcional podrà començar l'execució quan tingui tots els operants necessaris i la unitat estigui disponible.

- **Escriptura del resultat**:

Quan el control detecta que una unitat funcional ha generat un resultat, es comprova que no hi hagi un conflicte WAR, en cas de no existir, es pot procedir a guardar el resultat en el banc de registres.

Existirà un conflicte WAR quan es compleixin les tres condicions següents:

1. Hi ha una instrucció que no ha llegit els operants.
2. Un dels operants està en el mateix registre que el resultat de la instrucció completada.
3. L'altre operant és el resultat d'una instrucció anterior no finalitzada.

Un exemple de codi que pot provocar un conflicte WAR podria ser:

i) R0 ← R2 op R4
i+1) R10 ← R0 op R8
i+2) R8 ← R8 op R5

Si la instrucció *i* no ha finalitzat, la *i+1* no haurà pogut llegir els operants, mentre que la *i+2* s'haurà executat però no podrà escriure el resultat immediatament ja que si escrivís a R8, quan *i+1* llegís els operants no agafaria els valors correctes. Per tant, la instrucció *i+2*, s'haurà d'esperar a que *i+1* hagi llegit els operants per a poder escriure.

Cal dir que, en el cas que no tinguem un bus d'escriptura cap al banc de registres per cada unitat funcional, s'haurà de limitar, en un mateix cicle de rellotge, el nombre d'unitats que puguin escriure el resultat, aturant les restants.

Tot seguit veurem quina informació haurà de guardar el *marcador* per a poder complir amb l'estratègia de cada etapa.

Ara que coneixem l'estructura del marcador i la informació que s'hi guarda, podem repassar les etapes més importants de forma més específica:

En l'etapa d'emissió es busca una entrada associada a una unitat funcional que pugui executar la instrucció i s'emplenen tots els camps convenientment. Els valors de Qj i Qk s'agafen directament del camp addicional que hem afegit a cada registre i que indica quina unitat ha de generar-ne el nou valor. En cas que el registre destí estigui marcat per alguna unitat funcional estarem detectant un WAW i es para l'emissió. Si no es troba cap entrada vàlida, també s'atura l'emissió d'instruccions fins que n'hi hagi.

En l'etapa de lectura es busca en el marcador entrades amb els camps Rj i Rk indicant que els operants estan disponibles, es fa lectura dels mateixos en el banc de registres i s'inicia l'execució de la instrucció associada en la unitat funcional indicada.

Quan una unitat acaba la operació i pot escriure el resultat, avisa al marcador per que marqui com a disponible aquells camps Qj o Qk que feien referència a la unitat funcional que ha finalitzat.

Exemple:

Tenim la següent seqüència de codi.

```
1       add    r1, r2, r3     ;
2       sub    r2, r3, r2     ;
```

```
3       mul   r3, r2, r1     ;
4       add   r1, r0, r4     ;
5       sub   r3, r3, r5     ;
6       mul   r0, r4, r2     ;
7       sub   r1, r1, r5     ;
8       mul   r2, r3, r1     ;
```

Suposa un superescalar de 2 vies amb 2 Unitats Funcionals amb les següents característiques:

UF	Estacions de reserva	Operacions	Latència de finalització	Latència d'inici
ALU1	3	+, -	2	1
ALU2	2	*	4	2

Com es pot observar el la resolució de l'exercici que es troba en la pàgina següent es donen la següent sèrie de conflictes:

- En el cicle 3 es produeix un WAW entre la instrucció 1 i 4 en el registre R1 que fa que s'aturi la decodificació de la instrucció 4 i de les posteriors fins que aquest es solucioni.

- En el cicle 4 es produeix un conflicte estructural entre la instrucció 2 i la 3 produït per la latència d'entrada de la unitat funcional.

- En el cicle 8 es torna a donar un WAW entre la instrucció 3 i la 5 en el registre R3 que fa que s'aturi la decodificació de la instrucció 4 i de les posteriors fins que aquest es solucioni.

- En el cicle 17 es produeix un conflicte RAW entre la instrucció 8 i la 7 amb el registre R1.

Solució a l'exemple

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
ADD R1,R2,R3	F	D	L	X	X	W																	
SUB R2,R3,R2	F	D	L		X	X	W																
MULT R3,R2,R1		F	D					L	X	X	X	X	W										
ADD R1,R0,R4		F					D	L	X	X	W												
SUB R3,R3,R5							F							D	L	X	X	W					
MULT R0,R4,R2							F							D	L	X	X	X	X	W			
SUB R1,R1,R5														F	D	L	X	X	W				
ADD R2,R3,R1															F	D				L	X	X	W

Estacions de reserva i banc de registres el cicle:5

NOM	OCUP	OPER	FI	FJ	QJ	FK	QK	RJ	RK
+1	1	+	R1	R2		R3		SI	SI
+2	2	-	R2	R3		R2		SI	SI
+3									
*1	3	*	R3	R2	(+2)	R1	(+1)	NO	NO
*2									
SEN1									
SEN2									

	R1	R2	R3	R4	R5	R6	R0
Vi							
Qi	(+1)	(+2)	(*1)				

Estacions de reserva i banc de registres els cicle:12

NOM	OCUP	OPER	FI	FJ	QJ	FK	QK	RJ	RK
+1									
+2	4	+	R1	R0		R4		SI	SI
+3									
*1	3	*	R3	R2		R1		SI	SI
*2									
SEN1									
SEN2									

	R1	R2	R3	R4	R5	R6	R0
Vi		(+2)					
Qi	(+1)		(*1)				

Estacions de reserves i banc de registres els cicles:19

NOM	OCUP	OPER	FI	FJ	QJ	FK	QK	RJ	RK
+1	5	-	R3	R3		R5		SI	SI
+2	7	-	R1	R1		R5		SI	SI
+3	8	+	R2	R3		R1	(+2)	SI	NO
*1	6	*	R0	R4		R2		SI	SI
*2									
SEN1									
SEN2									

	R1	R2	R3	R4	R5	R6	R0
Vi							
Qi	(+2)	(+3)	(+1)				(*1)

Estacions de reserva i banc de registres els cicles:24

NOM	OCUP	OPER	FI	FJ	QJ	FK	QK	RJ	RK
+1									
+2									
+3	8	+	R2	R3		R1		SI	SI
*1									
*2									
SEN1									
SEN2									

	R1	R2	R3	R4	R5	R6	R0
Vi							
Qi		(+3)					

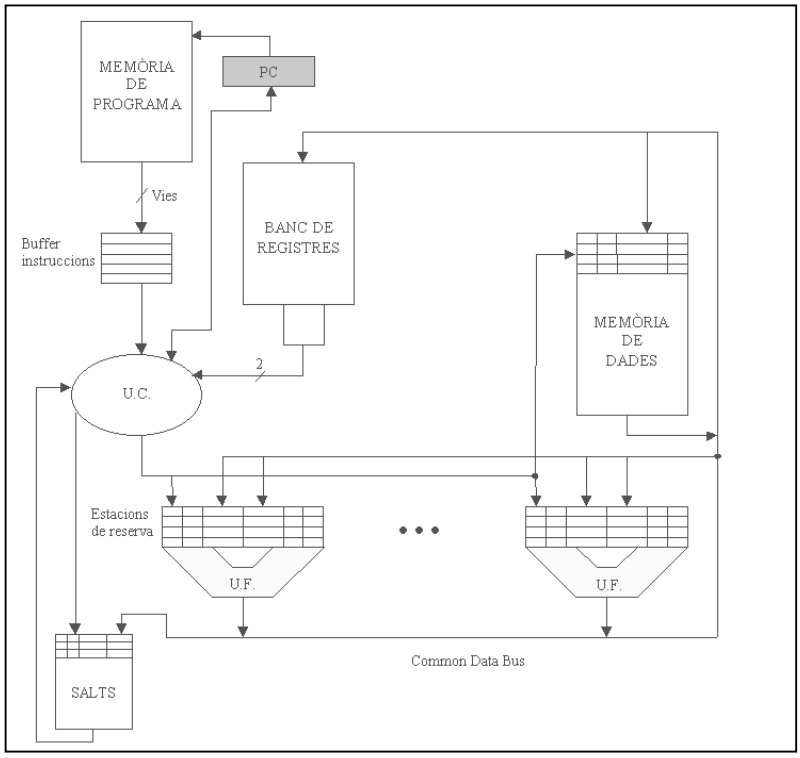
3.2. TOMASULO.

L'algorisme de Tomasulo també duu a terme una planificació dinàmica que permet que les instruccions s'iniciïn i finalitzin en desordre.

La diferència més important d'aquest mètode amb el de Scoreboard és que la detecció de conflictes i el control d'execució estan distribuïts en cadascuna de les unitats funcionals.

Cada unitat funcional té un nombre determinat d'*estacions de reserva* que controlen el moment en que pot començar l'execució d'una instrucció en la unitat funcional. En la tècnica de Scoreboard aquesta funcionalitat està centralitzada en el propi marcador. Les *estacions de reserva* són com petites memòries que té cada unitat funcional que permeten reservar aquesta unitat per a diferents instruccions que l'han d'utilitzar. La primera instrucció que disposi dels operants, serà la primera que s'executi en la unitat funcional. Per tant, el conjunt d'*estacions de reserva* d'una unitat funcional són com una llista d'espera on s'inscriuen totes les instruccions que s'han d'executar en aquella determinada unitat. El que passa, és que l'ordre amb que s'atenen les peticions no és, necessàriament, amb el que s'han fet les inscripcions.

L'esquema de l'arquitectura que implementa aquests algorisme és el següent:



Una altra novetat important és que disposem d'un bus comú de dades CDB (common data bus) que permet a les unitats que esperaven resultats agafar-los directament del bus sense esperar a que es trobin en el banc de registres. Aquesta característica farà que comparat amb l'Scoreboard obtenim un guany d'un cicle, és a dir, si una instrucció està a l'espera del resultat d'un registre d'una instrucció anterior quan aquesta instrucció anterior escrigui el seu valor el podrà llegir directament del bus sense haver d'espera que sigui escrit en el banc de registres.

Per exemple unes instruccions del tipus:

MULT S1,S2,S3
MULT S5,S1,S3

L'execució amb l'Scoreboard seria la següent:

6	7	8	9	10	11
X	X	W			
			L	X	X

La mateixa amb el Tomasulo seria:

6	7	8	9	10	11
X	X	W			
			X	X	W

En el cas que el banc de registres només tingui un port d'escriptura, en el CDB serà un bus simple i només hi podrà escriure una unitat funcional en un mateix cicle. Per contra, si el banc de registres té un port d'escriptura per cada unitat funcional, el CDB serà un bus múltiple permetent que, a cada cicle, hi puguin escriure totes les unitats funcionals.

També cal esmentar que a diferència de l'Scoreboard no se espera a llegir els operants fins que tots ells estiguin disponibles, sinó que, tan aviat es disposin d'ells s'aniran llegint.

Vegem quina informació hem de guardar per cadascun dels següents elements:

- **Per cada estació de reserva:**

Guardarem set camps diferents:

- *Identificador*: número que identifica a l'estació de reserva de forma unívoca entre totes les estacions de reserva que hi hagi en totes les unitats funcionals.
- *Operació*: indica quina operació haurà de realitzar la unitat funcional amb els operants que conté aquesta estació de reserva.
- *Qj*, *Qk*: indiquen quines estacions de reserva produiran els operants font. Si valen 0, llavors és que els operants font ja estan carregats a Vj i Vk.
- *Vj*, *Vk*: contenen el valor dels operants font
- *Ocupada*: indica que l'estació de reserva està ocupada amb informació d'alguna instrucció.

- **Per cada registre del Banc de registres:**

Tindrem un camp *Qi* (i és el número del registre assosciat) que serà el número d'estació de reserva que ha de generar-ne el nou valor. Si val 0 llavors és que ninguna unitat funcional està generant un resultat que s'hagi de gravar en aquest registre.

En el model de Tomasulo, només es necessiten tres etapes per cada instrucció: *emissió*, *execució* i *escriptura*. Vegem què és el que s'hi realitza en cadascuna d'elles:

- **Emissió:**

Es busca una unitat funcional que pugui executar la nova instrucció que tingui alguna estació de reserva lliure. En cas de no trobar-ne cap, s'atura l'emissió d'aquesta instrucció i de les següents fins que es lliberi una estació de reserva.

Un cop es troba una estació de reserva lliure, es marca com a ocupada i s'emplenen els diferents camps: si els operants estan en el banc de registres es llegeixen i es posen en *Vj* i *Vk*. Si algun d'ells no està disponible en els registres, ja que aquests estan esperant ser escrits per alguna unitat funcional (ho veurem en el camp *Qi* que hem afegit a cada registre), es posa la marca de

l'estació de reserva que generarà el nou valor en els camps *Qj* i/o *Qk*; en el camp *Operació* caldrà especificar el tipus d'operació que s'haurà de fer.

