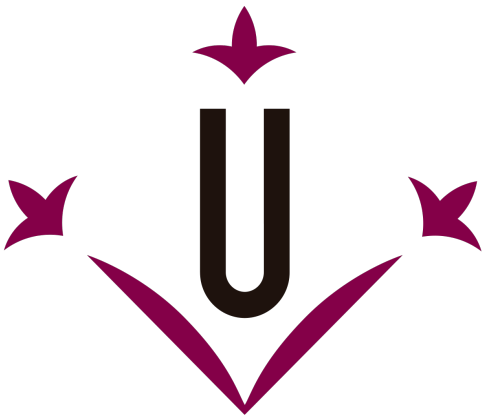

Universitat de Lleida

Processament Segmentat

Arquitectura de Dades

Escola Politècnica Superior

Grau en Enginyeria Informàtica

Yhislaine Nataly, Jaya Salazar

Valeria Adriana, Escalera Flores

23.05.2024

Índex

1 Introducció

En aquesta pràctica s'ha implementat un programa en llenguatge ensamblador (ASM), concretament en llenguatge MIPS, que realitza la següent operació:

$$\sum_{n=1}^{5} \frac{2n+3}{3n^3+4n+5} \tag{1}$$

L'objectiu de la pràctica no recau en la funcionalitat del programa, sinó en l'eficiència del mateix. Per aquest motiu, s'ha realitzat un anàlisi de les diferents opcions d'optimització que ofereix l'entorn de simulació WINMIPS. Aquestes opcions són:

- **Forwarding:** Permet que les dades es puguin llegir que una instrucció anterior hagi escrit en el registre de destí, és a dir, abans de la etapa *Write Back*
- **Delay slot:** Permet que la instrucció que segueixi a un salt condicional s'executi encara que el salt no s'hagi produït

2 Implementació

[1] En arquitectures MIPS, cada instrucció de salt (*branch*), com ara `bne`, conté un *delay slot*, és a dir, la instrucció immediatament següent s'executa independentment de si el salt es realitza o no. Aquest comportament s'aprofita per millorar el rendiment del pipeline del processador, evitant cicles de rellotge desaprofitats.

En el codi original, la instrucció de comparació del final del bucle era:

```
daddi r14, r0, 6
bne r1, r14, loop
```

Aquesta organització desaprofita el *delay slot*, ja que la instrucció següent al `bne` podria executar-se sense cap efecte útil si el salt es realitza. Per optimitzar-ho, es pot reordenar el codi de la següent manera:

```
bne r1, r14, loop
daddi r14, r0, 6    ; Instrucció al delay slot
```

[1] Respecte el que es va enviar cal mencionar que el codi es va canviar ja que haviem entes diferent l'implementació del delay

Ara, la instrucció `daddi r14, r0, 6` s'executa sempre, però com que només inicialitza un registre amb un valor constant necessari per a la comparació, és segura per incloure al *delay slot*. Aquesta reorganització conserva el comportament funcional original del programa, alhora que millora l'eficiència del codi.

Aquesta optimització és important en programes MIPS perquè permet reduir el nombre de cicles inactius i, per tant, millorar el rendiment global sense alterar la lògica del programa.

3 Resultats i anàlisis

3.1 Taula de resultats: nombre de cicles, CPI, RAW/WAW/WAR/structural stalls

	Sense opcions de millora	**forwarding**	**delay slot**	**Forwarding + delay slot**
Nombre instruccions	123	123	123	123
Nombre cicles	532	371	528	367
CPI	4325	3016	4293	2984
RAW stalls	401	340	401	340
WAW stalls	0	0	0	0
WAR stalls	0	0	0	0
Structural stalls	0	20	0	20

Figura 1: Taula de Resultats

3.2 Anàlisis de dependencies de dades

En aquesta secció s'analitzen les dependències de dades presents en el codi de càlcul de la suma, així com els possibles *stalls* estructurals associats a l'arquitectura pipeline de tipus MIPS. Les dependències poden ser de tres tipus: RAW (Read After Write), WAW

(Write After Write) i WAR (Write After Read).

Dependències RAW (Read After Write)

Les dependències RAW són les més comunes i es produeixen quan una instrucció necessita llegir un valor que encara no ha estat escrit per una instrucció anterior. A continuació es mostren alguns exemples presents al codi:

- `dmulu r5, r3, r4` depèn de `daddi r4, r0, 2` i `daddi r3, r2, 0`.
- `daddi r6, r5, 3` depèn de `dmulu r5, r3, r4`.
- `mtc1 r6, f1` depèn de `daddi r6, r5, 3`.
- `div.d f3, f1, f2` depèn de `cvt.d.l f1, f1` i `cvt.d.l f2, f2`.
- `add.d f4, f4, f3` depèn de `div.d f3, f1, f2` i de `l.d f4, 0(r20)`.
- `s.d f4, 0(r20)` depèn de `add.d f4, f4, f3`.
- `bne r1, r14, loop` depèn de `daddi r1, r1, 1` i de `daddi r14, r0, 6`.

