Laborator 4 - ISA, Pipeline, Hazarde

1. Arhitectura Setului de Instrucţiuni
    1.1 RISC (Reduced Instruction Set Computer)
    1.2 CISC (Complex Instruction Set Computer)
    1.3 RISC vs CISC
2. Paralelism - Pipelining
3. Hazarde
    3.1 Tipuri
    3.2 Soluții

În cadrul acestui seminar se vor studia elemente de bază din arhitectura procesoarelor.
Arhitectura unui procesor este definită prin ISA şi prin microarhitectură.

1. Arhitectura Setului de Instrucţiuni

Arhitectura setului de instrucţiuni (ISA - Instruction Set Architecture) este folosită pentru a abstractiza funcţionarea internă a unui procesor. ISA defineşte “personalitatea” unui procesor: cum funcţionează procesorul d.p.d.v. al programatorului, ce fel de instrucţiuni execută, care este semantica acestora, în timp ce microarhitectura se referă la cum este implementată ISA, pipeline-ul de instrucţiuni, fluxul de date dintre unităţile de execuţie (dacă sunt mai multe), registre şi cache. Astfel, o ISA poate fi implementată de diferite microarhitecturi: la ARM, ARMv6 este un exemplu de ISA, şi este implementată de 4 procesoare, la Pentium numele ISA este IA-32 şi este implementată de diferite procesoare produse de Intel, AMD, Via, Transmeta. Astfel, ISA este cea mai importantă parte a design-ului unui procesor; alte aspecte cum sunt interacţiunea cu cache-ul, pipeline-ul, fluxul de date în procesor putând fi schimbate de la o versiune la alta a procesorului.
La ora actuală există două filozofii de design pentru un procesor: Complex Instruction Set Computer (CISC) şi Reduced Instruction Set Computer (RISC). În afară de acestea există şi ISA-uri pentru procesoare specializate, cum sunt GPU-urile pentru plăci grafice si DSP-urile pentru procesare de semnal.

1.1 RISC (Reduced Instruction Set Computer)

Caracteristicile care definesc RISC sunt:

- puţine instrucţiuni: ~100-200; complexitatea care este eliminată din ISA este de fapt transferată programatorului în limbaj de asamblare şi compilatoarelor; codul conţine mai multe instrucţiuni, şi în multe cazuri (mai ales în trecut (anii '70, '80)) acest fapt constituie o problemă
- instrucţiunile sunt de obicei codificate pe lungime fixă; permite o implementare mai simplă a UC

- execuţia instrucţiunilor într-un singur ciclu (folosind pipeline)
- **arhitectură load-store** - numai instrucţiunile de tip LOAD şi STORE lucrează cu memoria
- un număr mai mare de registre (din cauza load-store): 32-64; familii de arhitecturi: ARM, AVR, AVR32, MIPS, PowerPC (Freescale, IBM), SPARC (SUN)

Exemplu: încărcarea unui cuvânt de date din memorie într-un registru la arhitectura AVR32:

```
ld.w Rd, Rp++
```

`Rd` - registrul destinație

`Rp` - registrul ce conține adresa cuvântului; aceasta este incrementată după copierea conţinutului

1.2 CISC (Complex Instruction Set Computer)

Arhitecturile de tip CISC pun accentul pe hardware având instrucţiuni complexe ce reduc dimensiunea codului şi simplificând partea de software. Cele mai importante caracteristici sunt:

- instrucţiunile pot accesa memoria, având deja încorporate load-ul şi store-ul
- varietate mare de moduri de adresare a memoriei
- instrucţiunile au dimensiuni variabile şi necesită mai mulţi cicli de ceas pentru a fi executate
- necesită un nivel suplimentar între hardware şi ISA, controlul microprogramului - instrucţiunile sunt implementate în microcod.
- familii de arhitecturi: IA-32, x86-64

Exemplu:
La IA-32: `add [ebx],eax` (se aduna operandul de la adresa data de registrul ebx cu cel din registrul eax si rezultatul se stocheaza la adr primului operand)