L'últim pas consisteix en marcar el camp *Qi* del registre destí de la instrucció (si ha d'escriure sobre algun registre) amb el número d'estació de reserva que s'ha reservat i assosciat a la instrucció.

- **Execució:**

  En aquesta etapa es busca, per cada unitat funcional, una estació de reserva que disposi de tots els operants i s'inicia l'execució de l'operació indicada per la pròpia estació de reserva. Per tant, és aquí on es detecten els conflictes RAW i no en l'etapa d'emissió com era en el cas de Scoreboard. En el cas de la tècnica del marcador un conflicte RAW provocava que l'emissió d'instruccions s'aturés. En el cas de Tomasulo, com que la detecció de RAW es fa en l'etapa d'execució, no hi ha aturades d'emissió i instruccions posteriors poden començar a executar-se.

- **Escriptura:**

  Quan una unitat funcional genera un resultat l'escriu en el bus comú de dades (CDB) que el durà a tots els registres i estacions de reserva on tinguin una marca (Qi en el cas dels registres i Qj i Qk en el cas de les estacions) que indiqui que estaven esperant aquest resultat. Un cop acabada l'escriptura s'allibera l'estació de reserva, pertanyent a la unitat funcional que ha generat el valor, de la qual s'havien agafat els operants.

  Com es pot veure el CDB s'utilitza per difondre resultats permeten a les instruccions poder iniciar l'execució sense esperar a que els operants estiguin en el banc de registres (aquest era un dels principals defectes de l'algorisme de Scoreboard).

La tècnica de Tomasulo elimina els conflictes WAW i WAR gràcies a les estacions de reserva i a les marques que indiquen qui generarà el nou valor d'un registre o d'un operant. Per exemple, si tenim les dos instruccions:

R2 ← R1 op R5
R2 ← R3 op R4

No ens ha de preocupar el fet que la segona pugui acabar abans que la primera i al final de l'execució R2 no contingui el resultat correcte. Això no passarà ja que, quan s'emet la primera, es marca el camp Q_2 del banc de registres amb el número d'estació de reserva on s'ha introduït la instrucció. Però quan s'emet la segona instrucció, es sobreescriu el camp Q_2 posant el número d'estació corresponent a aquesta nova instrucció. El fet de posar aquesta marca suposa que només el resultat generat per l'estació de reserva indicada podrà ser escrit en R2. Per tant, si la última en acabar és la primera instrucció, aquesta no escriurà sobre R2 ja que Q_2 no durà la marca de l'estació de reserva associada.

Per exemple si agafem el mateix codi que amb el Scoreboard:

```
1    add   r1, r2, r3    ;
2    sub   r2, r3, r2    ;
3    mul   r3, r2, r1    ;
4    add   r1, r0, r4    ;
5    sub   r3, r3, r5    ;
6    mul   r0, r4, r2    ;
7    sub   r1, r1, r5    ;
8    mul   r2, r3, r1    ;
```

Suposa un superescalar de 2 vies amb 2 Unitats Funcionals amb les següents característiques:

UF	Estacions de reserva	Operacions	Latència de finalització	Latència d'inici
ALU1	3	+, -	2	1
ALU2	2	*	4	2

Si tenim Tomasulo ens quedarà el processador després d'executar cadascun dels cicles indicats tindrem els següents estats (suposem les etapes: F, D, X, W).

Com es pot observar el la resolució de l'exercici que es troba en la pàgina següent es donen la següent sèrie de conflictes:

- En el cicle 3 es produeix un WAW entre la instrucció 1 i 4 en el registre R1 que degut al funcionament del tomasulo no fa aturar l'execució.

- En el cicle 3 es produeix un conflicte estructural entre la instrucció 1 i la 2 produït per la latència d'entrada de la unitat funcional.

- En el cicle 4 es produeix un conflicte RAW entre els registres R1 de les instruccions 1 i 3 i un altre RAW amb el registre R2 de les instruccions 2 i 3.

- En el cicle 4 es produeix un conflicte estructural entre la instrucció 2 i la 4 produït per la latència d'entrada de la unitat funcional.

- En el cicle 6 es produeix un conflicte estructural per manca d'estacions de reserva per decodificar la instrucció 5 que para la decodificació fins que aquest es soluciona.

- En el cicle 7 es produeix un conflicte RAW entre la instrucció 5 i la 3 pel registre R3.

- En el cicle es produeix un conflicte estructural entre la instrucció 3 i la 6 produït per la latència d'entrada de la unitat funcional.

- En el cicle 8 es produeix un conflicte RAW entre la instrucció 5 i la 8 pel registre R3

➢

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
ADD R1,R2,R3	F	D	X	X	W																		
SUB R2,R3,R2	F	D		X	X	W																	
MULT R3,R2,R1		F	D			X	X	X	X	W													
ADD R1,R0,R4		F	D		X	X	W																
SUB R3,R3,R5			F			D				X	X	W											
MULT R0,R4,R2			F			D		X	X	X	X	W											
SUB R1,R1,R5						F	D	X	X	W													
ADD R2,R3,R1						F		D				X	X	W									

Estacions de reserva i banc de registres el cicle:2

NOM	OCUPAT	OPER	VJ	QJ	VK	QK
+1	1	+	R2		R3	
+2	2	-	R3		R2	
+3						
*1						
*2						
SEN1						
SEN2						

	R1	R2	R3	R4	R5	R0
Vi						
Qi	(+1)	(+2)				

Estacions de reserves i banc de registres els cicles:9

NOM	OCUPAT	OPER	VJ	QJ	VK	QK
+1	5	-	R3	(*1)	R5	
+2	7	-	R1		R5	
+3	8	+	R3	(+1)	R1	(+2)
*1	3	*	R2		R1	
*2	6	*	R4		R2	
SEN1						
SEN2						

	R1	R2	R3	R4	R5	R0
Vi						
Qi	(+2)	(+3)	(+1)			(*2)

Estacions de reserva i banc de registres els cicle:6

NOM	OCUPAT	OPER	VJ	QJ	VK	QK
+1						
+2	2	-	R3		R2	
+3	4	+	R0		R4	
*1	3	*	R2	(+2)	R1	
*2						
SEN1						
SEN2						

	S1	S2	S3	S4	S5	S6
Vi						
Qi	(+3)	(+2)	(*1)			

Estacions de reserva i banc de registres els cicles:14

NOM	OCUPAT	OPER	VJ	QJ	VK	QK
+1						
+2						
+3	8	+	R3		R1	
*1						
*2						
SEN1						
SEN2						

	R1	R2	R3	R4	R5	R0
Vi						
Qi		(+3)				

4. TRACTAMENT DE LES INTERRUPCIONS I LES RUPTURES DE SEQÜÈNCIA.

Com ja passava el els processadors segmentats i supersegmentats un dels principals problemes que ens podem trobar és el fet del trencament de la seqüència de les instruccions. Aquest pot ser produït per una interrupció o per un salt.

En el cas dels superescalar donat que el nombre d'instruccions que s'executen alhora es força elevat el fet de parar l'execució de les instruccions fins saber quina serà la següent instrucció perjudicaria de forma important el throughput del sistema (pèrdues de rendiment molt importants), aquesta arquitectura opta per l'execució especulativa.

Les instruccions de salt poden provocar. Com hem vist, amb l'arquitectura segmentada i superescalar, s'intenta aconseguir un nombre de cicles mig per instrucció de 1. A causa dels salts, aquest CPI mitjà pot augmentar considerablement i arribar a doblar-se.

La causa d'aquest decrement en el rendiment és la pròpia naturalesa del salt: pot significar un canvi en la seqüència d'execució del programa, el que fa que dubtem de què fer mentre s'executa el salt.

En una instrucció de salt s'han d'obtenir dos resultats: l'adreça destí del salt i l'avaluació de la condició.

En un principi, l'adreça destí del salt es calcula, com a molt aviat, en l'etapa d'emissió o descodificació. Mentre que l'avaluació de la condició, per saber si el salt s'ha de realitzar (el que vol dir modificar el Program Counter amb PC=PC+x) o no, s'efectua en l'etapa d'execució on intervenen les unitats funcionals.

El problema està en decidir què fer mentre el salt s'està executant. La solució més senzilla és aturar l'emissió d'instruccions fins que els salt finalitzi i s'hagi modificat el PC en cas de confirmar-se. Obviament, aquesta solució és la que provoca la baixada de rendiment més gran, ja que estem els cicles que dura l'execució del salt sense poder tractar altres instruccions.

El fet de treballar amb una execució especulativa (suposar per un continuarà l'execució de les instruccions) això repercutirà en el fet de dotar d'una

sèrie d'elements que ens permetin tirar cap enrera totes les instruccions que s'han executat i no deurien haver-se executat.

4.1 TÈCNIQUES DE CONTROL D'EXECUCIÓ ESPECULATIVA

4.1.1 *INTRODUCCIÓ TEÒRICA*

En l'apartat de predicció de salts, ja hem explicat que, un cop es fa una predicció i s'executen unes instruccions determinades, fins al moment en que es pugui avaluar la condició del salt, l'execució serà especulativa. No podem estar segurs que allò que hem decidit executar sigui el que realment tocava. Si a l'avaluar la condició del salt, ens adonem que hem executat un tros de programa que no s'havia de fer, sorgeix el problema de desfer totes aquestes instruccions executades especulativament i retornar la màquina a un estat estable immediatament posterior al salt. Per tant, haurem de trobar algun mecanisme per recuperar un estat anterior de la màquina i reengegar-la en aquell moment.

Aquest problema també rep el nom de interrupcions precises, ja que en el cas d'una interrupció també es produeix un canvi de seqüència d'execució (passem d'executar un codi de programa a un altre remot) i per tant, un cop s'ha servit la interrupció, s'ha de continuar l'execució en el punt on s'havia deixat. No cal dir que el cas del tractament d'excepcions també és el mateix. En aquests dos casos la causa principal del problema és que les instruccions poden finalitzar en desordre i en el moment en que arriba la interrupció o excepció, podem tenir instruccions finalitzades enmig d'instruccions no acabades.

Abans de donar possibles solucions, definirem en quins estats es pot trobar el processador. Entendrem com a estat la situació en que es troba el banc de registres en un moment donat.

- **Estat en ordre**

  Estat dels registres en el punt immediatament anterior a la primera instrucció no finalitzada.

  Per tant, en aquest estat, el banc de registres conté els valors assignats més recentment produïts per la seqüència d'instruccions finalitzades consecutives més llarga possible.

És un estat segur, cap instrucció que l'ha creat pot ser desfeta.

- **Estat arquitectònic**

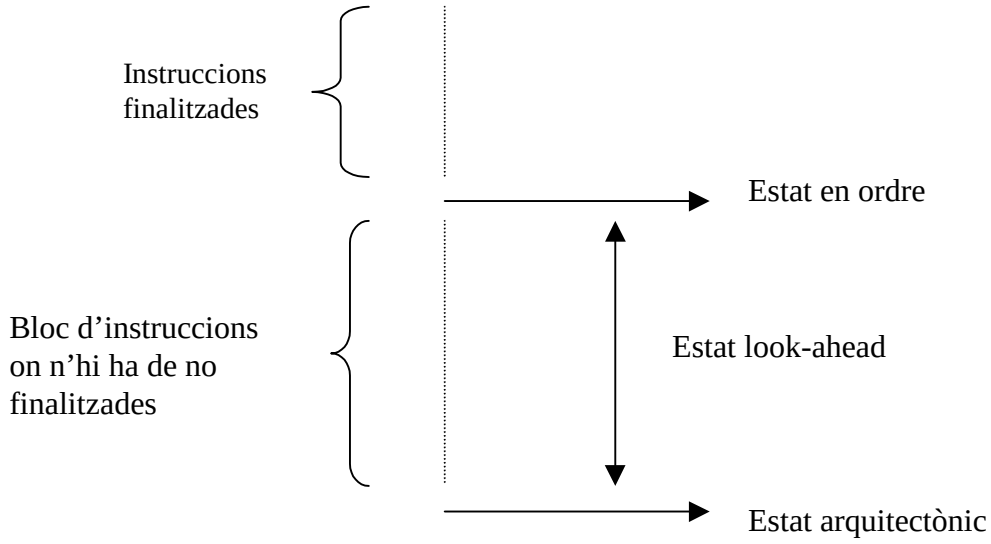
  Aquest estat consisteix en les assignacions a registres més recents, sense importar si ja s'han fet o encara estan pendents de fer-se. Aquestes assignacions són fruit d'instruccions completades o només iniciades i, per tant, són provocades per una seqüència d'instruccions consecutives en que n'hi ha alguna, com a mínim, de no finalitzada.

  L'estat arquitectònic en un cicle determinat és l'estat de la màquina tal com seria un cop finalitzés l'última instrucció iniciada fins a aquest cicle.

- **Estat look-ahead**

  L'estat look-ahead està format pels valors que estan pendents de ser generats per instruccions no finalitzades i els valors que han generat instruccions finalitzades però que no entren dins l'estat en ordre. És per tant, el conjunt d'assignacions que s'han de realitzar per fer arribar l'estat en ordre fins al punt que marca l'estat arquitectònic en aquest instant de temps.

