

IV.3. Predicția salturilor

Predicție (1)

- rezolvarea dependențelor de control
- ideea de bază - a "prezice" dacă un salt se execută sau nu
 - nu se așteaptă terminarea instrucțiunii de salt
- predicție corectă - fără blocaje în pipeline
- predicție eronată - se execută instrucțiuni care nu trebuiau executate
 - efectul acestora trebuie anulat

Predicție (2)

- spor de performanță - cât mai multe predicții corecte (nu neapărat 100%)
- o instrucțiune executată eronat produce efecte doar când rezultatul este scris la destinație
- rezultatele instrucțiunilor - memorate intern de procesor până când se verifică dacă predicția a fost corectă

Scheme de predicție

Tipuri de scheme

- statice
 - întotdeauna aceeași decizie
- dinamice
 - se adaptează în funcție de comportarea programului

Scheme statice de predicție (1)

1. Saltul nu se execută niciodată

- rata predicțiilor corecte $\approx 40\%$
- ciclurile de instrucțiuni
 - apar des în programe
 - salturi frecvente

Scheme statice de predicție (2)

2. Saltul se execută întotdeauna

- rata predicțiilor corecte $\approx 60\%$
- ratări dese - structuri de tip *if*

Scheme statice de predicție (3)

3. Salturile înapoi se execută întotdeauna, cele înainte niciodată

- combină variantele anterioare
- rată superioară a predicțiilor

Scheme dinamice de predicție (1)

- procesorul reține într-un tabel comportarea la salturile anterioare
 - salt executat/neexecutat
- un singur element pentru mai multe instrucțiuni de salt
 - tabel mai mic → economie de spațiu

Scheme dinamice de predicție (2)

Tipuri de predictor

- locali
 - rețin informații despre salturile individuale
- globali
 - iau în considerare corelațiile dintre instrucțiunile de salt din același program
- micști

Intel Pentium

- *Branch Target Buffer (BTB)*
 - cache asociativ pe 4 căi
 - 256 intrări
- Stările unei intrări
 - puternic lovit - se face salt
 - slab lovit - se face salt
 - slab nelovit - nu se face salt
 - puternic nelovit - nu se face salt

Implementarea BTB (1)

Memorarea și evoluția unei stări

- contor cu saturație pe 2 biți
 - poate număra crescător și descrescător
 - gama de valori - între 0 (00) și 3 (11)
 - din stările extreme nu se poate trece mai departe (doar înapoi)
- la fiecare acces, starea se poate schimba
 - condiția de salt este adevărată - incrementare
 - condiția de salt este falsă - decrementare

Implementarea BTB (2)

- codificarea stărilor
 - puternic lovit - 11 (se face salt)
 - slab lovit - 10 (se face salt)
 - slab nelovit - 01 (nu se face salt)
 - puternic nelovit - 00 (nu se face salt)
- de ce 4 stări?
 - a doua șansă - comportament pe termen lung

Implementarea BTB (3)

stare curentă	stare următoare	
	condiție salt adevărată	condiție salt falsă
00	01	00
01	10	00
10	11	01
11	11	10

