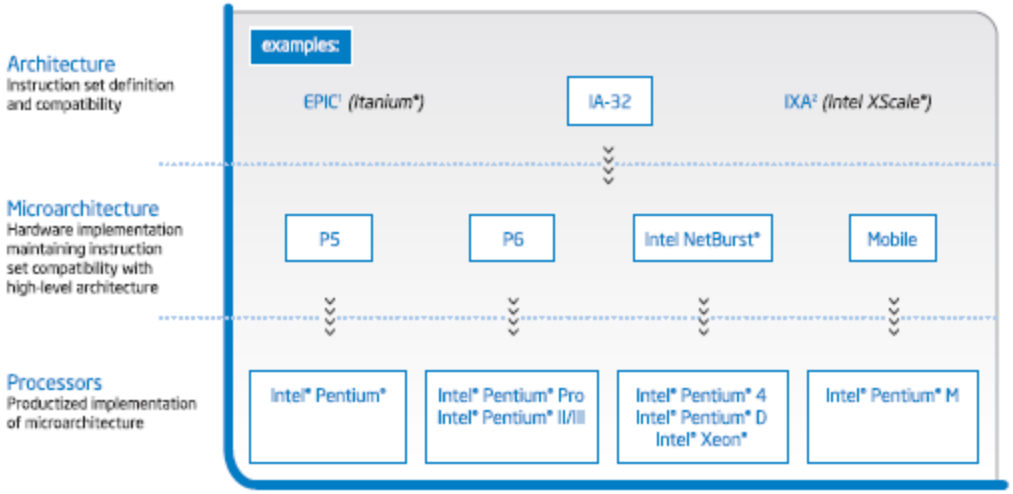
Curs 12

9. Microprocesoare moderne

9.1. Proiectarea microprocesoarelor

9.1.1. Arhitectura și structura microprocesoarelor

- Arhitectură vs. microarhitectură
  - Arhitectură: setul de instrucțiuni, registrele, structura datelor în memorie; sunt informații accesibile programatorului;
  - Microarhitectură: implementarea arhitecturii pe suportul fizic; este îmbunătățită continuu;



Proiectarea cu Microprocesoare

- Structura microprocesoarelor

  - Von Neumann
    - Codul și datele sunt reprezentate în memorie la fel și sunt accesate la fel;
    - Se folosesc aceleași magistrale; rezultă simplitate;
    - Întrucât toate accesele se fac la aceeași memorie și memoria este mai lentă ca procesorul (la toate nivelele tehnologice), aceasta va frâna execuția programului;

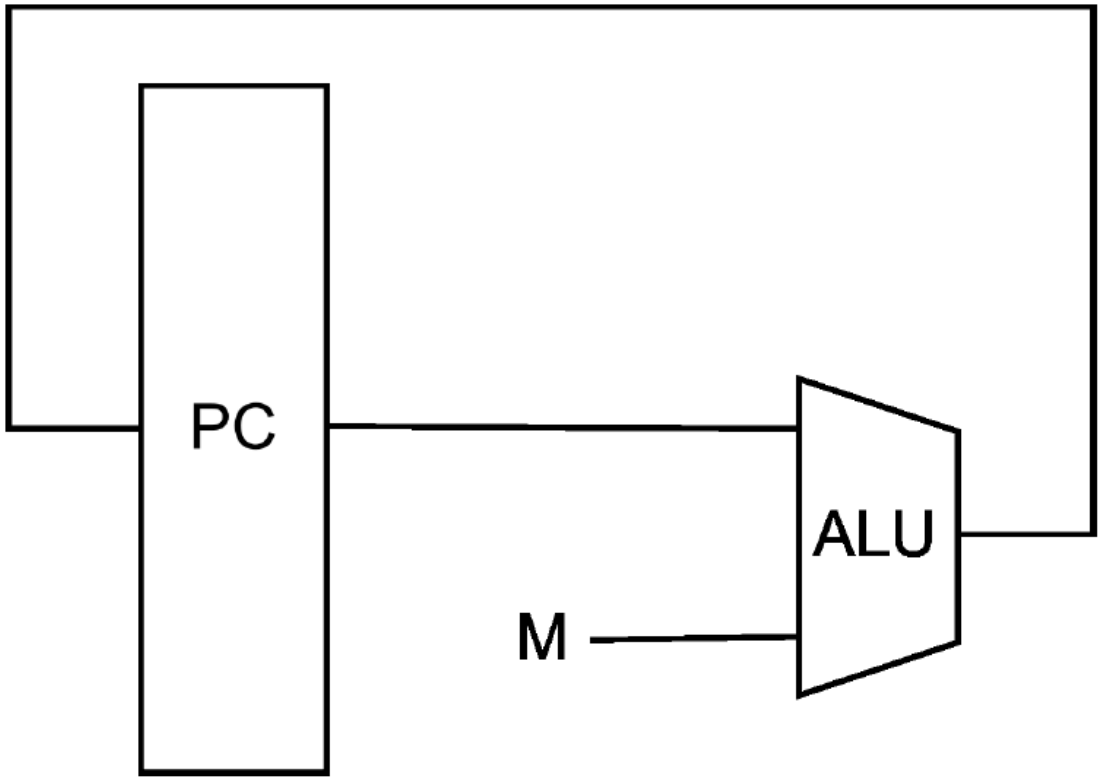
  - Harvard
    - Codul și datele sunt păstrate în memorii diferite; există 2 spații de adrese;
    - Accesul la cod și date se face prin magistrale diferite; rezultă viteză mare dar scheme mai complicate;
    - Structura Princetown: un singur spațiu de adrese dar 2 blocuri de memorii;

