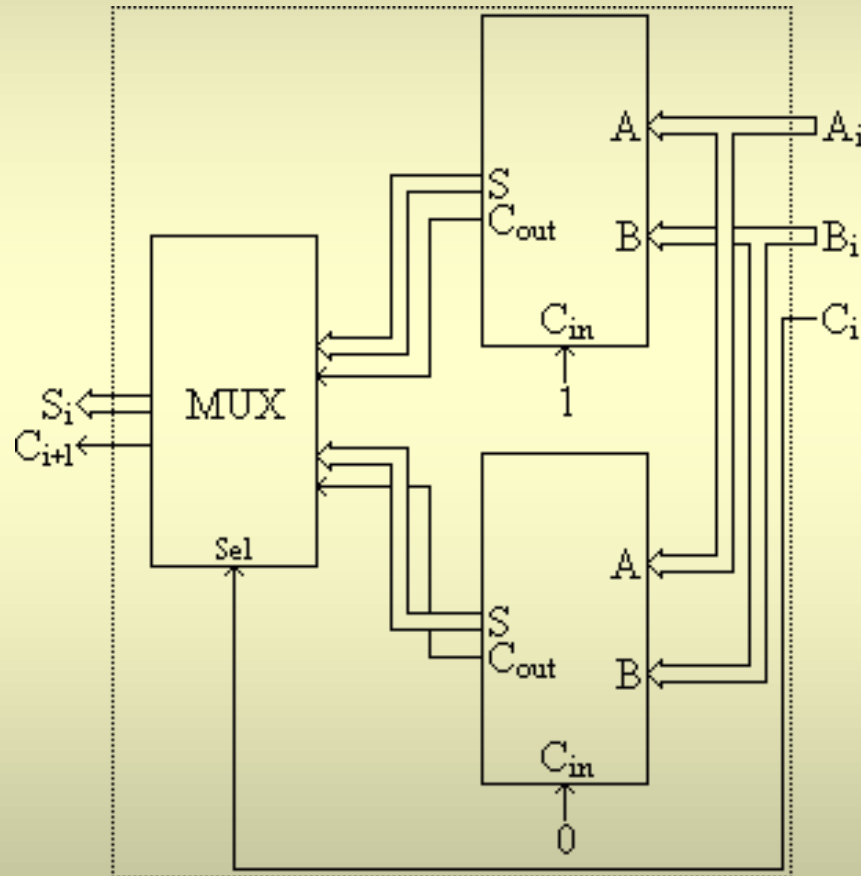


II.4. Execuția speculativă

Execuție speculativă

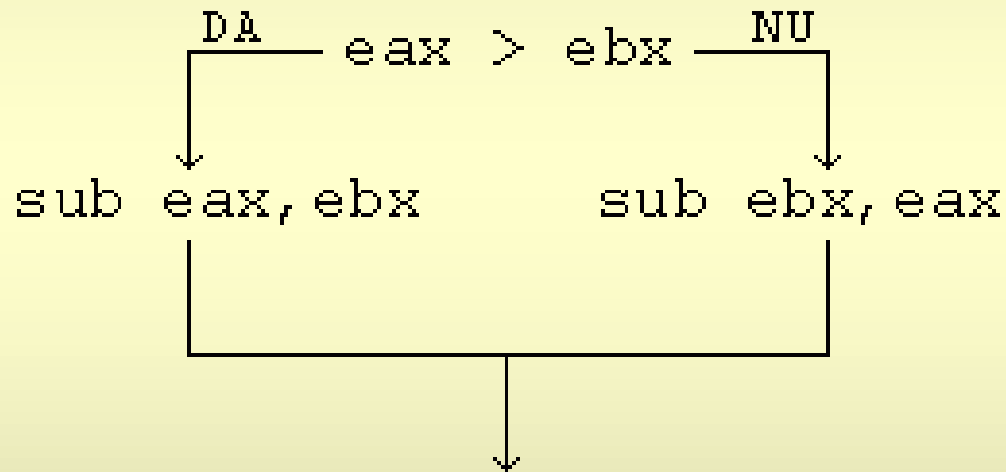
- înrudită cu predicția salturilor
- se execută toate variantele posibile
 - înainte de a ști care este cea corectă
- când se cunoaște varianta corectă, rezultatele sale sunt validate
- se poate utiliza și în circuite simple

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

II.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

II.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

II.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)

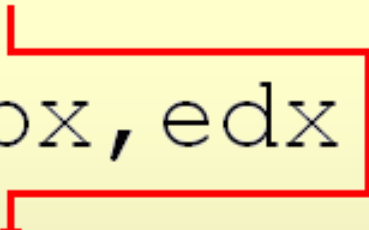
Dependențe WAR - rezolvare

<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
```



Dependențe WAW - rezolvare

<code>div ecx</code>		<code>div ecx, 3</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

II.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

II.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)

II.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

Memorie partajată centralizată

- denumiri
 - UMA (*Uniform Memory Access*)
 - SMP (*Symmetrical Multiprocessing*)
- memorie comună
- accesibilă tuturor procesoarelor
- timpul de acces la memorie
 - același pentru orice procesor și orice locație

Memorie partajată distribuită

- denumiri
 - DSM (*Distributed Shared Memory*)
 - NUMA (*Non-Uniform Memory Access*)
- memoria este distribuită fizic
- spațiu de adrese unic
 - văzut de toate procesoarele
- accesul la memorie - neuniform

Sisteme cu schimb de mesaje

- multicalculatoare
- fiecare procesor are propria memorie locală
 - nu este accesibilă celorlalte procesoare
- comunicarea între procesoare
 - transmiterea de mesaje explicite
 - similar rețelelor de calculatoare