La o arhitectura RISC:

```
load r1, $10
load r2, $11
add r1,r2
store  $10, r1
```

`$10, $11` sunt locații de memorie (adrese), numele instrucțiunilor sunt generice, nu aparțin unei arhitecturi anume

Pentru o arhitectura RISC am avea intai de incarcat operanzii din memorie folosind instrucțiuni de tip LOAD si apoi sa apelam instrucțiunea pentru inmultire, în final stocand rezultatul în memorie folosind o instrucțiune de tip STORE.

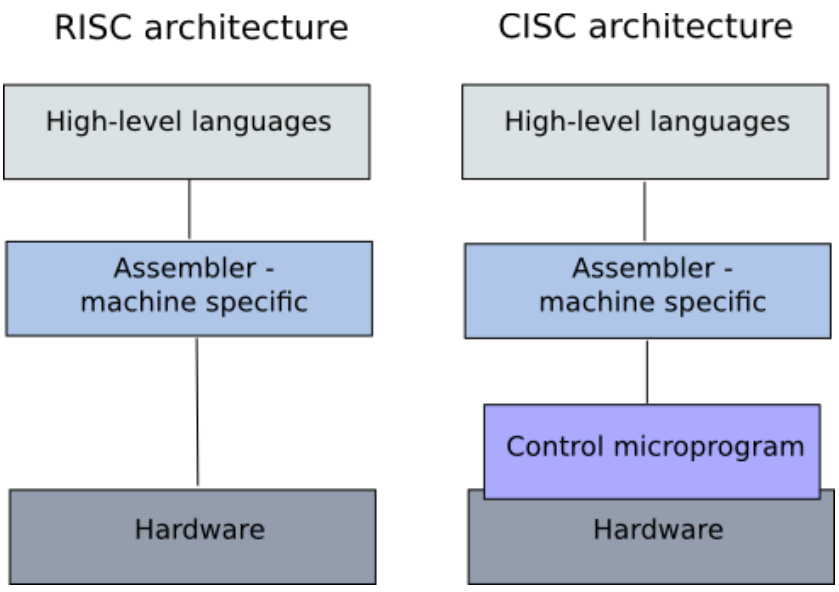

Fig 1.

1.3 RISC vs CISC

Regula 80-20: 80% din timp se folosesc 20% din instrucţiunile unui procesor. Multe instrucţiuni CISC sunt nefolosite de către programatori, iar majoritatea instrucţiunilor complexe pot fi sparte în mai multe instrucţiuni simple.

Performanța:

$$\frac{timp}{program} = \frac{timp}{ciclu} * \frac{instructiuni}{program} * \frac{cicli}{instructiune}$$

Ambele abordări fac compromisuri pentru a minimiza una dintre aceste fracţii: în cazul RISC, se preferă un număr mai mare de instrucţiuni pe program pentru a micşora numărul de cicli pe instrucţiune, în timp ce la CISC este invers.
Totuşi, lucrurile nu stau atât de simplu. Deşi arhitecturile RISC au fost devansate în anii 80 de către CISC datorită lipsei software-ului şi al unui segment de piaţă, precum şi datorită preţului ridicat al memoriei, o dată cu progresele tehnologice, procesoarele cu ISA RISC au monopolizat domeniul embedded. În plus, cele două arhitecturi au început să se apropie, majoritatea procesoarelor actuale având o arhitectura hibridă, de exemplu Intel Pentium având un interpretor CISC peste un nucleu RISC.

RISC	CISC
costuri mici de fabricatie, mai puține tranzistoare	hardware mai complex, costuri mai mari
consum redus de energie	consum mai mare
mai multe registre	mai mult hardware, controlul microprogramului
mai multe linii de cod per program	puţin cod

2. Paralelism - Pipelining

Chiar dacă un program este în mod tradițional interpretat secvenţial, instrucţiune cu instrucţiune, o parte dintre acestea nu sunt dependente între ele şi pot fi executate în paralel. Această metodă de a extrage paralelism la nivelul unui singur flux de instrucţiuni de numeşte paralelism la nivel de instrucţiunie sau Instruction-level parallelism (ILP).