En el següent esquema es pot veure d'una forma més clara aquests tres conceptes aplicats a un codi de programa:

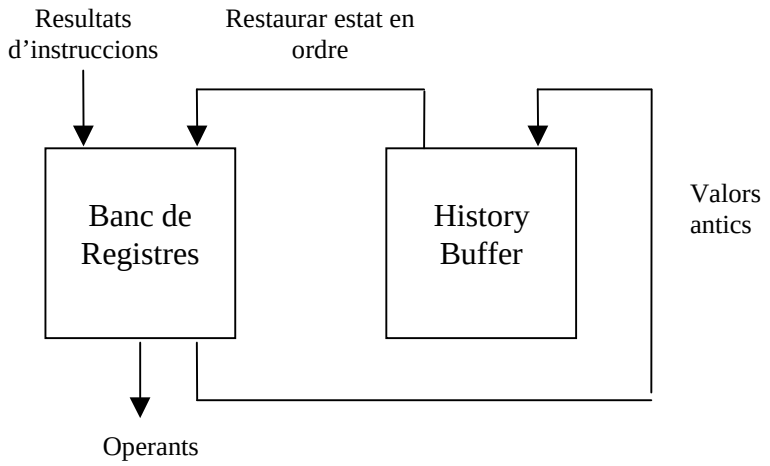


Tot seguit veurem tres tècniques que s'han proposat a partir del 1985 i que permeten restaurar l'estat en ordre en la màquina en quansevol moment, són el ***History Buffer***, el ***Reorder Buffer*** i el ***Future File***.

Cal dir que aquestes tècniques estan pensades per a restaurar l'estat en ordre en la màquina un cop es produeix una interrupció, una excepció o quan la predicció d'un salt ha estat errònia i s'han d'anul·lar les modificacions provocades per instruccions executades especulativament. Al fer l'estudi de les tècniques que veurem, ens basarem en aquest últim problema, ja que és l'únic que contemplarà el simulador. Cal dir, però, que el funcionament en els dos restants casos és pràcticament idèntic.

4.1.2 HISTORY BUFFER

L'organització d'un *history buffer* ve donat pel següent esquema:



El *history buffer* funciona com una pila (LIFO). Quan es descodifica una instrucció, es guarda en el *top* del *history buffer* una entrada amb el valor actual del registre sobre el que escriurà la instrucció (també es guarda altra informació com el PC de la instrucció). Si es dóna el cas que no hi ha espai en el *history buffer*, s'atura l'emissió d'instruccions.

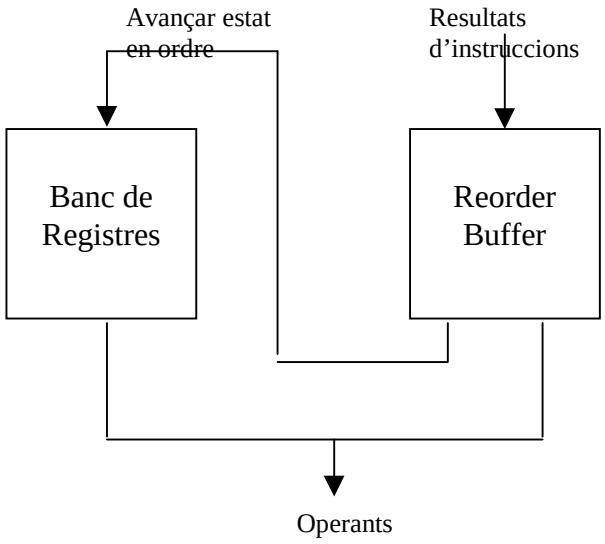
Cada cop es mira si l'entrada que es troba en el *bottom* del *history buffer* es corresponen amb una instrucció finalitzada, en aquest cas s'elimina aquesta entrada.

En cas que s'hagin d'eliminar les modificacions provocades per instruccions executades especulativament a causa d'un salt en el que la predicció no ha estat correcte, s'atura l'emissió d'insruccions i s'espera a que finalitzin les instruccions inacabades anteriors a la que ha provocat l'errada. Un cop finalitzades, a cada cicle de rellotge, es llegeix un valor del *top* del *history buffer* i es restaura en el banc de registres, fins a trobar l'entrada corresponent al salt. D'aquesta manera, estem restablint l'estat en ordre just abans del salt i tot seguit es podrà continuar l'execució a partir de l'adreça destí que ja haurà estat carregada en el PC. A l'extreure la informació del *history buffer* en forma de pila, ens assegurem que al final, en el banc de registres, quedin els valors més antics i, per tant, els correctes.

El history buffer té dos problemes. El primer és que es necessiten ports de lectura adicionals en el banc de registres per poder passar els valors actuals cap al history buffer. Per cada via de descodificació del processador superescalar es necessitarà un port de lectura per tots els registres. El segon problema és de rendiment. Cal pensar que, per restaurar l'estat en ordre es necessiten un nombre considerable de cicles (un per cada valor que s'hagi de restaurar en el banc de registres). En el cas de les excepcions no és gaire greu ja que no es produeixen gaire sovint. Però si parlem de salts en els que la predicció ha fallat, ens apareix un nombre excessiu de cicles necessaris.

4.1.3 REORDER BUFFER

El *reorder buffer* és una nova estructura que intenta millorar el rendiment del *history buffer*. En el següent esquema podem veure l'organització d'aquest nou mètode:



En aquest cas, el banc de registres conté l'estat en ordre i el *reorder buffer* conté l'estat look-ahead. L'estat arquitectònic no es troba explícitament enlloc, en realitat, s'obté combinant els dos anteriors.

El *reorder buffer* actua com una cua (FIFO). Quan es descodifica una instrucció, s'emplena una entrada en el *top* del *reorder buffer*. Cada entrada conté un camp on serà escrit el resultat de la instrucció assosciada un cop aquesta finalitzi. A cada cicle, es mira si en el *bottom* del *reorder buffer* hi ha una entrada amb el resultat ja escrit. De ser així, aquest resultat es grava en el banc de registres, actualitzant l'estat en ordre, i s'elimina l'entrada.

En el cas que no quedi espai disponible en el *reorder buffer*, l'emissió d'instruccions s'atura fins que n'hi hagi.

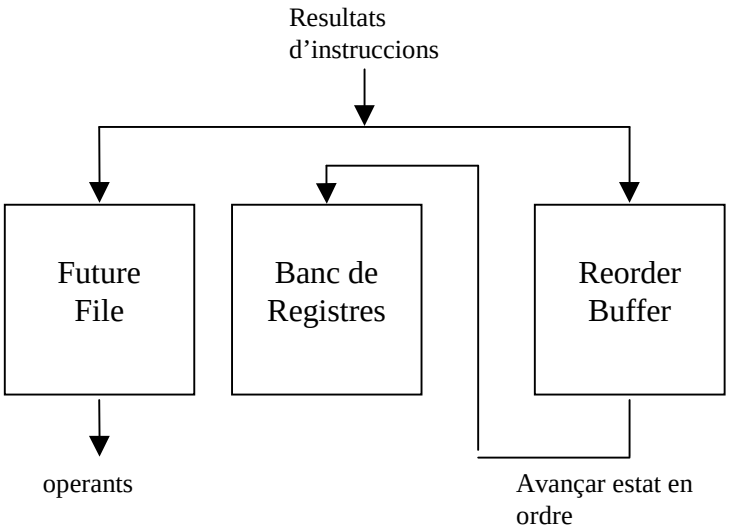
Quan una instrucció necessita un operant, el valor d'aquest es trobarà o bé en alguna entrada del reorder buffer o bé en el banc de registres. La lògica de control del reorder buffer ha de ser capaç de trobar el valor més nou del registre desitjat.

Si es produeix un error de predicció en un salt, per restablir l'estat en ordre just abans del salt, caldrà esperar a que acabin d'executar-se les instruccions anteriors al salt. Un cop hagin finalitzat es descarta el contingut del *reorder buffer* ja que en el banc de registres ja tindrem l'estat en ordre desitjat. No és gaire normal que un cop finalitzat un salt quedin instruccions anteriors per finalitzar, ja que el salt sol dependre del resultat d'aquestes. Per tant, normalment, per restablir l'estat en ordre en el punt desitjat, només caldrà buidar el *reorder buffer*.

El defecte del reorder buffer és la costosa implementació, ja que requereix memòries assosciatives prioritzades per buscar el valor més nou d'un registre.

4.1.4 - FUTURE FILE

Aquesta tècnica és una ampliació del reorder buffer i consisteix en afegir una nova estructura que funciona com un banc de registres, que contindrà l'estat arquitectònic:



El funcionament és exactament igual que en el *reorder buffer*, però quan es necessita obtenir un operant, només cal anar-lo a llegir directament al *future file*.

Quan una instrucció finalitza i ha d'escriure el seu resultat en algun registre, aquest nou valor es guarda en l'entrada corresponent del *reorder buffer* i també en el *future file*. El banc de registres només rebrà aquest valor quan l'entrada del *reorder buffer* arribi al bottom de la cua i s'elimini (tal com havíem vist en el punt anterior).

Quan es detecta un error de predicció d'un salt, caldrà esperar a que finalitzin les instruccions anteriors al salt i tot seguit, copiar el contingut del banc de registres (que conté l'estat en ordre) al future file i podrà continuar l'execució a partir de l'adreça correcta.

L'únic defecte d'aquest muntatge pot ser el cost en cicles de rellotge de volcar el contingut del banc de registres cap al future file. Nosaltres despreciarem aquest cost i considerarem que es pot fer en un sol cicle. Existeixen altres implementacions del future file que permeten que els operants es vagin a buscar tan en el propi future file com en el banc de registres (si no s'ha trobat el valor més nou en el primer). Així, un cop hem tractat un error de predicció de salt, es defineix que el banc de registres conté els valors més nous i no cal fer el volcat cap al future file.

5 TÈCNIQUES D'OPTIMITZACIÓ.

Amb les tècniques d'optimització intentem obtenir un millor rendiment de les execucions dels programes en les màquines superescalars.

Les pèrdues de rendiments que ens podem trobar amb aquests sistemes es troben fonamentalment en els bloqueigs de les instruccions provocats o per la manca d'unitats funcionals per a poder-les executar (conflictes estructurals) o bé per els bloqueigs provocats pels problemes de precedències entre elles (conflictes de dades). Per obtenir aquestes millores existeixen una sèrie de mètodes que es basaran en l'intent de reordenació de les instruccions i/o de registres per obtenir un major troughput del sistema i per tant un major rendiment de les unitats funcionals i la assignació correcta de registres.

5.1 ASSIGNACIÓ DE REGISTRES.

Un dels problemes que ens trobem dins el conflictes estructurals és el fet de la dolenta assignació de registres això farà que es produeixin conflictes alhora de la utilització dels mateixos. Per exemple:

```
1       ADD r1,r2,r3
2       SUB r2,r1,r3
3       MULT r1,r4,r5
```

Com es pot observar la instrucció 1 i la instrucció 3 utilitzen el registre "r1". Amb aquesta assignació es produirà un conflicte de WAW entre les dues instruccions. Aquest conflicte no s'hagués produït si la instrucció r1 en lloc d'utilitzar el registre "r1" utilitzés un altre registre. Normalment una correcta assignació de registre faria que conflicte estructural no es produís.

De la mateixa manera poden haver-hi moltes instruccions que pel no correcte reutilització de registres (sense que es produeixin conflictes estructurals) es fagin servir més registres del compte i això farà que quan es vulgui assignar un registre per una instrucció aquests estigui ocupat.

5.2 LIST SCHEDULLING

El software schedulling intenta ordenar totes les instruccions per així poder optimitzar l'execució en el hardware.

Imaginem que tenim el seguent programa i la seva execució:

	1	2	3	4	5	6	7	8
1: MULT S1,S2,S3	F	D	L	X	X	X	X	W
2: ADD S5,S1,S3	F	D						L
3: CMP S6,S1,0		F	D					L
4: BNE S6, LL1		F	D					
5: ADD S7,S2,S3			F	D	L	X	X	W
6: ADD S3,S4,S5			F	D				
7: MULT S2,S4,S2				F	D	L	X	X

Com es pot observar en l'execució existeix un bloqueig provocat per la instrucció 1 sobre les instruccions 2,3 i 4 i també podem veure com la instrucció 5 i 7 es poden executar sense cap problema donat a que cap instrucció els hi bloqueja.