Aquestes dependències requereixen l'ús de *forwarding* per evitar *stalls*, especialment entre operacions com `div.d` i `add.d`, que treballen amb valors de punt flotant.

Dependències WAW (Write After Write)

Les dependències WAW es produeixen quan dues instruccions escriuen al mateix registre. Aquest codi no presenta conflictes WAW greus, ja que la reutilització de registres es fa de manera seqüencial i controlada. Tot i així, per exemple:

- `f1` s'escriu tant en `mtc1 r6, f1` com en `cvt.d.l f1, f1`.
- `f4` s'escriu en `l.d f4`, després en `add.d f4`, i finalment en `s.d f4`.

Aquestes escriptures successives no generen *hazards* si es respecta l'ordre d'execució, però podrien requerir vigilància en sistemes amb execució fora d'ordre.

Dependències WAR (Write After Read)

Les WAR són menys comunes en aquest codi, però poden aparèixer en arquitectures fora d'ordre. Un exemple potencial seria:

- `f1` és llegit a `div.d f3, f1, f2` després de ser convertit en `cvt.d.l f1`, però si hi hagués una escriptura posterior fora d'ordre podria haver conflicte.

Com que MIPS sol seguir una execució en ordre, aquestes dependències no es manifesten com a problemes greus en aquest context.

Stalls estructurals

Els *structurals stalls* es produeixen quan dues instruccions necessiten el mateix recurs de maquinari simultàniament. En aquest codi poden aparèixer en:

- Operacions de punt flotant: com `div.d`, `add.d`, `cvt.d.l`. Si l'arquitectura té una sola unitat de coma flotant, es poden produir *stalls*.
- Accés a memòria: `l.d` i `s.d` podrien competir si no hi ha separació entre memòria de dades i instruccions.

Aquestes situacions poden ser resoltes amb la duplicació de recursos o programació acurada amb *delay slots* i reordenació d'instruccions.

3.3 Càlcul del *speedup* respecte l'execució seqüencial

Considerem la següent expressió matemàtica:

$$\sum_{n=1}^{5} \frac{2n+3}{3n^3+4n+5}$$

Per a cada iteració del bucle (de $n = 1$ a 5), es realitzen les següents operacions:

- **Multiplicacions**: $3n^3$, $4n$, i $2n \Rightarrow$ 3 instruccions de multiplicació (`mult`) $\times$ 11 etapes = 33 cicles
- **Sumes**: $2n+3$, $3n^3+4n+5 \Rightarrow$ 2 instruccions de suma (`add`) $\times$ 8 etapes = 16 cicles
- **Divisió**: 1 instrucció de divisió en coma flotant (`div.d`) $\times$ 28 etapes = 28 cicles

Total de cicles per iteració:

$$\text{Cicles per iteració} = 33 + 16 + 28 = \boxed{77 \text{ cicles}}$$

Com que el bucle s'executa per $n = 1$ fins a $n = 5$, són 5 iteracions:

$$\text{Execució seqüencial total} = 5 \times 77 = \boxed{385 \text{ cicles}}$$

Ara, calculem el *speedup* utilitzant els diferents casos d'execució segmentada:

$$\text{Speedup} = \frac{\text{Cicles execució seqüencial}}{\text{Cicles execució segmentada}}$$

- **Sense opcions de millora:** $\frac{385}{532} \approx 0{,}724$
- **Forwarding:** $\frac{385}{371} \approx 1{,}038$
- **Delay slot:** $\frac{385}{528} \approx 0{,}729$
- **Forwarding + delay slot:** $\frac{385}{367} \approx 1{,}049$

3.4 Comparativa entre execucions

Una vegada calculat el *speedup* segons els diferents casos, es pot afirmar que l'execució segmentada continua oferint una millora respecte a l'execució seqüencial, tot i que amb una acceleració més moderada, d'entre 0,724 i 1,049 vegades, segons les tècniques d'optimització aplicades.

Partint del cas base, **sense opcions de millora**, s'obté un *speedup* de 0,724, cosa que indica que, en aquest cas concret, la segmentació no és suficient per millorar el rendiment respecte a l'execució seqüencial, possiblement a causa de penalitzacions per dependències i latències acumulades.

Amb la implementació del **forwarding**, el *speedup* augmenta lleugerament fins a 1,038, ja que aquesta tècnica permet reduir les penalitzacions per dependències de dades, evitant estats de parada innecessaris. Això es tradueix en una lleugera millora del total de cicles respecte a l'execució segmentada sense optimitzacions.

L'ús del **delay slot** ofereix un *speedup* de 0,729. Aquest resultat indica que, tot i que el reordenament d'instruccions pot ajudar a reduir algunes ineficiències del pipeline, l'efecte és limitat si no es combina amb altres tècniques.

Finalment, la combinació de **forwarding i delay slot** ofereix el millor resultat, amb un *speedup* de 1,049. Aquesta combinació aprofita tant l'eliminació de parades per dependència de dades com l'aprofitament de cicles potencialment desaprofitats, aconseguint un rendiment una mica superior al seqüencial. Això posa en evidència que, per a càlculs

amb baixa intensitat computacional o poques iteracions, les tècniques de segmentació han de ser complementades amb optimitzacions per ser realment eficients.