Utilizarea cache-ului în predicție

Cache-ul de instrucțiuni

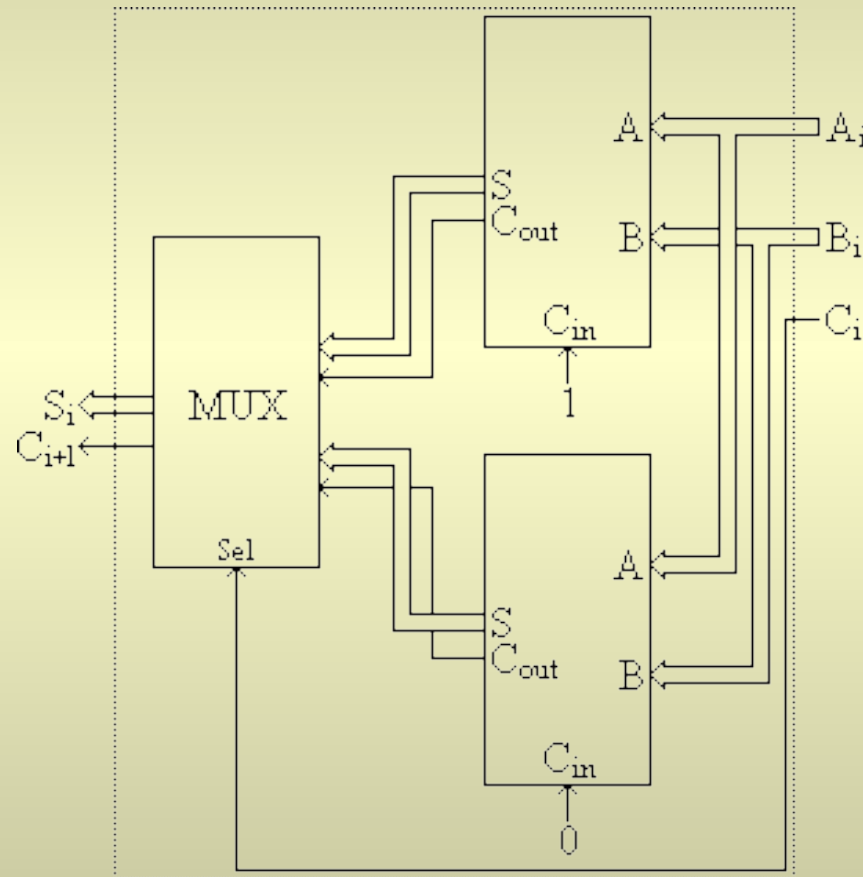
- reține vechea comportare a unui salt
 - condiție
 - adresă destinație
- *trace cache*
 - memorează instrucțiunile în ordinea în care sunt executate
 - nu în ordinea fizică

IV.4. Execuția speculativă

Execuție speculativă

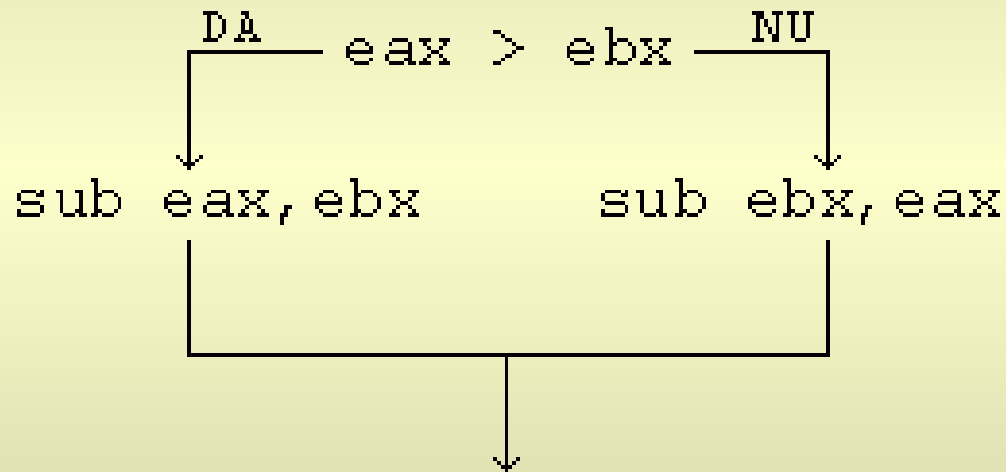
- înrudită cu predicția salturilor
- se execută toate variantele posibile
 - înainte de a ști care este cea corectă
- când se cunoaște varianta corectă, rezultatele sale sunt validate
- se poate utiliza și în circuite simple

Exemplu - sumatorul cu selecție



Cum funcționează? (1)

- instrucțiuni de salt condiționat



- ambele variante se execută în paralel
- cum se modifică regiștrii `eax` și `ebx`?

Cum funcționează? (2)

- nici una din variante nu îi modifică
- rezultatele scăderilor sunt depuse în regiștri temporari
- când se cunoaște relația între `eax` și `ebx`
 - se determină varianta corectă de execuție
 - se actualizează valorile `eax` și `ebx` conform rezultatelor obținute în varianta corectă

Execuție speculativă vs. predicție

- nu apar predicții eronate
 - rata de succes - 100%
- necesită mulți regiștri pentru rezultatele temporare
- gestiunea acestora - complicată
- fiecare variantă de execuție poate conține alte salturi etc.
 - variantele se multiplică exponențial

IV.5. Predicația

Predicație (1)