9.1.2. Arhitectura setului de instrucțiuni

- CISC (Complex Instruction Set Computer)
  - Instrucțiuni complexe, lungi, cu multe moduri de adresare și cu durată mare de execuție;
  - Programarea este mai simplă, compilatorul este complex, viteză redusă;
  - Exemple: arhitectura Intel x86, Motorola 68k;
- RISC (Reduced Instruction Set Computer)
  - Instrucțiuni simple, scurte, cu durată mică de execuție dar și cu efect redus;
  - Număr redus de moduri de adresare, instrucțiuni pe 1 sau puține cicluri, compilatoare mai simple, viteză mai mare;
  - Exemple: PowerPC, ARM, SPARC;
- VLIW (Very Long Instruction Word)
  - Instrucțiuni lungi, pe mai multe cuvinte;
  - Dacă procesorul este multicore, se pot grupa mai multe instrucțiuni/core, pentru a rezulta o instrucțiune lungă;

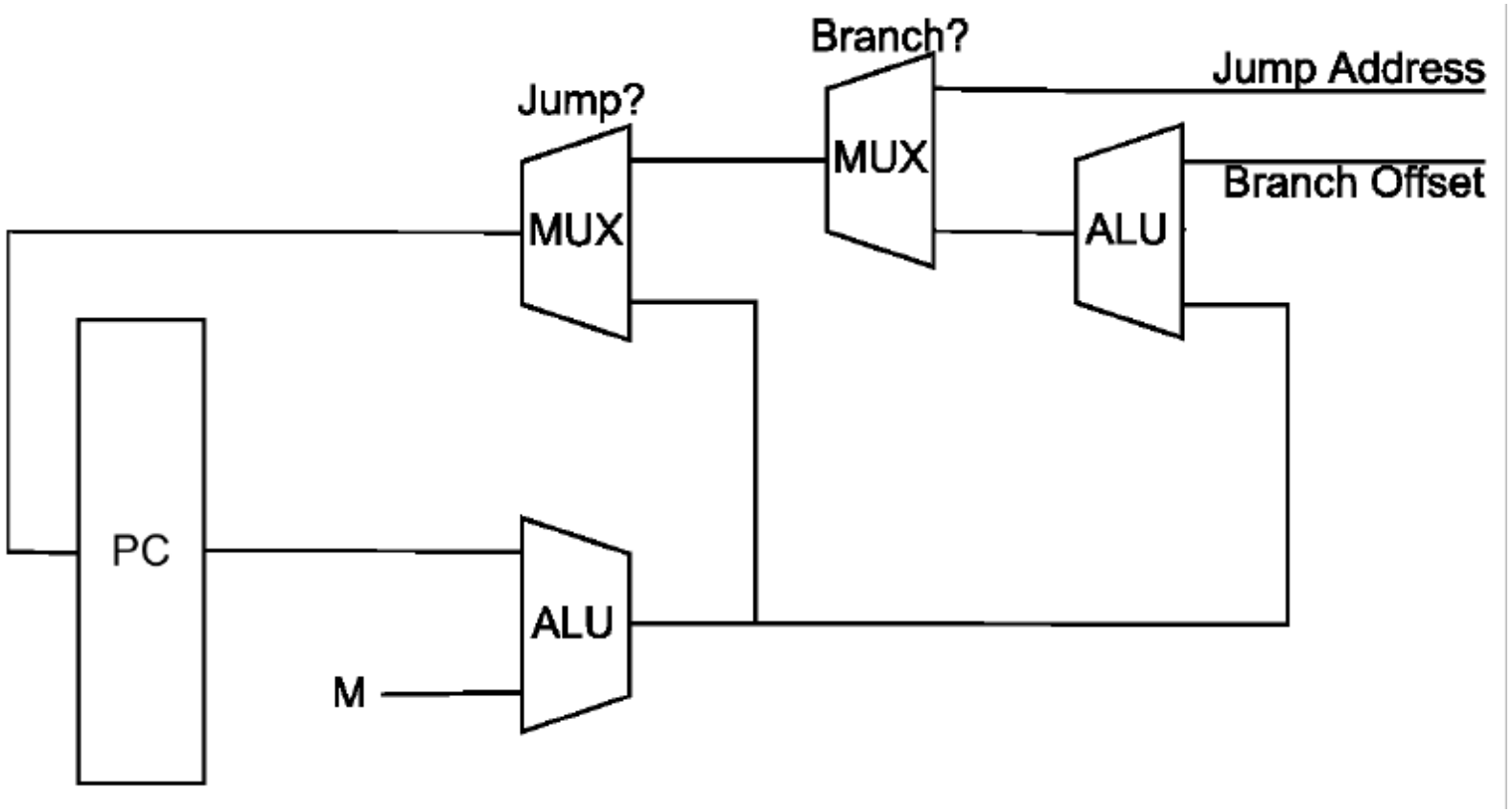
9.1.3. Componente

- RI, DCI, REG (PC), DCC, UAL, FPU
- PC

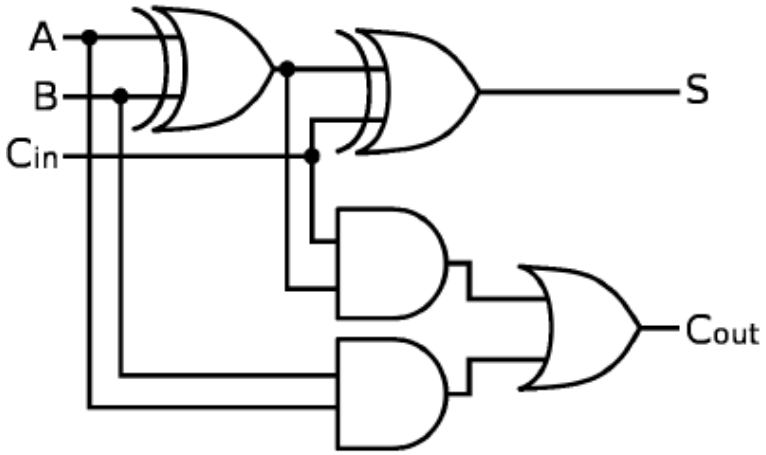


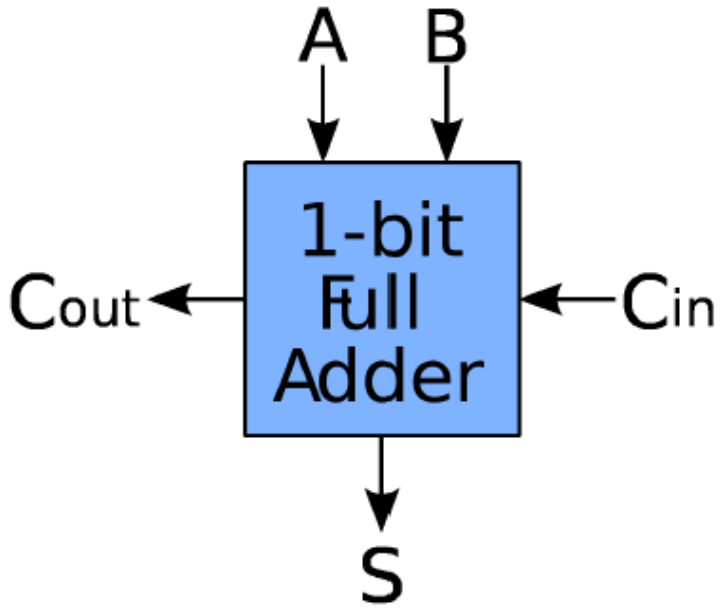
Proiectarea cu Microprocesoare

- PC complex:



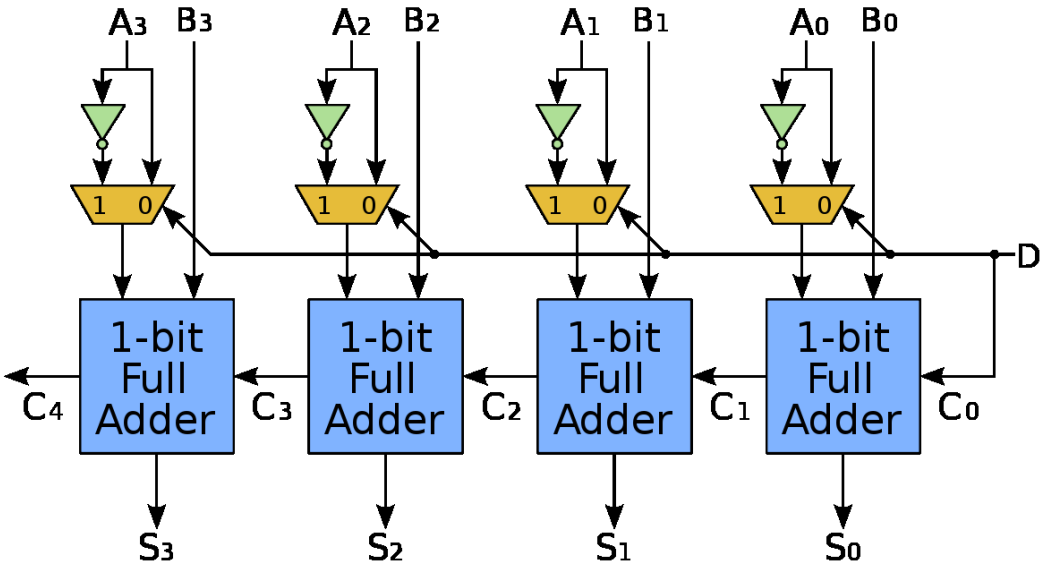
- UAL
  - Operații aritmetice;
  - Operații logice;
  - Configurații;
  - Operații aritmetice: adunare, scădere, înmulțire, împărțire
    - Sumator complet pe 1 rang:



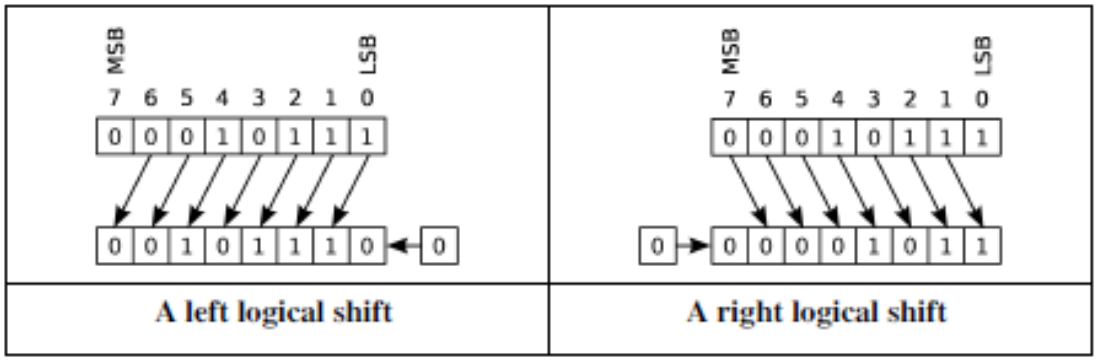


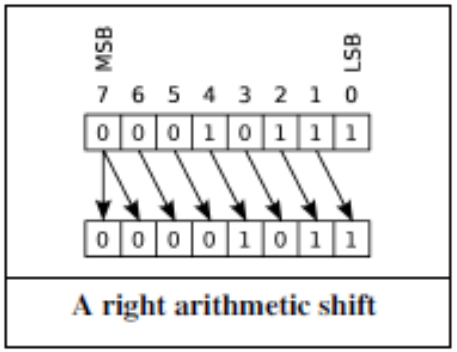
Proiectarea cu Microprocesoare

- Sumator/ scăzător paralel:



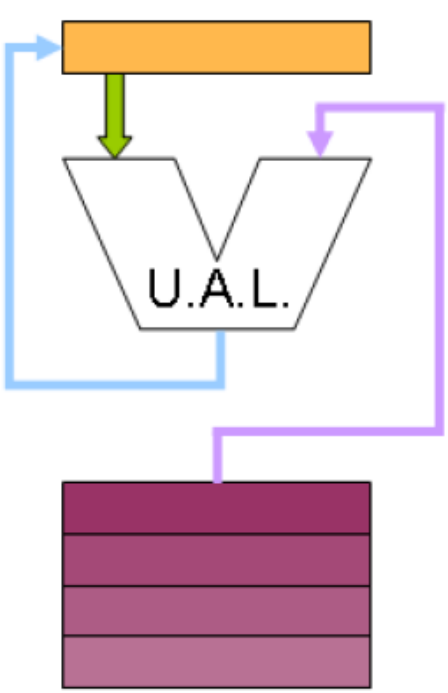
- Operații logice: deplasare logică, deplasare aritmetică, rotire:

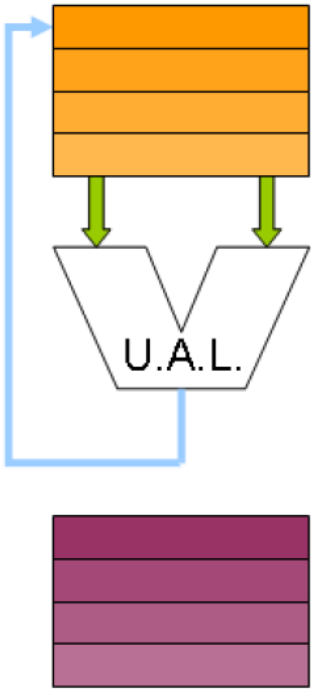


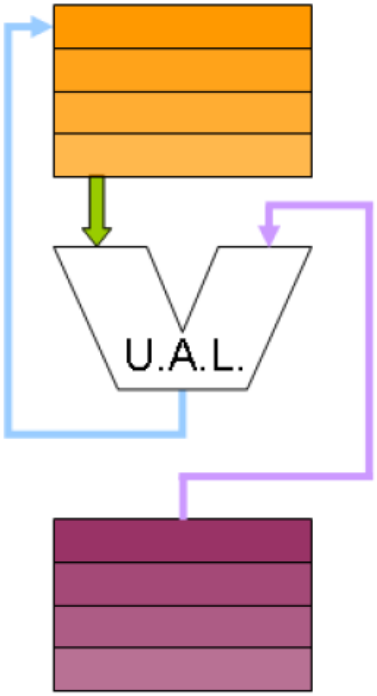


Proiectarea cu Microprocesoare

- Configurații:
  - Cu acumulator: un operand este în acumulator iar rezultatul se depune în acumulator; instrucțiuni mai scurte;
  - Registru – registru: instrucțiunea va cuprinde câmpuri pentru specificarea operanzilor; câmpurile sunt scurte;
  - Registru – memorie: instrucțiuni lungi întrucât trebuie să conțină și adrese;







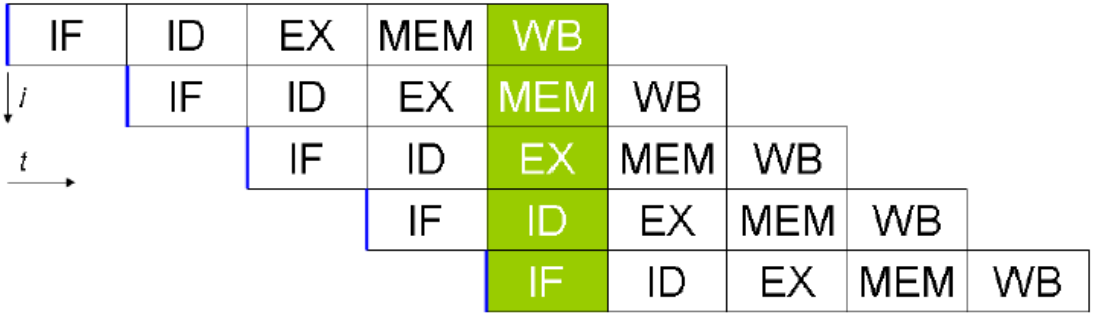
9.1.4. Execuția instrucțiunilor

- 1 ciclu instrucțiune = 5 cicluri mașină:
  - IF: aducere cod de instrucțiune (instruction fetch);
  - ID: decodificare cod de instrucțiune (instruction decode);
  - EX: generare semnale de comandă și control (execution);
  - MEM: citire operanzi;
  - WB: scriere rezultat;
- Procesoare cu execuție secvențială:

  - Hardware ineficient folosit, viteză mică;
  - Simplitate, ușor de stabilit durata secvențelor de cod;

Proiectarea cu Microprocesoare

- Procesoare cu bandă de asamblare (pipeline):