Si modifiquéssim l'ordenació de les instruccions veuríem que el resultat millora considerablement tant a nivell de troughput (en el cicle 8 tenim tres instruccions que han acabat en l'execució ordenada per 2 en la no ordenada) com de rendiment de les unitats funcionals:

	1	2	3	4	5	6	7	8
1: MULT S1,S2,S3	F	D	L	X	X	X	X	W
5: ADD S7,S2,S3	F	D	L	X	X	W		
7: MULT S2,S4,S2		F	D	L		X	X	W
2: ADD S5,S1,S3		F	D					L
3: CMP S6,S1,0			F					L
4: BNE S6, LL1			F					
6: ADD S3,S4,S5			F	D				

Per obtenir aquests resultats és molt important que el compilador tingui en compte els possibles conflictes que es poden donar. En les proves realitzades s'ha observat que sense el schedulling es produeix un temps de overall del 75 % mentre que després d'ordenar es redueix un 20% el seu valor.

5.2.1 BLOCS BÀSICS

A l'hora de realitzar l'ordenació del programa és molt important la informació que s'extreu del mateix.

El primer pas que s'hauria de fer és la divisió del programa en blocs bàsics. Entenem per blocs bàsics aquell conjunt d'instruccions que quan s'executa una

posteriorment sempre s'executaran totes les altres que formen en bloc. Per tant en un bloc no existeix cap instrucció de salt.

Degut a que totes les instruccions s'executen juntes es coneixen totes les dependències que existeixen entre elles i per tant es pot construir un graf de control de flux a on es posi la relació que existeixen entre elles.

Per exemple tenim el següent codi:

```
1   mov r20 var_1
2   mov r21 var_2
3   load r0,(r20)0
4   load r1,(r21)0
5   sub r2,r0,1
6   mult r3,r1,2
7   cmp r4,r2,r3
8   bne r4,LL3
9   mov r22 var_3
10  load r5, (r22)0
11  sub r6,r5,1
12  add r7,r0,1
13  cmp r8,r6,r7
14  beq r8,LL3
15  cmp r9,r3,r6
16  bne r9,LL6
17  add r10,r5,r0
18  mult r11,r1,r0
19  mult r12,r10,r11
20  store r12, (r22)0
21  b LL7
22  LL6: sub r13,r5,r0
23  div r14,r1,r0
24  add r15,r13,r14
25  store r15, (r20)0
26  b LL7
27  LL3: sub r16,r0,r1
28  store r16,(r21),0
29  LL7:
```

Aquest codi es pot expressar en pseudocodi amb la següent estructura:

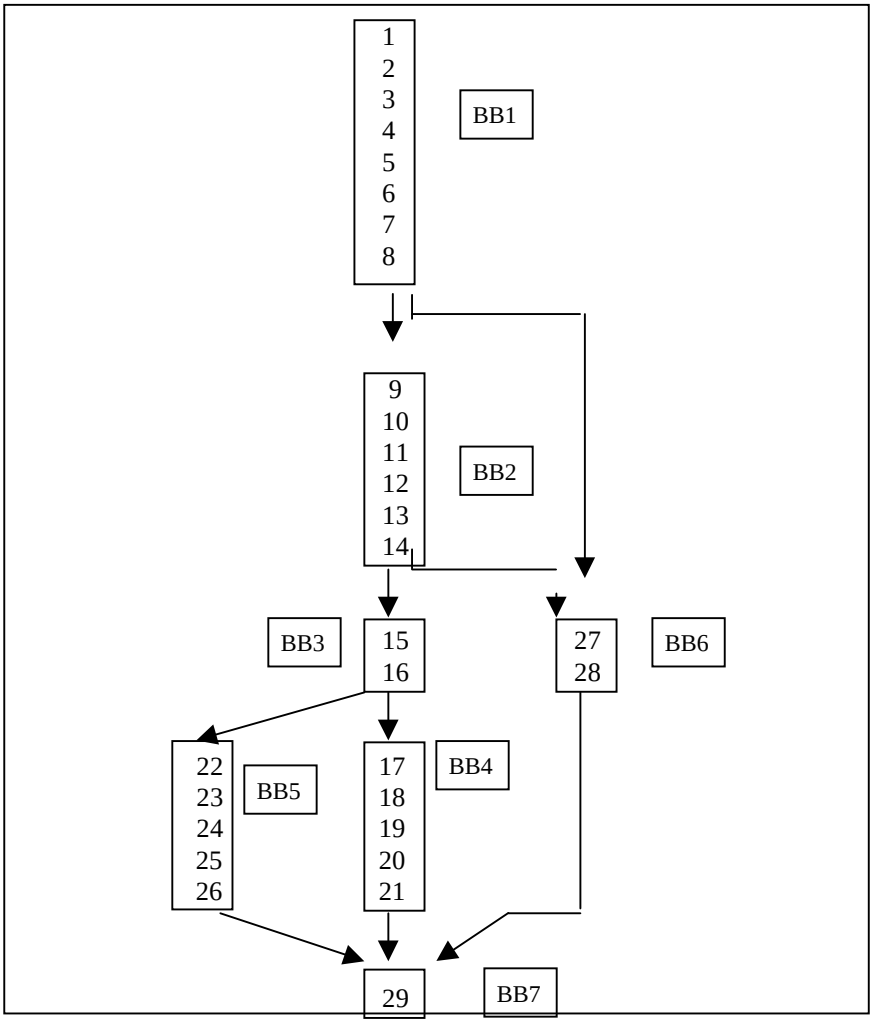
```
IF (A-1==B*2)&&(C-1!=A+1)
  {
  IF(B*2=C-1)
     {      F=C+A;
            G=B*A;
            H=F*G;
     }
  ELSE
     {      I=C-A;
            J=B/A;
            K=I+J;
     }
  }
ELSE
  L=A-B;
```

Un bloc bàsic es construeix de la següent manera:

- La primera instrucció d'un bloc serà la primera instrucció del programa o la primera instrucció després d'un salt o aquella instrucció a on marca el salt.

- Com a última instrucció la instrucció abans de salt o el final del mateix.

Una vegada s'han dividit en blocs bàsics es pot realitzar el graf de flux de dades de cada bloc bàsic. En aquest graf presentarem les dependències veritables que existeixen que han de satisfer pel schedulle de les instruccions. Estarà format per depedències acícliques (no existeixe bucles en el bloc bàsic) a on els nodes representen les instruccions i els arcs les depedències entre elles:



També si volguéssim podríem generar el graf tenint en compte les dependències del tipus anti i output dependències (WAR i WAW) que es podrien donar per a cada registre, però això ens faria complicar el graf donat que les

dependències es poden modificar de forma considerable. Aquest tipus de dependències no es podran amb la reordenació software solucionar degut a la dificultat que això planteja, si bé hi ha mètodes com la reasignació de registres que ajudaran a evitar que això no es produeixi.

5.2.2 EL CAMÍ CRÍTIC.

Aquest mètode consisteix en determinar el camí que fa que determini el temps d'execució mínim de la seqüència sencera de instruccions que no es poden executar de forma concurrent (temps més petit d'execució possible). També és conegut com a EST.

Els passos a seguir en l'algorisme per a la construcció del camí seria el següent:

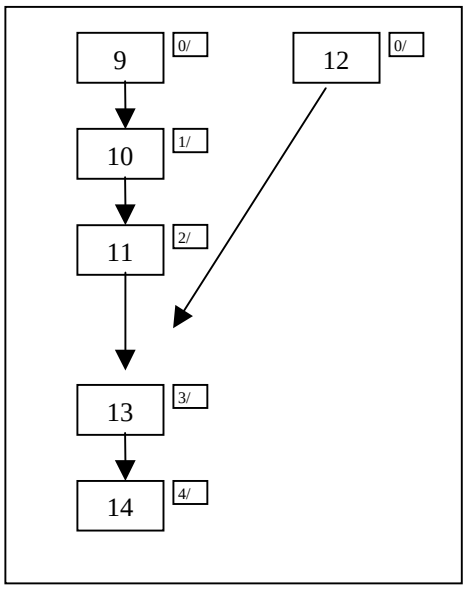
1) Agafar un bloc bàsic i en els nodes de l'arrel assignar-li 0 al valor de l'EST.
2) Baixar un nivell i en cada fulla assignar-li com a EST el valor màxim del resultant de sumar-li a l'EST de cada pare el temps d'execució del mateix.
3) Anar fent la operació 2 fins a arribar al final del graf.

El camí crític serà aquell que partint del node arrel va baixant fins el darrer nivell passant en cada nivell pel node en que el seu EST és màxim.

Utilitzant el següent bloc bàsic i tenint en compte les següents latències obtenim el següents valors:

UNITATS	NÚMERO	LATÈNCIA ENTRADA	LATÈNCIA SORTIDA	OPERACIONS
De salt	1	1	1	b, ble,beq,bne
Senceres	1	1	2	Add
Multiplicació	1	2	4	Mult
Accés memòria	1	1	1	Load, store
Complements	1	1	1	Mov, cmp

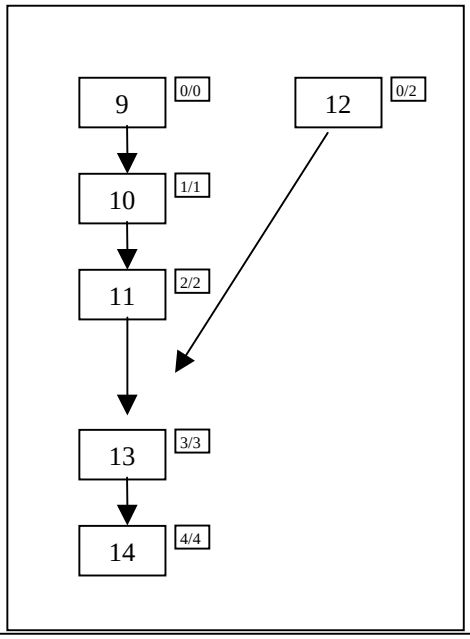
```
9   mov r22 var_3
10 load r5, (r22)0
11 sub r6,r5,1
12 add r7,r0,1
13 cmp r8,r6,r7
14 beq r8,LL3
```



En aquest cas el camí crític serà 9(0)-10(1)-11(2)-13(3)-14(4).

De la mateixa manera podriem obtenir el temps major d'execució de cada instrucció (LST). Per això partim de l'últim node agafant el seu valor màxim de EST i anem pujant a els pares de node i restant-li al valor del fill el valor corresponent a la seva lantència.

En el nostre cas seria el següent:



Com es pot comprovar en el camí crític el valor de l'EST i del LST és el mateix.

5.2.3 REORDENAMENT D'INSTRUCCIONS.

Amb l'ordenació d'instruccions té com a objectiu obtenir una execució de les intruccions correcta amb un temps mínim. Però cal considerar que això comporta un cost elevat de temps per a garantir que s'ha aconseguit una ordenació óptima. La ordenació d'instruccions és fàcil per un processador ideal amb un nombre infinit de recursos perquè el temps d'execució del codi és determinat tan sols per les depedència de les dades.

Una vegada tenim el camí crític calculat s'ha d'escollir quin mètode de reordenació s'escollirà: EST o LST. La diferència entre un i altre es l'ordre en que s'utilitzen les instruccions. Per ambdós mètodes es dibuixa una taula que té tantes columnes com unitats funcionals disposem i tantes files com el valor màxim de l'EST +1 (donat a que el primer valor de l'EST és 0). Les files representen el cicle d'execució. Després d'haver construït la taula escollirem el mètode:

- EST:
  1) Escollirem totes les fulles i les posarem a la darrera fila indicant el valor de l'EST que tenen.

  2) Agafant una per una cada instrucció i tenint prioritat per valor màxim d'EST farem:
     a. Mirar si la unitat funcional que li correspon a la instrucció està lliure o en el cas d'estar ocupada la latència de l'entrada de la mateixa s'ha complert. Si això es compleix introduir la instrucció en la casella corresponent.
     b. Passar a la següent instrucció tenint en compte el valor del seu EST.
     c. Repetir a) i b) fins que la llistat es buidi.

  3) Mentre hi hagin instruccions pujar un cicle (cicle-1) i generar una nova llista amb totes les instruccions que no s'han pogut executar

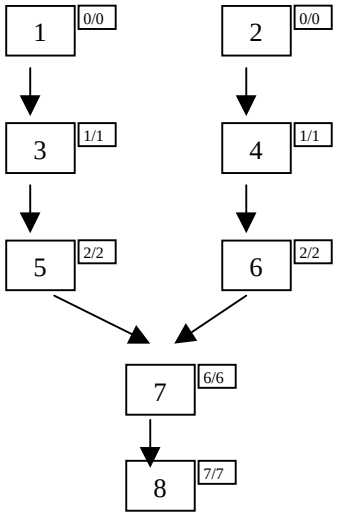
afegint a aquesta llista totes les instruccions que ha quedat com a fulles una vegada s'han eliminat totes les instruccions que s'han executat sempre i quan la latència d'execució de la instrucció respecte al nombre de cicles que han passat amb la instrucció que li precedia (latència de sortida de la instrucció) s'hagi complert. Tornar a executar el pas 2).