- folosită în arhitectura Intel IA-64
 - și în alte unități de procesare
- asemănătoare cu execuția speculativă
- procesorul conține regiștri de predicate
 - predicat - condiție booleană (bit)
- fiecare instrucțiune obișnuită are asociat un asemenea predicat

Predicație (2)

- o instrucțiune produce efecte dacă și numai dacă predicatul asociat este *true*
 - altfel rezultatul său nu este scris la destinație
- instrucțiunile de test pot modifica valorile predicatelor
- se pot implementa astfel ramificații în program

Exemplu

- pseudocod

```
if (R1==0) {
```

```
    R2=5;
```

```
    R3=8;
```

```
}
```

```
else
```

```
    R2=21;
```


Exemplu (continuare)

- limbaj de asamblare "clasic"

```
cmp R1, 0
```

```
jne E1
```

```
mov R2, 5
```

```
mov R3, 8
```

```
jmp E2
```

```
E1: mov R2, 21
```

```
E2:
```

Exemplu (continuare)

- limbaj de asamblare cu predicate

cmp R1, 0, P1

<P1>mov R2, 5

<P1>mov R3, 8

<P2>mov R2, 21

- predicatele P1 și P2 lucrează în pereche
 - P2 este întotdeauna inversul lui P1
 - prima instrucțiune îi modifică pe amândoi

IV.6. Execuția out-of-order

Execuție out-of-order

- instrucțiunile nu se mai termină obligatoriu în ordinea în care și-au început execuția
- scop - eliminarea unor blocaje în pipeline
- posibilă atunci când între instrucțiuni nu există dependențe

Exemplu

```
in al, 278
```

```
add bl, al
```

```
mov edx, [ebp+8]
```

- prima instrucțiune - foarte lentă
- a doua instrucțiune trebuie să aștepte terminarea primeia
- a treia instrucțiune nu depinde de cele dinaintea sa - se poate termina înaintea lor

IV.7. Redenumirea regiștrilor

Dependențe de date

- apar când două instrucțiuni folosesc aceeași resursă (variabilă/registru)
- numai când cel puțin una din instrucțiuni modifică resursa respectivă
- dacă resursele sunt regiștri, unele dependențe se pot rezolva prin redenumire


Tipuri de dependențe de date

- RAW (*read after write*)
 - prima instrucțiune modifică resursa, a doua o citește
- WAR (*write after read*)
 - invers
- WAW (*write after write*)
 - ambele instrucțiuni modifică resursa

Dependențe RAW

- dependențe "adevărate"
- nu pot fi eliminate

```
mov  eax, 5  
sub  ebx, eax
```

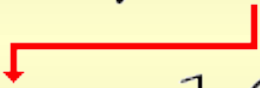


- valoarea scrisă în `eax` de prima instrucțiune este necesară instrucțiunii următoare

Dependențe WAR

- numite și antidependențe


```
add esi, eax  
mov eax, 16
```



```
sub ebx, eax
```

- prima instrucțiune trebuie executată înaintea celei de-a doua (nu se poate în paralel)

Dependențe WAR - rezolvare

<code>add esi, eax</code>		<code>add esi, eax</code>
<code>mov eax, 16</code>		<code>mov reg_tmp, 16</code>
<code>sub ebx, eax</code>		<code>sub ebx, reg_tmp</code>


Dependențe WAW

- dependențe de ieșire

```
div ecx
sub ebx, edx
mov eax, 5
add ebp, eax
```

The diagram illustrates a Write-After-Write (WAW) dependency between the instructions `div ecx` and `mov eax, 5`. A red line connects the `ecx` register in the first instruction to the `eax` register in the third instruction, indicating that the value of `ecx` is overwritten by the value of `eax` before it is used in the `sub ebx, edx` instruction.