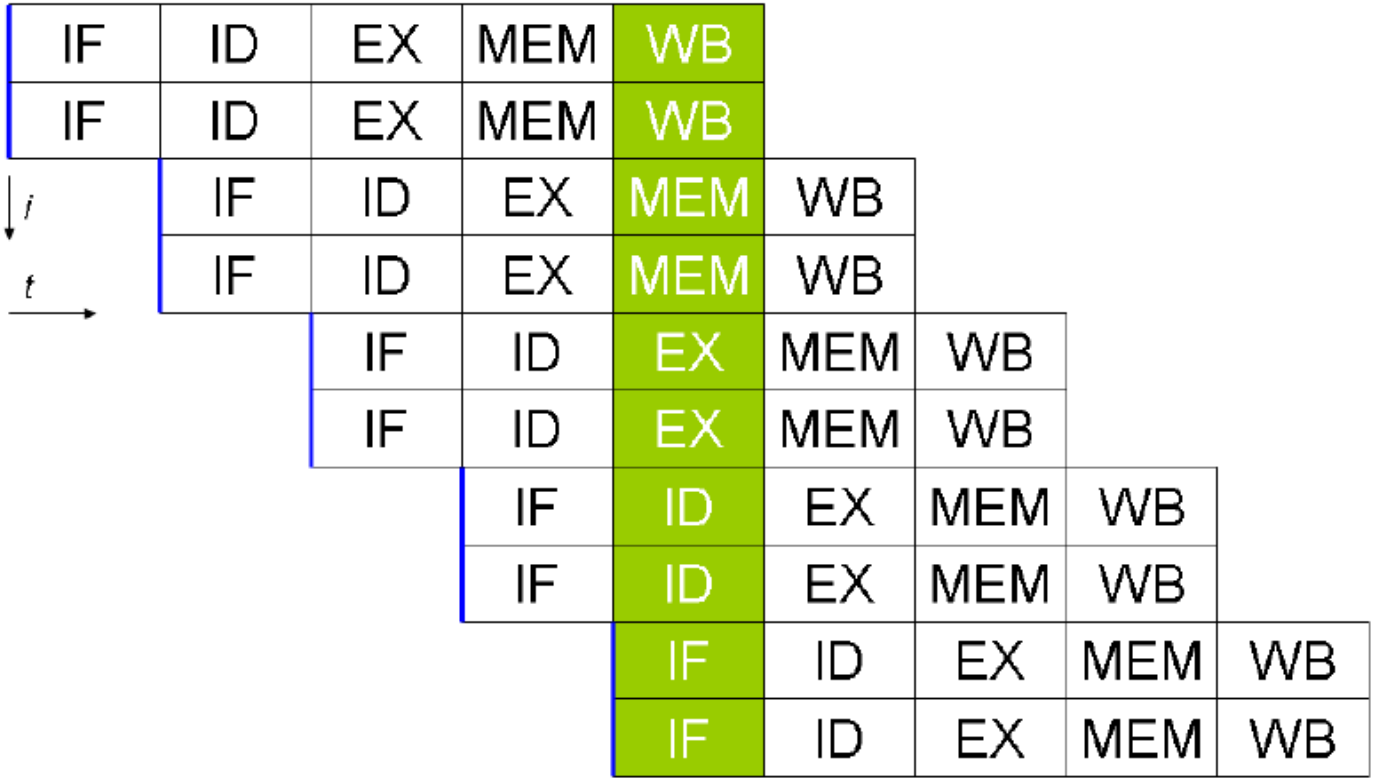


  - Rata de execuție crește de 5 ori (banda de asamblare are 5 nivele);
  - Hardware eficient folosit; hardware complex;
  - Latența este dată de cel mai lent nivel

- Procesoare cu superpipline:
  - Nivelele mai lente sunt divizate în nivele cu latență mai mică;
  - Hardware complex;

Proiectarea cu Microprocesoare

- Procesoare superscalare:



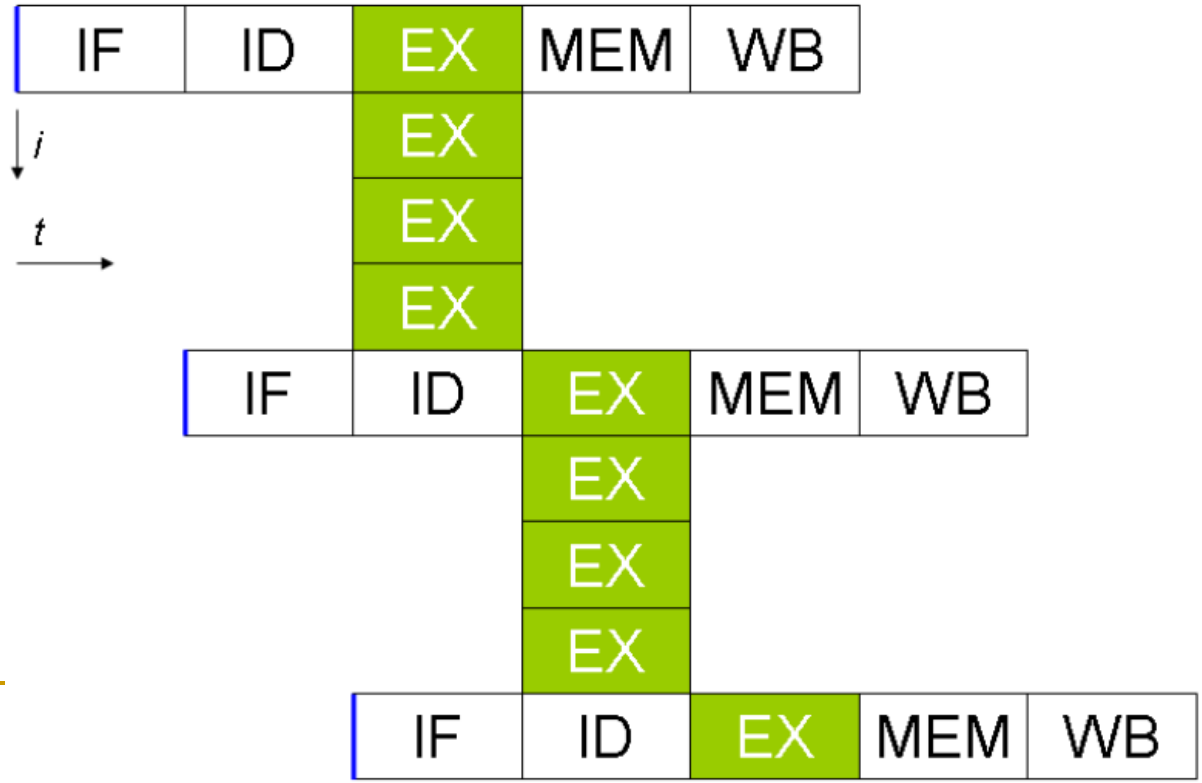
  - Au un singur nucleu dar mai multe resurse de execuție (multiple ALU, multiple FPU etc.);
  - Execută simultan mai multe instrucțiuni;

Proiectarea cu Microprocesoare

- Hyperthreading:
  - Tehnologie descoperită și aplicată de Intel la mai multe procesoare, apărute după Pentium 4; ex.: Intel Core processor, Intel Core M processor, Intel Xeon processor etc.
  - Presupune multiplicarea unor unități din interiorul microprocesorului folosite la execuția unui fir, ca registre, indicatori etc. dar nu și nucleul procesorului;
  - Se vor crea căi distincte pentru execuția mai multor fire, rădăcina lor fiind comună, și anume nucleul procesorului; acesta va fi multiplexat între diferitele fire de execuție; multiplexarea se face prin hardware;
  - Rezultă execuția aparent simultană a mai multor fire; la un moment dat, nucleul procesorului execută un singur fir.