Exemple:

Si agafem les instruccions:

```
1    mov r20 var_1
2    mov r21 var_2
3    load r0,(r20)0
4    load r1,(r21)0
5    sub r2,r0,1
6    mult r3,r1,2
7    cmp r4,r2,r3
8    bne r4,LL3
```

Una vegada generat el graf de dependències:



La llista de instruccions respecte el valor de l'EST serà el següent:

Cicle	Llista	Salt	+	*	L/S	Comp
1	1(0)					1
2	3(1),2(0)				3	2
3	4(1),3(1)				4	

4	5(2),6(2)		5	6		
5	7(6)					7
6	8(7)	8				

SOLUCIÓ: 1,3,2,4,5,6,7,8

- LST:
  1) Escollirem totes les arrels i les posarem a la primera fila indicant el valor de l'LST que tenen.

  2) Agafant una per una cada instrucció i tenint prioritat per valor mínim d'LST farem:
     a. Mirar si la unitat funcional que li correspon a la instrucció està lliure o en el cas d'estar ocupada la latència de l'entrada de la mateixa s'ha complert. Si això es compleix introduir la instrucció en la casella corresponent.
     b. Passar a la següent instrucció tenint en compte el valor del seu LST.
     c. Repetir a) i b) fins que la llistat es buidi.

  3) Mentre hi hagin instruccions baixar un cicle (cicle+1) i generar una nova llista amb totes les instruccions que no s'han pogut executar afegint a aquesta llista totes les instruccions que ha quedat com a arrel una vegada s'han eliminat totes les instruccions que s'han executat sempre i quan al nombre de cicles que han passat amb la instrucció que li antecedia (latència de sortida del pare/s) s'hagi complert. Tornar a executar el pas 2).

Exemple:

Agafant l'exemple anterior el resultat serà:

Cicle	Llista	Salt	+	*	L/S	Comp
1	1(0),2(0)					1
2	3(1),2(0)				3	2
3	4(1),5(2)		5		4	
4	6(2)			6		
5	7(6)					7
6	8(7)	8				

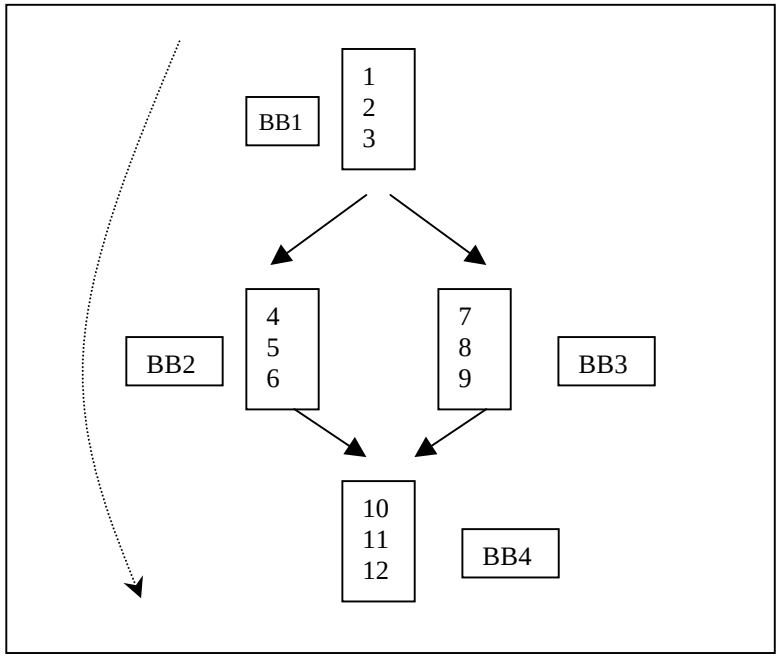
SOLUCIÓ: 1,2,3,4,5,6,7,8

5.3 TRACE SCHEDULLING

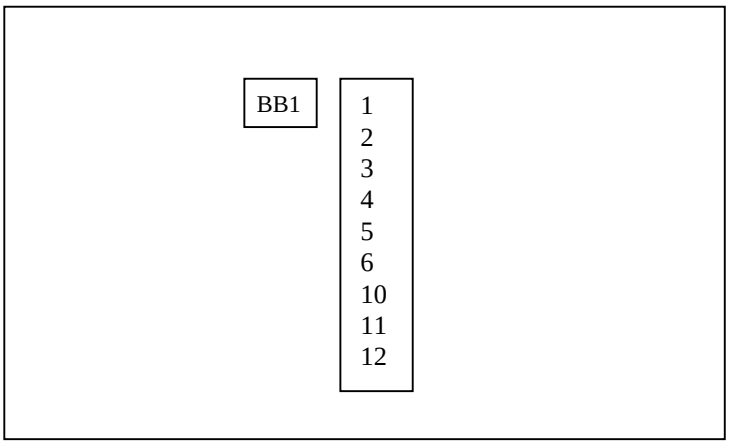
El mètode que em estat veient fins ara intenta optimitzar les instruccions tenint en compte els blocs bàsics existents de forma que quan un programa tinguin un nombre elevat d'instruccions de trencament de seqüència el nombre de blocs bàsics serà elevat. Això comportarà que el nombre d'instruccions que conté cada bloc bàsic serà petita i el guany obtingut amb la reordenació serà més petit.

El mètode del trace schedulling es basa en fer una especulació del camí seguit pel programa una vegada es troba amb una instrucció condicional. Després de haver considerat el camí seguit es crea un bloc bàsic gran que agrupa totes les instruccions que formen el camí d'execució especulat. Una vegada fet això s'haurà de fer un list schedulling d'aquest bloc bàsic.

Per exemple si tinguéssim el següent graf:



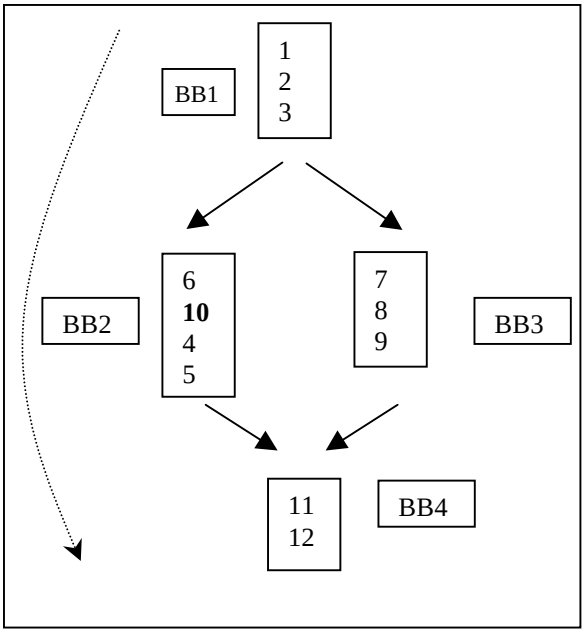
El trace schedulling resultant podria ser el següent:



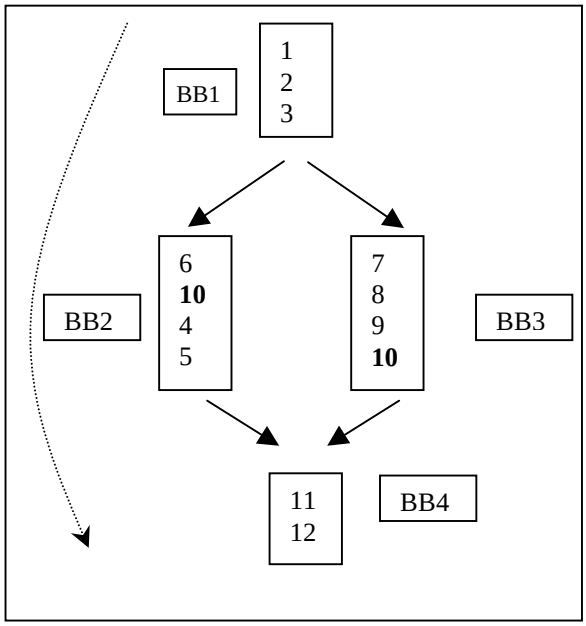
Sobre aquest s'haurà de calcular el List Schedulling i un vegada s'ha realitzat el mateix s'haurà de veure el graf resultant per veure el codi de compensació que s'haurà d'introduir en el cas de que el camí seguit pel programa no sigui el que havíem predit. Per exemple es poden donar el següents casos:

1. Una instrucció de bloc bàsic 4 (la instrucció 10) queda ordenat en el bloc bàsic 2. La solució consistiria en introduir en el bloc bàsic 3 la instrucció 10 per a que també s'executi en el cas de que el camí sigui el no predeït.

RESULTAT:

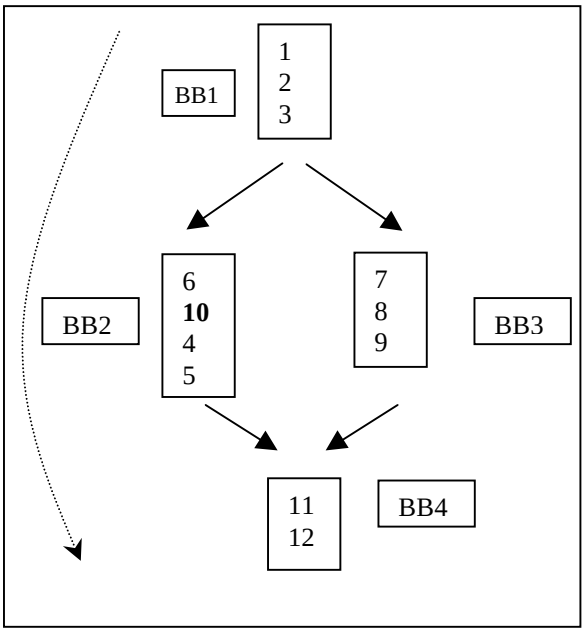


SOLUCIÓ:

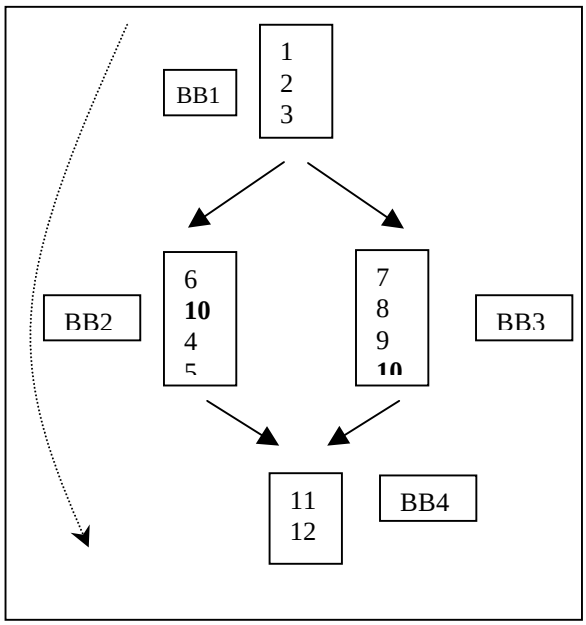


2. Una instrucció del bloc bàsic 2 (la instrucció 6) queda ordenat dins el bloc bàsic 1 i per tant s'executarà tant si el camí és el predeït com si no ho fos amb el que s'executa una instrucció que no s'hauria d'executar. La solució és introduir en el bloc bàsic 3 quelcom per desfer la instrucció 6 executa en el cas de no complir-se la predicció.

RESULTAT:



SOLUCIÓ:



A part d'aquests problemes es poden donar problemes mixtes més complexes a on s'hauran d'estudiar totes les posibles problemàtiques que es puguin anar produint.

Un exemple de trace schedulling seria el següent:

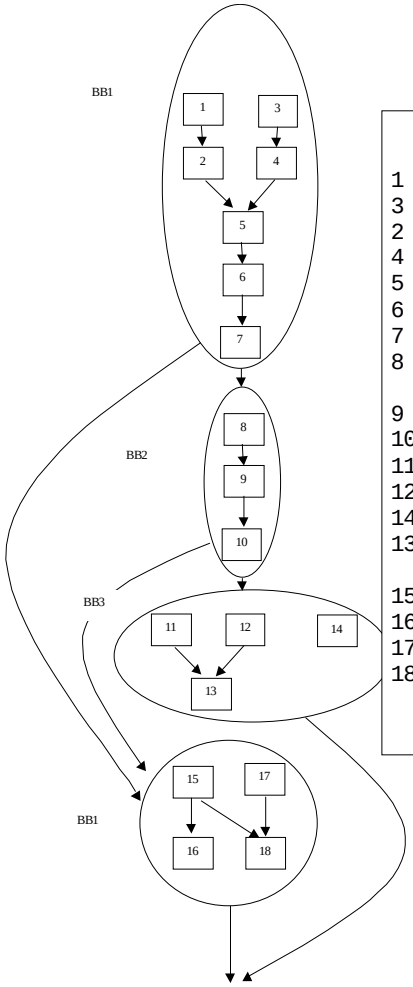
Suposem la següent seqüència de codi que s'executa en un processador superescalar de 4 vies, amb dos unitats enteres (latència 1 d'entrada i de sortida), una de desplaçament (latència 1 d'entrada i de sortida), una d'accés a memòria (latència 2 d'entrada i de sortida) i una de salt (latència 1 d'entrada i de sortida).

```
1        load   r1 <- base_a(r9)
2        shiftl  r0 <- 2,r1
```

```
3           load   r2 <- base_b(r9)
4           shiftr 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) Dibuixa el graf de blocs bàsics.