Dependențe WAW - rezolvare

<code>div ecx</code>		<code>div ecx</code>
<code>sub ebx,edx</code>		<code>sub ebx,edx</code>
<code>mov eax,5</code>		<code>mov reg_tmp,5</code>
<code>add ebp,eax</code>		<code>add ebp,reg_tmp</code>

Utilitate

- ajută la creșterea performanței?
- crește potențialul de paralelizare
- mai eficientă în combinație cu alte tehnici
 - structura superpipeline
 - execuția out-of-order

Eficiența abordării

- mai mulți regiștri
 - folosiți intern de procesor
 - nu sunt accesibili programatorului
- de ce?
 - redenumirea se face automat
 - programatorul poate greși (exploatare ineficientă a resurselor)
 - creșterea performanței programelor vechi

IV.8. Hyperthreading

Hyperthreading (1)

- avem un singur procesor real, dar acesta apare ca două procesoare virtuale
- sunt duplicate componentele care rețin starea procesorului
 - regiștrii generali
 - regiștrii de control
 - regiștrii controllerului de întreruperi
 - regiștrii de stare ai procesorului

Hyperthreading (2)

- nu sunt duplicate resursele de execuție
 - unitățile de execuție
 - unitățile de predicție a salturilor
 - magistralele
 - unitatea de control a procesorului
- procesoarele virtuale execută instrucțiunile în mod întrepătruns

De ce hyperthreading?

- exploatează mai bine structura pipeline
 - când o instrucțiune a unui procesor virtual se blochează, celălalt procesor preia controlul
- nu oferă același câștig de performanță ca un al doilea procesor real
- dar complexitatea și consumul sunt aproape aceleași cu ale unui singur procesor
 - componentele de stare sunt foarte puține

IV.9. Arhitectura RISC

Structura clasică a CPU

CISC (*Complex Instruction Set Computer*)

- număr mare de instrucțiuni
- complexitate mare a instrucțiunilor → timp mare de execuție
- număr mic de regiștri → acces intensiv la memorie

Observații practice

- multe instrucțiuni sunt rar folosite
- 20% din instrucțiuni sunt executate 80% din timp
 - sau 10% sunt executate 90% din timp
 - depinde de sursa de documentare...
- instrucțiunile complexe pot fi simulate prin instrucțiuni simple

Structura alternativă

RISC (*Reduced Instruction Set Computer*)

- set de instrucțiuni simplificat
 - instrucțiuni mai puține (relativ)
 - și mai simple funcțional (elementare)
- număr mare de regiștri (zeci)
- mai puține moduri de adresare a memoriei

Accesele la memorie

- format fix al instrucțiunilor
 - același număr de octeți, chiar dacă nu toți sunt necesari în toate cazurile
 - mai simplu de decodificat
- arhitectură de tip *load/store*
 - accesul la memorie - doar prin instrucțiuni de transfer între memorie și regiștri
 - restul instrucțiunilor lucrează doar cu regiștri

Avantajele structurii RISC

- instrucțiuni mai rapide
- reduce numărul de accese la memorie
 - depinde de capacitatea compilatoarelor de a folosi regiștrii
- accesele la memorie - mai simple
 - mai puține blocaje în pipeline
- necesar de siliciu mai mic - se pot integra circuite suplimentare (ex. cache)

IV.10. Arhitecturi paralele de calcul

Calcul paralel - utilizare

- comunicare între aplicații
 - poate fi folosită și procesarea concurentă
- performanțe superioare
 - calcule științifice
 - volume foarte mari de date
 - modelare/simulare
 - meteorologie, astronomie etc.

Cum se obține paralelismul?

- structuri pipeline
 - secvențial/paralel
- sisteme multiprocesor
 - unitățile de calcul - procesoare
- sisteme distribuite
 - unitățile de calcul - calculatoare

Performanța

- ideal - viteza variază liniar cu numărul de procesoare
- real - un program nu poate fi paralelizat în întregime
- exemple
 - operații de I/O
 - sortare-interclasare

Scalabilitatea (1)

- creșterea performanței o dată cu numărul procesoarelor
- probleme - sisteme cu foarte multe procesoare
- factori de limitare
 - complexitatea conexiunilor
 - timpul pierdut pentru comunicare
 - natura secvențială a aplicațiilor

Scalabilitatea (2)

- funcțională pentru un număr relativ mic de procesoare
- număr relativ mare
 - creșterea de performanță nu urmează creșterea numărului de procesoare
- număr foarte mare
 - performanța se plafonează sau poate scădea

Sisteme de memorie

- după organizarea fizică a memoriei
 - centralizată
 - distribuită
- după tipul de acces la memorie
 - comună (partajată)
 - locală

Tipuri de sisteme multiprocesor

- sisteme cu memorie partajată centralizată
- sisteme cu memorie partajată distribuită
- sisteme cu schimb de mesaje