O metodă de a implementa ILP este banda de asamblare (pipeline). Banda de asamblare presupune că atunci când o instrucțiune *i* este executată, instrucţiunea *i+1* este decodificată şi instrucţiunea *i+2* este citită, astfel fiecare subsistem din procesor este ocupat la un moment dat.

Modelul de bază pentru un pipeline la o arhitectura RISC este cel din 4 etape (Fig. 2): Instruction Fetch, Instruction Decode, Execute, Register Write Back. În stagiul Fetch se aduce instrucţiunea din memorie, de la adresa dată de contorul program (PC - *program counter*). În stagiul Decode se decodifică instrucţiunea, în Execute se execută efectiv operaţia de către unitatea aritmetico-logica, iar în Writeback se stochează rezultatele în regiştrii sau memorie. Pentru CISC, avem de obicei un număr mai mare de etape (>12).

Banda de asamblare ***superpipeline*** este mai lungă decât banda obişnuită (cele 4 stagii F, D, E, W sunt împărţite la rândul lor în mai multe stagii, Fig. 3). Teoretic cu cât banda de asamblare este mai segmentată, mai multe instrucţiuni pot fi executate în paralel şi, fiecare stagiu făcând mai puţine operaţii, se poate mări frecvenţa ceasului. Însă, alţi factori caracteristici ai design-ului pot scădea performanţele şi nu întotdeauna un procesor cu superpipelining este mai bun decât unul cu bandă de asamblare normală. Pentru o arhitectură şi un set de instrucţiuni date există un număr optim de etape ale benzii de asamblare. Peste acest număr se reduce performanţa. Pentru procesoarele RISC, unde instrucţiunile se execută într-un singur ceas, această durată de execuţie este dată de instrucţiunea cea mai lungă.
Banda de asamblare mărește productivitatea (throughput-ul), adică numărul de instrucţiuni executate în unitatea de timp, însa timpul de execuţie (latenţa) a unei instrucţiuni nu se micşorează, ci se poate chiar mări.

Pentru a paraleliza şi mai mult execuţia instrucţiunilor, procesoarele ***superscalare*** introduc benzi de asamblare independente ce funcţionează în paralel. Se măreşte astfel numărul de resurse şi complexitatea procesorului, însă mai multe instrucţiuni pot fi aduse, decodificate ţi executate în acelaţi timp, după cum este ilustrat şi în Fig. 4.

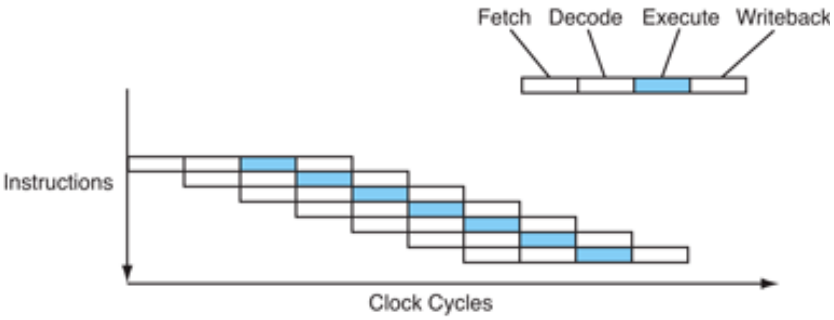

Fig 2. Execuţia instrucţiunilor în banda de asamblare.