```
          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:
```

Aplica Trace Scheduling a la seqüència més llarga (amb més instruccions), fes l'assignació d'unitats funcionals començant per les instruccions que tindran un pc més petit. Utilitza la taula per donar la solució.

Utilitzant el valor LST com a prioritat tenim:

Taula de reserva

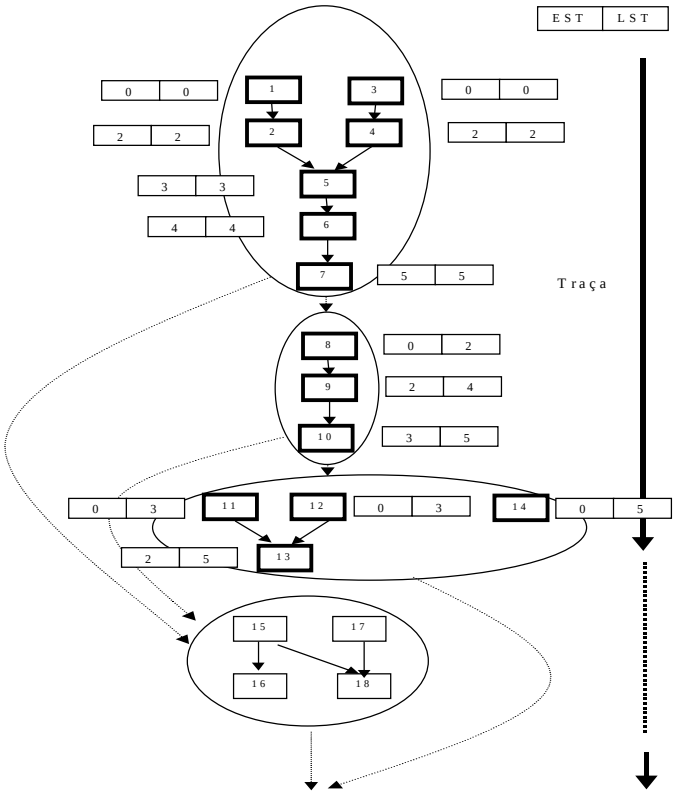
cicle	Llista	UF: enter	UF: enter	UF: shifter	UF: memòr.	UF: salt	UF:
1	1/0, 3/0, 8/2, 11/3, 12/3, 14/5	12			1	14	
2	3/0, 8/2, 2/2, 11/3			2			
3	3/0, 8/2, 11/3				3		
4	8/2, 4/2, 11/3			4			
5	8/2, 11/3, 5/3	5			8		
6	11/3, 6/4, 9/4	6	9				
7	11/3, 7/5, 10/5				11	7	

8	10/5, 13/5					10	
9	13/5				13		

Un cop ordenat i afegit el codi de compensació el codi podria ser:

```
            1           load    r1 <- base_a(r9)
            12          const   r10 <- 27
            2           sll     r0 <- 2,r1
            3           load    r2 <- base_b(r9)
            4           sra     r3 <- 2,r2
            5           and     r4 <- r3,r0
            8           load    r6 <- base_c(r9)
            6           cmp     r5 <- r4,100
            9           cmp     r7 <- r6,0
            11          load    r8 <- base_c(r9)
            7           bg      r5,LL2
            10          be      r7,LL2
            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:
```

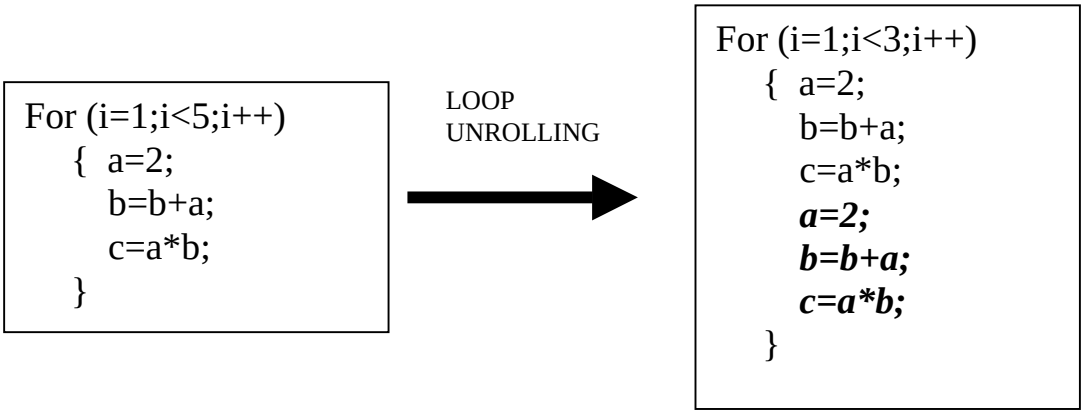
Observis que la instrucció 14 s'ha posat l'ultima en comptes de la primera com sortiria del anàlisi de la taula de reserva. Com és lògic no és possible posar el salt incondicional al principi. Per un altre cantó no fa falta afegir codi de compensació ja que la instrucció 13, que és l'única que s'hauria de desfer, és posterior als dos salts condicionals.



5.4 LOOP UNROLLING.

Un altre cas d'optimització del software el tenim en el cas dels bucles que ens podem trobar en el codi generat. Els bucles per definició són un conjunt d'instruccions seqüencial que es repetiran un conjunt de vegades segons ens indiqui la variable de control, i per tant, tenen com a característica que en molt casos coneixem el nombre de passades que es faran sobre aquell conjunt d'instruccions. A part també tenim que normalment els bucles estan formats per un conjunt petit d'instruccions que el formen i per tant la reordenació mitjançant el list schedulling (o trace schedulling) d'aquestes instruccions no optimitza de forma molt apreciable l'execució del mateix.

El loop unrolling consisteix en desdoblar el màxim de vegades possibles el bucle duplicant les instruccions i reduint la variable de control de la mateixa amb la finalitat de millorar la reutilització de registres, minimitzar recorrències i pujar el paral·lelisme a nivell d' instrucció. El nombre de vegades que es desenrotlla és determinat de manera automàtica pel compilador o bé pel programador des de la línia de comandes (en compilar) o mitjançant directives.. Per exemple si tenim:



Aquest mètode es podrà aplicar sempre que coneixem una variable de control que ens asseguri el nombre de passades que es produiran. Una vegada fet el Loop Unrolling s'aplicarà el List Schedulling o el Trace Schedulling del codi resultant.

Un altre exemple del Loop Unrolling ho trobem en el desdoblament de operacions que es realitzen en arrays per dotar d'una major velocitat:

Codi original en C

```
for (i = 0; i  < 24; i++)
{
      v[i] = v[i-2] + X * w[i];
}
```

Desenrotllament

```
for (i=0;i<24;i++)
{
        v[i]=v[i-2]+X*w[i];
        v[i+1]=v[i-1]+X*w[i+1];
       v[i+2]=v[i]+X*w[i+2];
       v[i+3]=v[i+1]+X*w[i+3];
}
```

Si la variable índex no és múltiple de l' increment el desenrotllament es complica una mica:

Codi original

```
for (i = 0; i < n; i++)
{
   a[i] = b[i];
```

Codi optimitzat

```
for (i=0; i<(n%4); i++)
{
     a[i+0]=b[i+0];
```

```
}                                   a[i+1]=b[i+1];
                                    a[i+2]=b[i+2];
                                    a[i+3]=b[i+3];
                                    i+=4;
                                }
```

Si els límits del cicle són constants i el nombre d'iteracions és petit, el cicle pot ésser substituït completament per còpies del seu cos, tal com es mostra al següent exemple:

Codi Original:

```
for (i=1; i < 5 ; i++)
{
 a[i] = b[i]/a[i-1];
}
```

Codi optimitzat

```
a[1] = b[1] / a[0];
a[2] = b[2] / a[1];
a[3] = b[3] / a[2];
a[4] = b[4] / a[3];
```

Pels problemes de càlcul que han de convergir a un resultat, els compiladors refusen el desenrotllament, ja que pot alterar el resultat.

Un cicle desenrotllat és més gran que la versió original, pel que serà més llarg a la caché d'instruccions, pel que pot produir-se que la versió desenrotllada sigui més lenta al produir-se més fallades de caché d'aquest tipus.

Un exemple seria el següent:

En un sistema superescalar tenim el següent codi:

```
1       mov r12 var_S1
2       load r1,(r12)0
3       add r0,r1,r2
4       add r3,r2,r1
5       store r3,(r12)0
6       mov r11, var_S3
```

Aquest codi es pot expressar en pseudocodi amb la següent estructura:

```
S0=S1+S2;
S1=S2+S1;
FOR (I=0;I<=5;I++)
    {       S4=S1*S0;
            S1=S2+S1;
            IF S1>S4  S3=S4+S3;
    }
```

```
7         mov r4,0
8    LL2:  cmp r5,r4,5
9         ble r5,LL5
10        b LL3
11   LL5: load r13, (r12)0
12        mult r6,r13,r0
13        add r7,r2,r13
14        store r7, (r12)0
15        cmp r8,r7,r6
16        ble r8,LL6
17        load r9, (r11)0
18        add r10,r6,r9
19        store r10, (r11)0
20   LL6: add r4,1,r4
21        b LL2
22   LL3:
```

Si suposem que la disposem de les següents unitats funcionals:

UNITATS	NÚMERO	LATÈNCIA ENTRADA	LATÈNCIA SORTIDA	OPERACIONS
De salt	1	1	1	b, ble
Senceres	1	1	2	add
Multiplicació	1	2	4	mult
Accés memòria	1	1	1	load, store
Complements	1	1	1	mov, cmp

Sabent que la condició del IF es compleix un 70 % de les vegades.

Contesta:

a) Aplica la tècnica de Loop Unrolling a aquest codi. Proposa, en cas de fer falta, codi de correcció.
b) Dibuixa els blocs bàsics que representa el codi original
c) Realitza un trace schedulling (utilitzant l'EST) del codi original, i tenint en compte les condicions donades. Critica el resultat obtingut de manera automàtica mitjançant aquest mètode. Comenta si fa falta o no codi corrector, i, en cas de fer falta, que hauria de fer aquest codi.

Solució:

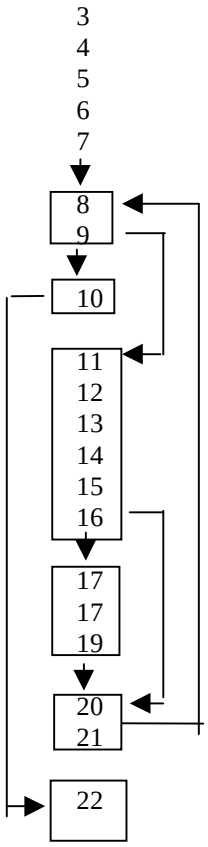
a) Loop Unrolling. Desenvoluparem dos cops la iteració. No fa falta afegir cap codi de correcció, la iteració té els límits predefinits.