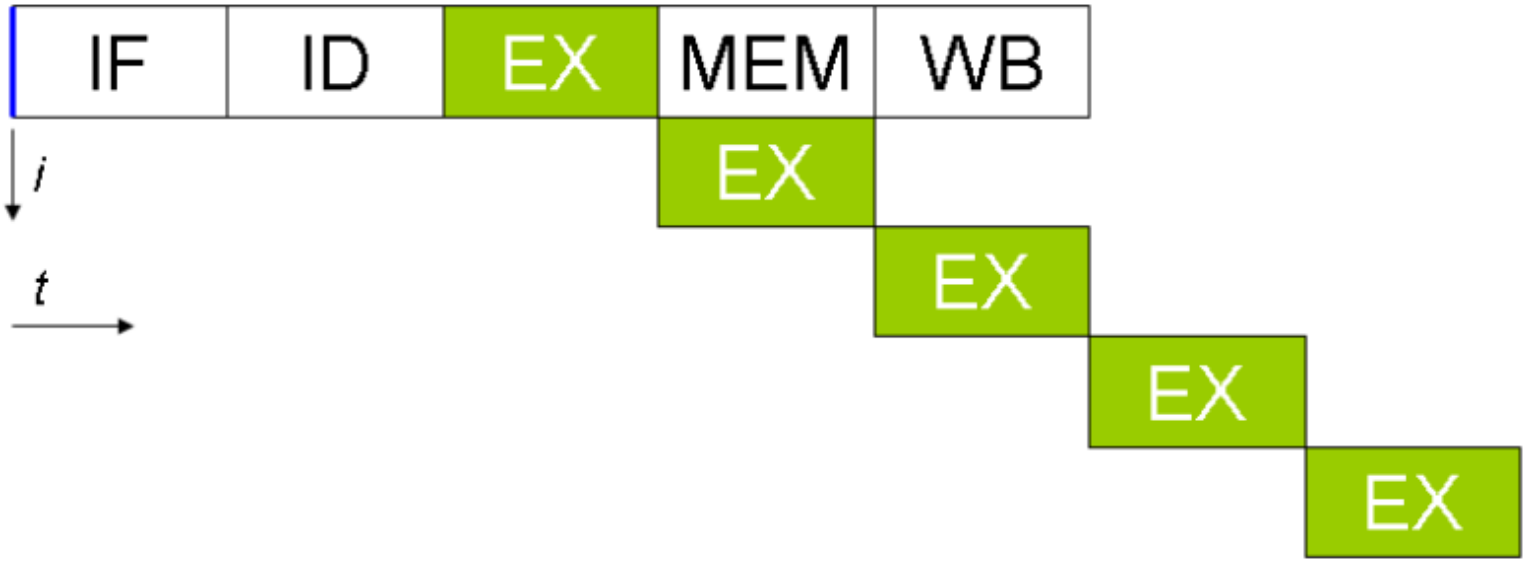
Proiectarea cu Microprocesoare

- Procesoare VLIW (Very Long Instruction Word)
  - La procesoarele scalare un planificator hardware decide ce instrucțiune este executată, de care nucleu și când;
  - La procesoarele VLIW fiecare nucleu primește o instrucțiune de executat la fiecare cuvânt; planificarea este făcută de compilator;
- Procesoare vectoriale



- Fac aceeași operație asupra mai multor seturi de date;
- Numite SIMD în clasificarea Flynn;
- Se presupune existența a mai multor UAL;
- Procesoarele grafice; la familia Intel, facilitățile MMX și SSE;

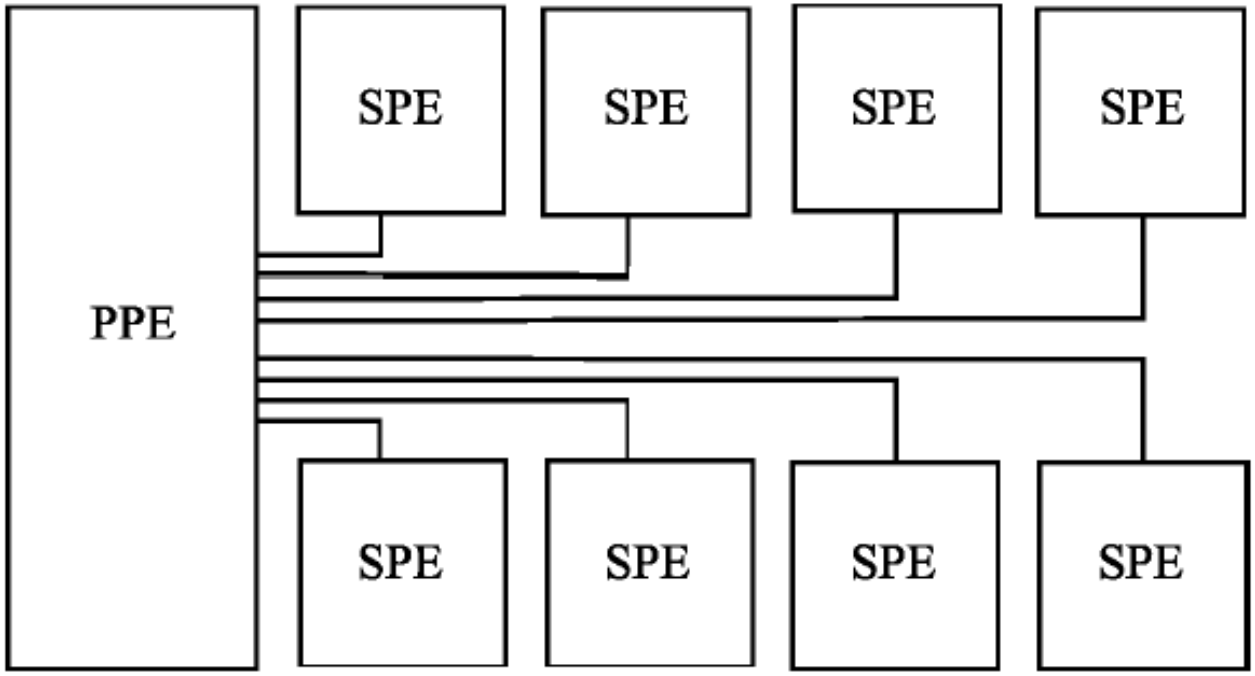
- Procesoare vectoriale cu execuție serială:



- Se refolosește un singur UAL;

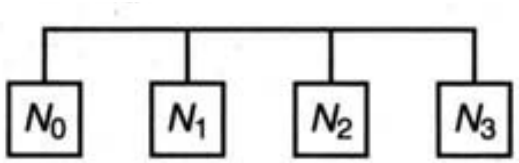
Proiectarea cu Microprocesoare

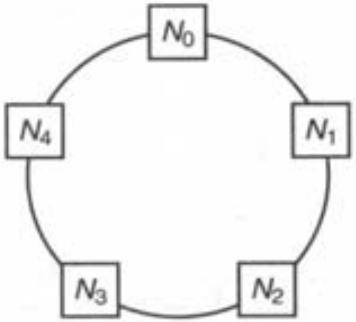
- Procesoare multicore
  - Procesoare cu mai multe nuclee independente
    - Simetrice: nuclee identice; ex.: Intel Core 2 (Core 2 Duo cu 2 nuclee, Core 2 Quad cu 4 nuclee);
    - Asimetrice: nuclee diferite; ex. IBM Cell processor: 1 nucleu se ocupă de comunicare iar celelalte 8 de operații aritmetice în virgulă flotantă, cu mare viteză;

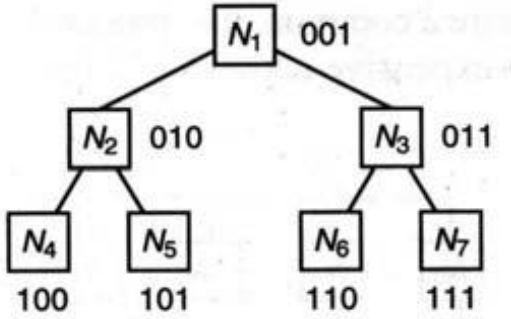


Proiectarea cu Microprocesoare

- Procesoare manycore
  - Mai multe nuclee (zeci, sute, mii) ca procesoarele multicore (unități, zeci);
  - Optimizate pentru procesare masiv paralelă, cu latență mică la nivelul procesării paralele dar latență mai mare la nivelul procesării mononucleu; procesoarele multicore optimizează latența la nivelul procesării mononucleu;
- SoC and NoC:
  - Mai multe procesoare legate prin rețele de interconectare;
  - Rețele de interconectare:







9.1.5. Probleme în execuția instrucțiunilor

- Întreruperile;
- Hazardul
  - Hazardul de date; la accesul datelor:
    - Read after write (RAW): citirea începe înaintea încheierii scrierii; dacă scrierea durează prea mult, la citire se va obține o dată incorectă;
    - Write after read (WAR): scrierea începe înaintea încheierii citirii; citirea va genera o dată incorectă;
    - Write after write (WAW): 2 scrieri au loc simultan la aceeași locație; dacă nu sunt decalate, una din date se va pierde;
  - Hazardul de control:
    - Atunci când intră în banda de asamblare o instrucțiune de salt condiționat; până la procesarea ei și aflarea ramurii corecte, pot fi executate instrucțiuni inutile;
  - Hazardul structural:
    - 2 instrucțiuni accesează simultan aceeași resursă hardware;
  - Hazardul electronic.

Proiectarea cu Microprocesoare

- Tratarea hazardului:
  - Furtul de cicluri (Stall): atunci când unitatea de control sesizează un hazard va opri aducerea instrucțiunilor și va insera în banda de asamblare instrucțiuni NOP; astfel, instrucțiunile cu hazard vor fi executate singure fără a afecta restul codului;
  - Avansul datelor (Forwarding): atunci când rezultatul unei instrucțiuni trebuie folosit ca un operand al următoarei instrucțiuni, data obținută la ieșirea din UAL în instrucțiunea curentă va fi avansată la intrarea în UAL corespunzătoare instrucțiunii următoare; hazardul va fi evitat dar este necesară o unitate de avansare a datelor;
  - Redenumirea registrelor (Register renaming): unui registru îi pot corespunde mai multe suporturi fizice; fie instrucțiunea: add R1, R2, R1, cu execuția (R1) + (R2) → R1; R1 poate avea ca suport fizic un registru și o locație de memorie; operandul sursă va fi luat din registru iar rezultatul se va depune în memorie; se va evita hazardul RAW sau WAR;
  - O proiectare care să evite tranzițiile între 2 stări care diferă prin mai mult de 1 rang; se va evita hazardul electronic.