3.5 Valoració de l'eficiencia

Per determinar quina de les opcions d'optimització és més idònia per al nostre codi, és convenient analitzar el nombre de cicles per instrucció (CPI) i els riscos de dades que es produeixen en cada cas. El CPI es calcula com el quocient entre el nombre de cicles i el nombre d'instruccions. Així, es pot veure com el CPI ideal és 1, ja que voldria dir que per cada instrucció s'executa un cicle. El següent gràfic mostra el CPI per les diferents opcions d'optimització, incloent-hi l'execució seqüencial.

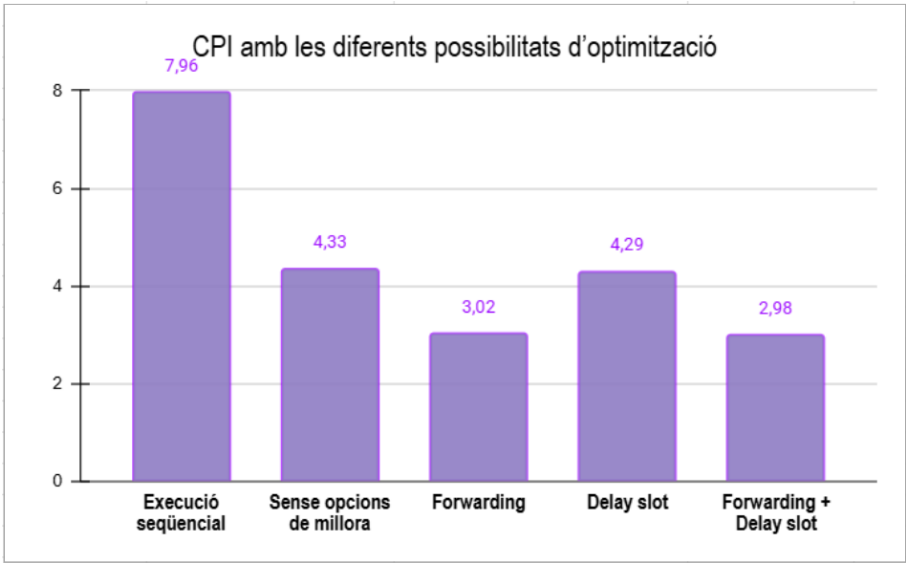

Figura 2: CPI amb les diferents possibilitats d'optimització

Es pot veure com l'execució segmentada, davant l'execució seqüencial, redueix el CPI en un 63%, aportant una millora substancial. Pel que fa a les diferents possibilitats d'optimització, el factor de millora que té més impacte en el nostre codi és clarament el *forwarding + delay slot*, El *delay slot*, en canvi, presenta una millora mínima respecte l'execició sense opcions de millora.

Pel que fa als riscos de dades, es pot observar clarament en la taula que el *forwarding* és el factor més eficaç a l'hora de reduir els RAW stalls, passant de 401 a 340, és a dir, una reducció de 15%. Això implica que moltes instruccions que abans requerien pauses poden ara continuar executant-se sense necessitat d'esperar. En canvi, el *delay slot* no aporta cap reducció en els RAW stalls, ja que només s'aprofita puntualment en la darrera

iteració dels bucles. Tot i que el *forwarding* introdueix 20 *structural stalls* a causa del solapament entre instruccions, el benefici global és clar: una reducció significativa del CPI (de 4,325 a 3,016), fet que demostra que el *forwarding* optimitza l'ús del pipeline i redueix la latència associada als riscos de dades.

Finalment, si s'hagués d'escollir una configuració d'optimització pel nostre codi en concret, escolliríem l'opció de *forwarding + delay slot*, ja que el seu CPI és el que més s'acosta al CPI ideal d'1. Per tant, en aquest cas, no optaríem per la combinació de *forwarding* perquè la seva millora és nul·la.

4 Conclusions

Aquesta pràctica ens ha permès observar de manera pràctica i detallada el funcionament d'un processador segmentat mitjançant el simulador **WinMIPS64**, així com els efectes que tenen les diferents tècniques d'optimització sobre el rendiment d'un programa. Hem pogut analitzar com opcions com el *forwarding* i el *delay slot* afecten directament al nombre de cicles d'execució, al CPI i als riscos de dades. Gràcies a l'anàlisi dels resultats obtinguts, hem entès per què certes configuracions milloren més que d'altres i hem pogut veure l'impacte real d'aquestes millores en l'execució segmentada. En definitiva, aquesta pràctica ens ha ajudat a consolidar els conceptes teòrics mitjançant una aplicació pràctica i visual del pipeline i les seves optimitzacions.

5 Bibliografía

Concepció Roig Mateu, *Processament Segmentat*, 2025.

W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 10a edició, Pearson, 2015.

A. J. Smith, *Cache Memories*, ACM Computing Surveys, vol. 14, núm. 3, pp. 473-530, 1982.