```
1        mov r12 var_S1
2        load r1,(r12)0
3        add r0,r1,r2
4        add r3,r2,r1
5        store r3,(r12)0
6        mov r11, var_S3
7        mov r4,0
8    LL2:  cmp r5,r4,5
9        ble r5,LL5
10       b LL3
11   LL5: load r13, (r12)0
12       mult r6,r13,r0
13       add r7,r2,r13
14       store r7, (r12)0
15       cmp r8,r7,r6
16       ble r8,LL6
17       load r9, (r11)0
18       add r10,r6,r9
19       store r10, (r11)0
20   LL6: load r13, (r12)0
21       mult r6,r13,r0
22       add r7,r2,r13
23       store r7, (r12)0
24       cmp r8,r7,r6
25       ble r8,LL7
26       load r9, (r11)0
27       add r10,r6,r9
28       store r10, (r11)0
29   LL7: add r4,2,r4
30       b LL2
31   LL3:
```

Aquest codi es pot expressar en pseudocodi amb la següent estructura:

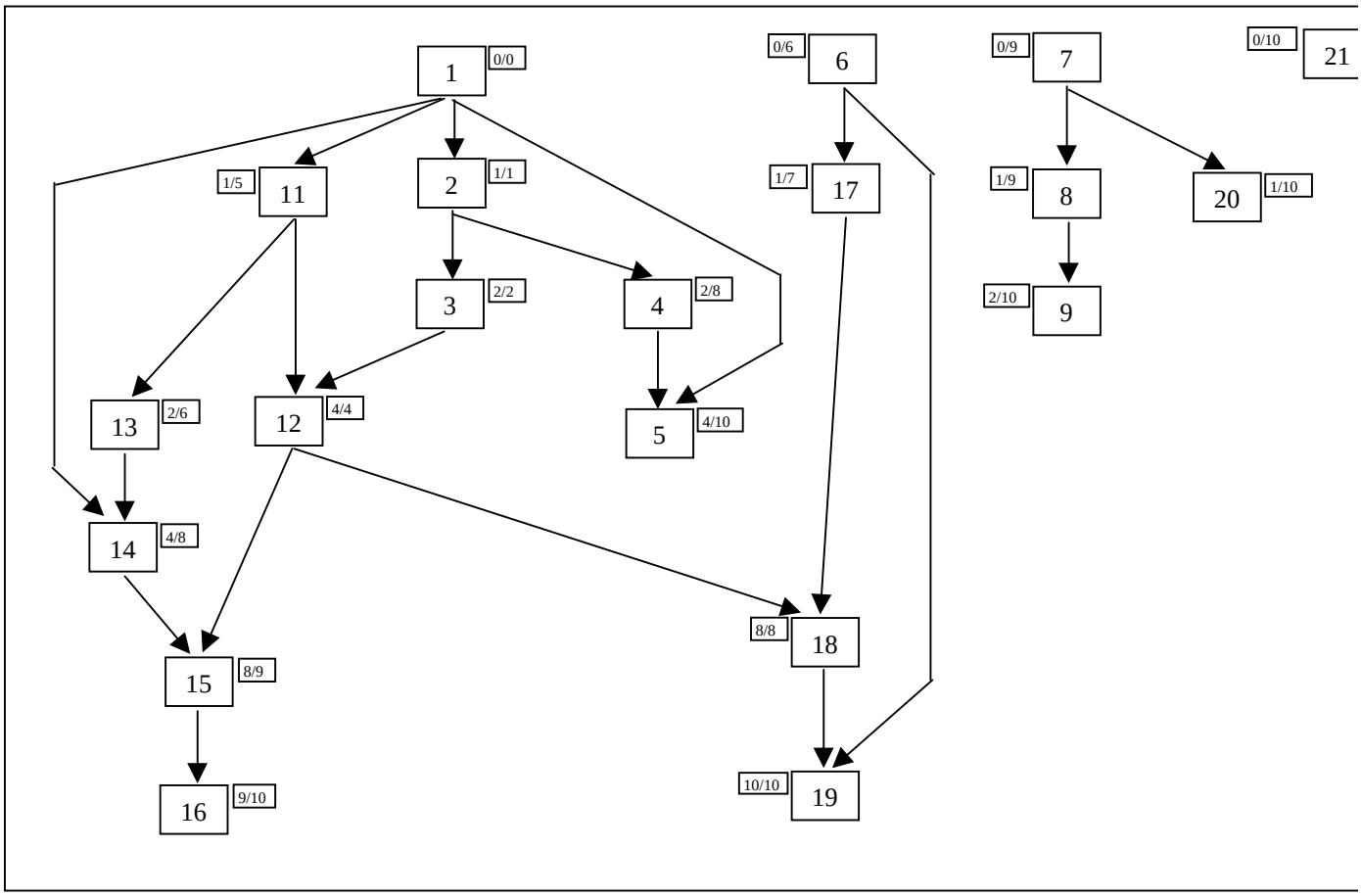
```
S0=S1+S2;
S1=S2+S1;
FOR (I=0;I<=5;I=I+2)
    {    S4=S1*S0;
         S1=S2+S1;
         IF S1>S4  S3=S4+S3;
         S4=S1*S0;
         S1=S2+S1;
         IF S1>S4  S3=S4+S3;
    }
```

b) Blocs bàsics:



c) Trace scheduling: Seguint les indicacions de l'enunciat, escollim la traça que recorre el IF, per tant, considerarem totes les instruccions excepte la 10 que és la de sortida de la iteració. Finalment hauriem d'incorporar-la i afagir el codi de correcció necessari per solucionar els casos en que no s'entra en el IF.

- Càlcul dels EST i LST



- Taula de reserva (Utilitzem els EST com a prioritat, i comencem des dels cicles alts.

Cicle	Llista	Salt	+	*	L/S	Comp
1	1(0)					1
2	2(1)				2	
3	3(2)		3			
4	11(1)				11	
5	12(4)			12		
6						
7	13(2), 6(0)		13			6

8	4(2), 17(1), 7(0)		4		17	7
9	18(8), 14(4), 8(1), 21(0)	21	18		14	8
10	15(8), 5(4), 9(2), 21(0)	9			5	15
11	19(10), 16(9), 5(4), 9(2), 20(1), 21(0)	16	20		19	

- Codi generat:

```
1          mov r12 var_S1
2          load r1,(r12)0
3          add r0,r1,r2
11    LL5: load r13, (r12)0
12         mult r6,r13,r0
13         add r7,r2,r13
6          mov r11, var_S3
4          add r3,r2,r1
17         load r9, (r11)0
7          mov r4,0
18         add r10,r6,r9
14         store r7, (r12)0
8     LL2:    cmp r5,r4,5
21         b LL2
15         cmp r8,r7,r6
5          store r3,(r12)0
9          ble r5,LL5
19         store r10, (r11)0
16         ble r8,LL6
20    LL6: add r4,1,r4
10         b LL3
22    LL3:
```

Instruccions de la iteració

El mètode de Trace Scheduling no té en compte les estructures iteratives, per tant, el resultat d'aplicar-lo fa que moltes instruccions de fora de la iteració es barregin amb les de la iteració. Com a conseqüència tenim un codi estrany difícilment aplicable. Una solució seria fer el Trace Scheduling per parts, les iteracions per un cantó i la resta per un altre.

Respecte al codi de correcció, en principi, faria falta desfer les instruccions 19 en el cas de no haver d'entrar en el IF. Per un altre canto, degut al desordenament de la iteració, també hauríem de tenir en compte les instruccions a desfer quan es surt de la iteració. Aquest cas té el problema sobreafegit que s'ha comentat en el primer paràgraf.

5.5 Altres tècniques d'optimització sw.

5.5.1 Tècniques de folding/compactació de codi

La tècnica d'optimització més trivial és la que es refereix a la compactació del codi, és a dir, que algunes expressions poden ésser simplificades mitjançant l'avaluació del compilador. Per exemple, donat el següent codi:

a=8+4+var1/var2

el compilador generaria:

```
a=12+var1/var2
```

Dins d'aquesta mateixa tècnica hi ha el que es coneix com a propagació de constants. Consisteix em substituir les constants pel seu valor concret. Per exemple:

```
const int a=8.7;
resultat1=a*8;
resultat2 = a*variable1;
```

El codi generat en aquest cas seria:

```
resultat1=69.6;
resultat2 = 8.7*variable1;
```

Aquest tipus d'optimització requereix que es tinguin presents les regles bàsiques d'àlgebra, com la commutativa i l'associativa. Òbviament, les tècniques de folding s'usen quan es tenen optimitzacions locals. Internament un compilador té als seus algorismes d'optimització un mòdul de reducció de variables simples, d'altres de simplificació d'operacions en assignació i d'altres de simplificació d'expressions lògiques i d'aritmètica.

5.5.2 *Eliminació de codi redundant i codi mort*

Una altra de les tècniques més comuns d'optimització en compiladors és l'eliminació d'operacions redundants en subexpressions, aconseguint així una reducció de codi. Per exemple, suposem l'expressió:

```
x=a*sin(y)+(sin(y)*2);
```

Com que la funció sin és molt costosa, el compilador generaria el següent codi:

```
temp1 = sin(y);
x = a*temp1+(temp1*2);
```

Això es fa freqüentment allà on hi ha avaluació d'expressions on hi ha dependències amb constants, això com quan hi ha operacions entre índexs a vectors o matrius.

L'eliminació de codi mort consisteix en eliminar fragments de codi inabastable o que no té sentit executar, ja que no té cap efecte. Per exemple, si tenim el fragment:

```
i=1;
temp=1;
temp=2;
i=temp*2;
```

veiem que les dues primeres instruccions no tenen sentit, ja que amb les dues segones perdem els valors que havíem assignat a i i temp, pel que el codi generat seria una cosa del tipus:

```
temp=2;
i=temp*2;
```

Codi inabastable seria aquell que mai s'arriba a executar, com, per exemple, aquell que es troba després del return d'una funció:

```
v=kilv(p);
return(v);
v=9;
```

5.5.3 Coma flotant

Els números en coma flotant (double, float i long double) són representacions inexactes dels números reals, a l'igual que les operacions que s'hi realitzen. Els compiladors d'IRIX generen codi de coma flotant d'acord amb els estàndards IEEE 754.

Si es desitja velocitat i no és tant important l'exactitut dels resultats, es pot implementar al nivell 3 d'optimització, on es realitzen transformacions d'expressions aritmètiques al codi sense considerar els estàndards.

En la ORIGIN2000, l'opció de -OPT:roundoff=2 es pot invocar tota sola sense especificar l'opció -O3 a la línia de compilació i ambdós tenen el mateix efecte d'optimització de coma flotant al codi.

5.5.4 Inlining

Escriure un programa en mòduls proporciona molts avantatges: es pot obtenir un tamany de codi petit i una consistència que permet una major reutilització del codi. Però les crides a funcions poden ser oepracions motl costoses, especialment si la funció és petita. La tècnica d'inlining substitueix la crida de la funció pel seu cos.

El codi en línia pot provocar que s'augmenti la quantitat de codi, pel que no és recomanable quan augmenta molt la seva mida.

Alguns compiladors usen aquesta tècnica solament quan les funcions s'han definit dins l'arxiu, però n'hi ha alguns que permeten també l'inlining de funcions definides a altres fitxers.

Amb aquesta tècnica podem reduir el nombre d'invocacions a mètodes, pel que eliminem el nombre de salts i, per tant, tenim un doble guany:

Eliminem instruccions de més (cada salt implica empilar paràmetres, saltar i desempilar)

Permetem realitzar millor altres tècniques d'optimització.

Un exemple d'aplicació seria:

```
int add(int x, int y)
{
        return(x+y);
}
int sub(int x, int y)
{
        return(add(x,-y));
}
```

figura 1: Exemple de codi sense optimitzar (Inlining)

El codi optimitzat quedaria:

```
int sub(int x, int y)
{
        return(x+(-y));
}
```

5.5.5 Loop fusion

La fussió de cicles és una optimització convencional del compilador que transforma dos o més cicles adjacents en un de sol. A més a més, permet la reutilització de dades que estan als registres de la CPU i una millora a l'ús de la caché (si el cicle és gran). L'ús de les proves de dependències de dades permet la fusió de cicles tant com sigui possible.

En el següent exemple, els dos primers cicles poden fussionar-se i optimitzar-se conjuntament. La fussió d'aquests cicles disminueix el nombre d'intruccions for i la sincronització requerida per elles, millorant així l'eficiència i la velocitat.

Codi original

```
for (i=0;i<m;i++)
{
        a[i] = b[i] + c[i];
}

        for (j = 0; j < m; j++)
        {
                d[j] = a[j] + e[j];
        }
```

Codi optimitzat

```
for (ij=0;ij<m;ij++)
{
        a[ij]=b[ij]+c[ij];
        d[ij]=a[ij]+e[ij];
}
```

Com es pot observar, al codi sense optimitzar a[i] apareix en els dos cicles, pel que es pot optimitzar combinant els dos cicles com a la part dreta, permetent que els elements a[ij] estiguin disponibles immediatament per usar-se a la propera instrucció, permetent la reusabilitat de dades en mantenir el seu valor al registre corresponent.

5.5.6 Loop interchange

L'intervanvi de cicles consisteix en intercanviar un cicle inferior pel superior amb el propòsit de millorar l'accés a memòria. Això sol succeïr en programes en Fortran, on l'emmagatzemantge de les dades es fa per columna i la referència a ells es realitza de manera inapropiada (també en C depèn de la programació. Per exemple

Codi original

```
for (i=1; i<m; i++)
{
  for (j=1;j<n;j++)
  {
      a[i][j] = a[i-1][j] + 1.0;
  }
}
```

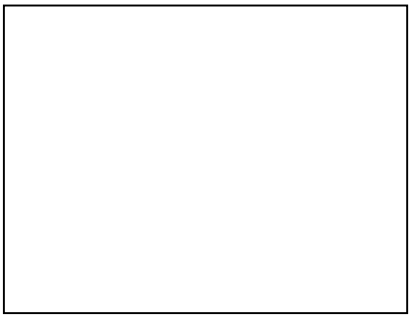
Intercanvi de cicles

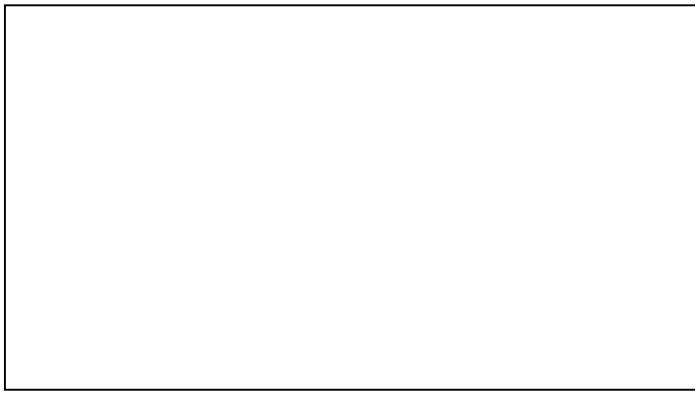
```
for (j=1;j<n;j++)
{
for (i=1; i<m; i++)
{
        a[i][j] = a[i-1][j] + 1.0;
}
}
```

En el costat esquerre, si m és un número gran i n un número molt petit, la càrrega d'informació de les dades de memòria RAM a caché serà molt feixuc, ja que les dades referenciades estan molt separades entre elles (es troben en diferents blocs).

5.5.7 *Loop fission/distribution*

Aquesta tècnica és l'oposada a la "loop fusion". En comptes de fusionar loops distribuirem els loops en múltiples peces o els fissionarem. Com ja teníem en la fusió, la fissió ens permetrà poder aplicar noves tècniques d'optimització. Per exemple:





```
for(i=0;i<N;i++)
{
  a[i]=b[i]+c[i];
  d[i]=d[i+1]+e[i];
}
```

Utilitzant “loop fission”, ens quedarà:

```
for(i=0;i<N;i++)
  a[i]=b[i]+c[i];

for(j=0;j<N;j++)
  d[j]=d[j+1]+e[j];
```

A més, la tècnica “loop fission” també ens permetrà millorar la utilització dels registres en loops interns grans. Si l’optimitzador decideix que hi pot haver-hi un problema de falta de registres s’activa automàticament el “loop fission”. L’optimitzador utilitza heurístiques per decidir com dividir les instruccions dels loops resultants.

5.5.8 *Caché Blocking*

És una tècnica d' optimització de l' ús de la memòria caché en la que matrius d' una grandària considerable (que no caben completament a la caché) es divideixen en petits blocs que s' ajusten a la mida d' aquesta memòria, disminuïnt així les falles de caché

Es produeix una falla de caché quan el processador no troba una dade a la memòria caché, havent d' esperar que aquesta dada sigui copiada del caché secundari o de memòria principal. Per exemple:

```
#define m 1000
#define n 1000
#define p 1000
float a[m,p];
float b[p,n];
float c[m,n];

for (j=1; j<n;j++)
{
        for (i=1;i<m;i++)
        {
                a[i][j] = rand();
                b[i][j] = 2.0;
        }
}

for(j=1;j<n;j++)
{
        for(i=1;i<m;i++)
```

```
        {
                for (k=1;k<p;k++)
                {
                        c[i][j] = a[i][k]*b[k][j];
                }
        }
 }
```

Si m, n i p són petits, l'execució d'aquest programa serà ràpida, però si són molt grans serà un cas contrari, es tindrà massa falles de caché.

Això es pot veure més clarament amb l'execució del programa amb perfex sense especificar el nivell d'optimització.

```
% cc multmat.c -o multmat
% perfex -e 25 multmat  <-  Per obtenir el número de falles de cahcé de L1

0 Cycles.....49002735555
25 Primary data caché misses.....1138524786
```

Temps d'execució 251 segons = 4.2 minuts

```
% perfex -e 26 multmat  <-  Per obtenir les falles de caché de L2

0 Cycles.....49018761168
26 Secondary data caché misses.....44515279
```

Aquest comportament es deu a que:

Cada element de la matriu c[i][j] es calcula llegint tota la fila de a[i][k] i tota la columna de b[k][j]

Per calcular una columna de c[i][j] cal llegir totes les files de la matriu a[i][k] i totes les columnes de b[k][j].

Per tant, per calcular tota la matriu c[i][j] de m x n cal recorrer la matriu a n cops i la matriu b m vegades. Com que les matrius a i b no caben a la caché, els valors de les files i les columnes que estan a la caché han de ser esborrats quan calgui llegir la resta d'elements (que estan a la memòria principal), provocant que no hi hagi reutilització de les dades anteriors que estaven a la caché. En aquest

exemple es portaran les dades de memòria principal a caché secundari n vegades per a i m cops per b.

La tècnica de cache blocking ens en dona la solució. Consisteix en dividir les matrius en submatrius o blocs, teninr multiplicacions de matrius de mida petita independents. Això permet que es puguin carregar les dades de cada bloc a caché i reutilitzar-los tots els cops que sigui necessari sense necessitat de llegir a memòria principal

Si compilem l'exemple amb O3 i executem amb perfex:

```
% perfex -e 25 multmat  <- Per obtenir les falles en caché de L1

0 Cycles.....31220848
25 Primary data caché misses.....1426
```

Obtenim un temps d'execució de 0.1 segons.

```
% perfex -e 26 multmat  <-  Per obtenir les falles en caché de L2

0 Cycles.....31214529
26 Secondary data caché misses.....397
```

Per tant es redueix el temps d'execució de 4.2 minuts a 0.1 segons.

Per tal d'utilitzar aquesta tècnica s'empra el pragma següent:

#pragma blocking size ([l1][l2])

on l1 és la mida de bloc per la caché nivell 1 i l2 la mida de la caché de nivell 2.

5.5.9 *Software Pipelining*

Aquesta és una de les tècniques que donen un rendiment més alt en arquitectures superescalars. Com ja se sap en una arquitectura superescalar es poden executar diverses instruccions en un cicle de rellotge. El compilador en aquesta tècnica genera seqüències d'instruccions que s'adapten cuidadosament a les unitats d'execució múltiples del processador. De forma que cada cicle de rellotge estigui plenament ocupat fent que es treballi a la màxima capacitat.

Així en aquesta tècnica s'executen instruccions simultàniament de diferents iteracions de forma que ens trobem que hi pot haver una iteració que comença a executar-se mentre que una altra s'està acabant.

En la **¡Error!No se encuentra el origen de la referencia.** veiem una iteració que tot seguit estudiarem:

```
for ( i = 0; i < N; i ++)
{
        A[i] = A[i] * B + C;
}
```

Podem veure les instruccions d'una iteració, en total són 6 instruccions.

Cicles de rellotge	Instrucció	Comentari
1	LOAD	Llegeix A[i]
2	MUL	Multiplica A[i] per B
3	-	La multiplicació tarda 2 cicles
4	ADD	Suma C a A[i]*B
5	-	La suma tarda 2 cicles
6	STORE	Guarda a memòria A[i]

Aplicant la tècnica de software pipelining en aquest bucle tenim que les instruccions s'executaran de la forma que veiem en la figura 3.

Cicle de rellotge	Iteració 1	Iteració 2	Iteració 3	Iteració 4	Comentari
1	LOAD	-	-	-	Llegeix A[1]
2	MUL	-	-	-	A[1] * B
3	-	LOAD	-	-	Llegeix A[2]
4	-	MUL	-	-	A[2] * B
5	ADD	-	LOAD	-	A[1]*B + C i llegeix A[3]

6	-	-	MUL	-	A[3]*B
7	-	ADD	-	LOAD	A[2]*B + C i llegeix A[4]*B
8	STORE	-	-	MUL	Escriu A[1] i A[4]*B
9	-	-	ADD	-	A[3]*B + ©
10	-	STORE	-	-	Escriu A[2]
11	-	-	-	ADD	A[4]*B + C
12	-	-	STORE	-	Escriu A[3]
13	-	-	-	-	
14	-	-	-	STORE	Escriu A[4]

figura 2: Exemple de Software Pipeling amb 4 iteracions

Aquí podem veure que el número de cicles que hem tardat per executar 4 iteracions són 14 donant un promig de 3’5 instruccions per iteració i no de 6 com hem vist anteriorment.

Aquesta tècnica no es pot aplicar sempre, té unes limitacions. Així doncs no podem aplicar software pipeling quan tenim que una iteració és depenent d’una altra, quan tenim iteracions petites, si es fan crides a funcions o si tenim aliasing1.

Per tal de compilar utilitzant aquesta optimització caldrà afegir a la línia de compilació el paràmetre –OPT:swp= [ON/OFF].

Així tenim que la línia per compilar ens queda de la següent forma:
cc –n32 –mips4 –r10000 –OPT:swp=ON nomfitxer.cpp

si volem deshabilitar l’optimització es fa així
cc –n32 –mips4 –r10000 –OPT:swp=OFF nomfitxer.cpp

Gather-Scatter Optimization

[1] Dos variables diferents fan referència a la mateixa direcció de memòria

En la tècnica de “software pipelining” intentem millorar el rendiment executant en paral·lel diverses instruccions de múltiples iteracions. Això pot resultar difícil quan aquestes instruccions són condicionals. Considerem el següent exemple:

```
for(i=0;i<N;i++)
{
 if(T[i]>0) A[i]=2*B[i];
}
```

figura 3: Exemple de codi sense aplicar Gather-Scatter

Ignorant la condició IF, la tècnica de SW pipelining pujarà el load de B[i-1], executante’l en paral·lel en les primeres iteracions de la multiplicació. Si tenim en compte la condició, això no és estrictament possible. El generador de codi transformarà els IF del loop per executar-se el cos del IF en cada iteració. La conversió d’aquest IF no funcionarà de manera efectiva quan el camí del IF no s’agafi majoritàriament. Una alternativa pot ser fer una optimització del tipus “gather-scatter”, la qual dividirà el loop com segueix a continuació:

```
inc=0;
for(i=0;i<N;i++)
{
 Tmp[inc]=i;
 if(T[i]>0) inc=inc+1;
}
for(i=0;i<inc;i++)
{
 A[Tmp[i]]=2*B[Tmp[i]];
}
```

figura 4: Exemple del codi després d’aplicar Gather-Scatter

Ara el generador de codi transformarà el primer IF del primer loop, però no li caldrà transformar el segon. El segon, per tant, podrà optimitzar-se fent un SW pipelining estalviant així haver de fer multiplicacions innecessàries.

5.5.10 Prefetching

Inicialment els processadors carregaven instruccions i dades directament de la memòria en els registres. Això provocava que el processador no treballés a la velocitat per la qual estava dissenyat degut al retard (latència) que es generava al estar llegint directament de memòria. Amb la implementació d'una cache secundària (L2) que fes d'intermediària en la transferència d'informació entre el processador i la memòria principal es va aconseguir reduir aquesta latència.

En aquesta cache secundària es guarda un interval de direccions que són un subconjunt de direccions de la memòria principal, i permet que dades i instruccions de l'aplicació estiguin allà, l'accés a aquesta memòria es excepcional i influeix en la velocitat que millora l'execució dels nostres programes.

De la mateixa manera, per accelerar més el rendiment de la nostra aplicació, s'ha implementat l'ús de cache primària (L1) que es troba en el propi processador (on-chip cache) i actua d'intermediària entre la cache secundària i el processador, les instruccions i dades es col·loquen de la memòria principal a la cache secundària i de la cache secundària a la cache primària. D'aquí sorgeix la tècnica Data Prefetching.

5.5.11 Global code motion

Tipus de planificació global. Es tracta de distribuir les instruccions en blocs bàsics durant un camí d'execució per tal de millorar el paral·lelisme d'instruccions i aprofitar millor els recursos de la màquina.

Aquesta tècnica ja s'utilitza en altres compiladors amb altres noms (seria una espècie de "trace scheduling", TS).

En els compiladors MIPS aquesta tècnica s'anomena "speculative code motion", ja que les instruccions mogudes s'executen de manera especulativa, a diferència d'altres tècniques com TS, on necessitem aplicar un codi de compensació un cop feta la planificació per evitar execucions erroneas.

Alhora de moure instruccions podem trobar-nos que podem moure instruccions que puguin causar excepcions al executar-se. Si som capaços de poder

desconnectar el tractament d'excepcions per alguns segments de codi, podrem millorar l'optimització d'aquests.

Tindrem diferents nivells d'especulació per tal d'optimitzar el moviment d'instruccions.

- Especulacions agressives: no moure instruccions a blocs bàsics on s'estan utilitzant de forma òptima els recursos del sistema.

- Especulació d'arrays: una manera d'especulació que pot oferir uns bons resultats és l'anomenada "bottom loading". Aquesta tècnica s'utilitza en el cas que es moguin instruccions que tractin arrays, de manera que pot succeir que accedim fora dels límits de l'array. Aquesta opció ens permetrà referències fora del límit dels arrays fent un "padding" dels arrays per prevenir referències que puguin causar excepcions de memòria.

- Especulació de punters: aquesta opció ens permetrà fer moviments de manera especulativa de loads que tinguin involucrades accions sobre punters. En tindrem de dos tipus. La primera permetrà moviments de loads amb punters que poden ser NULL. La segona, permetrà moviments d'aquelles que tinguin referències del tipus *(p+n), per una n suficientment petita, cap a blocs que continguin encara referències a *p. Assumint que si p és una adreça vàlida p+n també ho serà.

- Especulació de loads estàtics (static loads): aquesta opció permetrà moviments de manera especulativa de loads que pertanyin a àrees de dades estàtiques (static data areas).