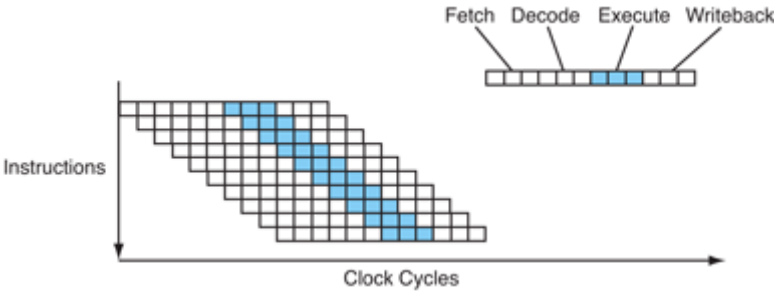

Fig 3. Execuţia instrucţiunilor în superpipeline

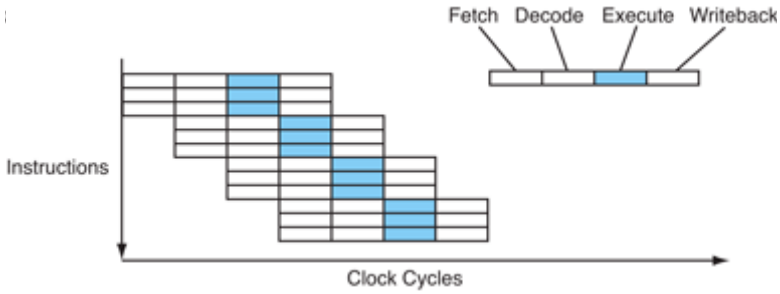

Fig 4. Execuţia instrucţiunilor într-un procesor superscalar

Sursa imaginilor este aici.

3. Hazarde

În cazul unui procesor cu bandă de asamblare, pentru a obţine performanţă maximă, este necesar ca o instrucţiune să fie prezentă în fiecare stagiu al pipeline-ului. Acest lucru nu este posibil însă datorită faptului că instrucţiunile pot avea diferite dependenţe între ele ceea ce face ca execuţia lor în pipeline să ducă la rezultate nedorite, situaţie care poartă numele de hazard.

3.1 Tipuri

Există trei tipuri de dependenţe care pot conduce la hazarde:

1. **Structurale**: când două instrucţiuni aflate în execuţie în pipeline doresc acces la aceeaşi structură hardware în acelaşi timp.
Exemplu:

```
add   r1, r2
load  r3, [r4 + offset]
```

- ambele instrucţiuni necesită acces la unitatea de adunare: prima pentru a executa o adunare, iar a doua pentru a calcula adresa de la care se face citirea din memorie.

2. **De date**: când două instrucţiuni folosesc acelaşi registru sau aceeaşi locaţie de memorie şi cel puţin una dintre ele este o scriere

• **RAW (Read after Write):** instrucțiunea I_{i+1} citeşte o locaţie de memorie scrisă de instrucţiunea I_i
Exemplu:

```
add r1, r2
add r3, r1
```

- valoarea lui `r1` este disponibilă în a doua instrucţiune abia după ce prima instrucţiune a scris rezultatul

• **WAR (Write after Read)**: instrucțiunea I_{i+1} scrie o locaţie de memorie care este citită de instrucţiunea I_i
Exemplu:

```
add r2, r1
add r1, r3
```

- valoarea lui `r1` nu poate fi modificată în a doua instrucţiune până când prima instrucţiune nu a terminat citirea lui

• **WAW (Write after Write):** instrucțiunea I_{i+1} scrie o locaţie de memorie care este scrisă şi de instrucţiunea I_i

Exemplu:

```
mov r1, r2
mov r1, r3
```

- este necesar ca cele două scrieri în `r1` să se execute în ordinea din program pentru a nu modifica valoarea văzută de instrucţiunile următoare

3. **De control**: când o instrucţiune de salt (branch, jump, call, ret) este executată; instrucţiunile de salt pot modifica PC (contorul program), sărind la o altă instrucţiune decât cea imediat următoare; în acest caz adresa următoarei instrucţiuni care trebuie citită nu se cunoaşte până când instrucţiunea de salt nu este executată