- Creșterea eficienței benzii de asamblare
  - Scade la apariția instrucțiunilor care rup secvențialitatea unui program: salturi, apeluri, întreruperi; pierderea este mare la apariția instrucțiunilor de salt sau apel condiționat;
  - Soluții:
    - Execuția speculativă: se ghicește ramura unui salt/apel condiționat care va fi continuată; în gazul ghicirii eronate, banda trebuie golită; în cazul ghicirii corecte, nu se pierde timp;
    - Întârzierea execuției sau furtul de ciclu: după instrucțiunea de salt/apel condiționat se plasează o instrucțiune care va fi executată indiferent de ramura care va continua; dacă nu sunt dependențe de date, această instrucțiune poate fi NOP; se câștigă timp pentru ca instrucțiunea de salt/apel condiționat să stabilească ramura pe care va continua;
    - Predicția ramurii: se ghicește ramura care se va executa dar nu aleator ci bazat pe anumite considerente, de obicei istoria salturilor;
      - Ex.: execuția unei bucle; o buclă este executată de mai multe ori înaintea terminării ei, însemnând că au loc mai multe ramificări în o direcție și doar o singură ramificare în cealaltă direcție; fie secvența în pseudocod HLL:

while (`condition`)

executa

end

Transformarea în pseudocod limbaj de asamblare:

start loop:

compare `condition` to 0

branch to *end loop* if equal

executa

branch to *start loop*

end loop:

Ramificarea la *start loop* se va executa de multe ori iar la *end loop* o singură dată ca atare este foarte probabil ca ramificarea curentă să fie executată în același loc.

- Optimizarea buclelor: fie secvența în pseudocod HLL:

while (`loop condition`)

if (`branch condition`)

executa 1

else

executa 2

end

Se știe, statistic, că *branch condition* este fals, 0, 90% din timp iar *loop condition* va fi adevărat, 1, 100% din timp;
Transformarea în pseudocod limbaj de asamblare:

start loop:
1) compare `loop condition` and 0
2) branch to *end loop* if equal
3) compare `branch condition` and 0
4) branch to *branch true* if not equal
5) executa 2
6) branch to *end if*
7) branch true
8) executa 1
9) end if
10) branch to *start loop*
end loop:

Se observă că ramificarea la linia 10 se va executa în majoritatea timpului iar ramificarea la linia 4 va apărea doar dacă `branch condition` este 1. `branch condition` este 1, însă, doar 10% din timp ca atare predicția va fi ineficientă (doar 10% din cazuri).

Proiectarea cu Microprocesoare

- O variantă mai eficientă este:

while (`loop condition`)
    if (`not branch condition`)
        executa 1
    else
        executa 2
end

Transformarea în pseudocod limbaj de asamblare:

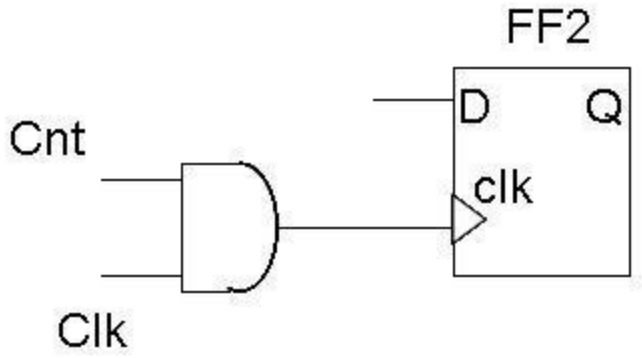
start loop:
1) compare `loop condition` and 0
2) branch to *end loop* if equal
3) compare `branch condition` and 1
4) branch to *branch false* if not equal
5) executa 1
6) branch to *end if*
7) branch false
8) executa 1
9) end if
10) branch to *start loop*
end loop

Se observă că ramificarea la linia 10 se va executa în majoritatea timpului iar ramificarea la linia 4 va apărea dacă `branch condition` este 0. `branch condition` este 0 în 90% din timp ca atare predicția va fi eficientă (în 90% din cazuri). Predicția va fi reușită în 90% din cazuri.

- Un predictor poate funcționa ca un numărător: dacă ramificarea este executată numărătorul este incrementat iar dacă nu este executată, numărătorul este decrementat; astfel se poate obține istoria ramificărilor.

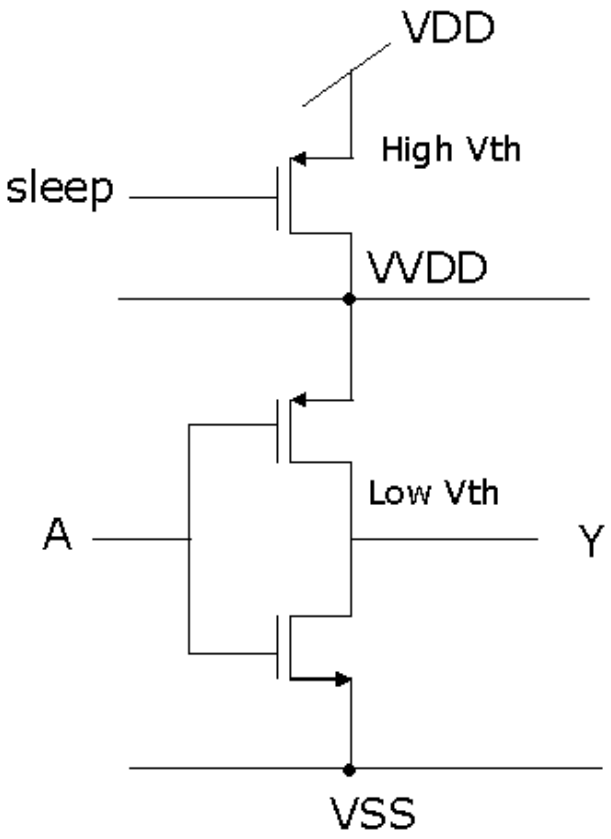
9.1.6. Frecvență vs. putere

- Creșterea frecvenței este limitată de soluțiile pentru scăderea puterii;
  - Clock-gating:



  - Există o validare a tactului care permite încărcarea informației în registre doar dacă datele s-au modificat; este economisită putere dar și limitată frecvența;

- Power-gating:



  - Tranzistorul de sus poate opri alimentarea etajului final, economisind putere dar introduce timpul necesar tranziției sale, limitând frecvența;