Exemplu:

```
if (a == b)           (i1)
     d = a+b          (i2)
     c = 1            (i3)
else
     d = a * b        (i4)
```

În banda de asamblare se aduc din memorie la rand instrucțiunile i1,i2,i3, insa daca in urma executiei instrucțiunii *if* (tradusa in assembler printr-o instrucțiune de comparare si apoi una de salt conditionat (jmp - jump) se sare la ramura de else apare un hazard de control. Instrucțiunile i2,i3 nu mai trebuiesc executate, și în schimb trebuie adusă instrucțiunea i4, deci se face 'flush' la pipeline și se aduce instrucțiune i4, pierzandu-se astfel din eficiența prelucrării în paralel a instrucțiunilor.

3.2 Soluții

- O soluţie simplă ar fi adăugarea de către programator sau compilator sau automat, de către UC a procesorului, a unor instrucţiuni `nop` pentru întârzierea execuţiei instrucţiunilor cu dependinţe; aşa numitele *pipeline bubbles* sau *pipeline stalls.*
- ***Forwarding*** - dacă unul sau mai multi operanzi ai unei instructiuni depind de o instructiune aflata înainte in pipeline, rezultatul acesteia poate fi forwardat catre stagiile anterioare. Aceasta soluție necesită hardware suplimentar pentru logica de control.
  Exemplu:
  ```
  instr k:        add r1,r2
  instr k+1:      add r1,5
  ```
  Rezultatul obtinut pentru instrucțiunea k în unitatea aritmetico-logică, în stagiul de Execute este forwardat stagiului Decode al instructiunii k+1.
- ***Out-of-order execution(OOE)*** este o tehnica de planificare dinamica(dynamic scheduling) prin care instrucțiunile sunt executate în altă ordine decât cea a programului, însă rezultatele sunt ordonate ca si cum ar fi fost executate normal. Prin această rearanjare, se evită adăugarea de întârzieri, intrând în pipeline instrucțiuni fără dependențe de date între ele. Aceasta soluție necesită hardware suplimentar(registre suplimentare, buffers pentru cozi de așteptare etc), iar algoritmul cel mai cunoscut este cel al lui Tomasulo .

- Deoarece registrele generale sunt limitate, iar codul unui program îi refolosește des, generând hazarde de date, putem folosi tehnica de ***register renaming.*** Aceasta minimizează apariția hazardelor, prin adaugarea unor registre suplimentare, ce nu sunt vizibile programatorului. Redenumirea registrelor rezolvă hazardele de date WAR, WAW însa nu si RAW, fiind folosită și în algoritmul Tomasulo de planificare OOE.
  Exemplu:
  ```
  instr k    add r1,r2,r3
  instr k+1  mul r3,r2,r1
  instr k+2  sub r1,r2,r4  //hazard WAW intre instr k si k+2
  ```
  După redenumirea registrelor:
  ```
  add l1, l2, l3
  mul l5, l2, l4
  add l6, l2, l7
  ```

- În cazul hazardelor de control, putem amâna aducerea instrucțiunilor din branch prin adăugarea de întarzieri în pipeline sau intercalarea altor instrucțiuni independente de acel branch, însă soluția cea mai bună este predicția ramificatiilor (**branch prediction**) care minimieaza numărul de cicli pierduți. Această predicție poate fi statică, facută de către compilator prin optimizari de tipul *loop unrolling* (de exemplu in bucla *for,* a cărei conditie este neîndeplinită doar o dată pentru a se ieși din buclă) sau profile-driven(se creează profile ale programelor în urma mai multor execuții). Predicția dinamică se face de catre procesor prin intermediul unor circuite (*branch predictors*) și în general se bazează pe menținerea unor tabele cu informații despre fiecare dintre ramificații.

Referințe

Patterson, David A.; Hennessey, John L, *Computer Architecture: A Quantitative Approach*, 4th Edition
RISC vs CISC
RISC vs CISC wars
Pipeline, IPL, Hazarde