Universitatea de Vest din Timisoara, Facultatea de Matematica si Informatica

ARHITECTURA CALCULATOARELOR

Informatica, an I, 2021-2022

Dr. Mafteiu-Scai Liviu Octavian

Universitatea de Vest din Timisoara, Facultatea de Matematica si Informatica
ARHITECTURA CALCULATOARELOR, Informatica, an I, 2021-2022

CURS 11

1. Microarhitecturi – generalitati si proiectare
- exemplificare cu MIPS si MU0
2. Limbajul de asamblare pentru procesoare ARM

Dr. Mafteiu-Scai Liviu Octavian

Microarhitectura =

conexiunea dintre logica (matematica) si arhitectura (hardware).

Microarhitectura =

un anume aranjament de registrii, unitati aritmetico-logice (ALU) , masini cu stari finite (FSM), unitati de memorie si alte blocuri logice necesare pentru a alcatui o arhitectura.

Nota:

O arhitectura care ruleaza aceleasi programe poate fi descrisa prin mai multe microarhitecturi, fiecare din acestea fiind diferita de celelalte prin designul intern si evident prin performante si costuri.

In functie de modul de implementare al ISA (Instruction Set Architecture) avem:

- **RISC** (**R**educed **I**nstruction **S**et **C**omputing) – instructiunile pot fi executate intr-un singur ciclu procesor
- **CISC** (**C**omplex **I**nstruction **S**et **C**omputing) - instructiunile necesita mai multe cicluri procesor

Exemplu microarhitectura:

Mic-1, care implementeaza IJVM (Integer Java Virtual Machine).

Ce este JVM (Java Virtual Machine) ? – o masina virtuala care permite ca limbajul Java sa fie rulat pe orice masina (PC-uri, smartphones)

Contine:

- *un program mic (firmware in ROM) cu rol de a aduce, decoda si executa instructiunile IJVM*
- *variabile de stare ce pot fi accesate de catre programul din ROM*

Instructiunile din IJVM sunt foarte simple: cod operatie + operand

Microarhitectura MIPS (Microprocessor without Interlocked Pipeline Stages)

Arhitectura de tip RISC dezvoltata de catre MIPS Computer Systems, Inc., US, 1985

Initial 32 bits (MIPS32), actualmente 64 bits (MIPS64)

Arhitectura MIPS sta la baza sistemelor incapsulate precum dispozitive Windows CE, rutere, console de jocuri video (Nintendo 64, SonyPlayStation, PlayStation 2 si PlayStation Portable)

Mai mult de ½ din procesoarele RISC folosesc MIPS

Simplificat:

un procesor MIPS = 1 contor de program PC (Program Counter) **+ 32 registrii**

Ce este un registru procesor?

*un **registru de procesor** este un spatiu mic de stocare disponibil in CPU, spațiu al cărui conținut poate fi accesat mai rapid decât zonele de stocare din memoria principală.*

Microarhitectura MIPS

Pentru o mai usoara intelegere, consideram un set minimal de instructiuni:

- instructiunile aritmetico-logice: **add, sub, and, or, slt** (*Set on less than (signed))*
- instructiunile de lucru cu memoria: **lw** *(load word),* **sw** *(store word)*
- ramificatii/selectii/decizii: **beq** *(Branch on equal)*

Fiecare dintre acestea este detaliata in continuare:

ADD – adunare (cu depasire)

Aduna continutul a doi registrii si stocheaza rezultatul in al treilea registru: $d = $s + $t

Sintaxa: **add $d, $s, $t**

SUB - scadere

Scade continutul a doi registrii si stocheaza rezultatul in al treilea: $d=$s-$t;

Sintaxa: **sub $d, $s, $t**

Microarhitectura MIPS

AND – conjunctie logica pe biti

AND logic pe doi registrii si stocare rezultat in al treilea: $d = $s & $t;

Sintaxa: **and $d, $s, $t**

OR – disjunctie logica pe biti

$d = $s | $t;

Sintaxa: **or $d, $s, $t**

Nota1: operatiile **and** si **or** se fac pe perechi de biti corespondenti din cei doi operanzi

Nota2: Operatiile pe biti sunt mult mai rapide decat impartirea, de cateva ori mai rapide decat inmultirea si uneori mai rapide decat adunarea.

SLT – setare conditionala registru pe 1 (set on less than)**(signed)**

Daca $s este mai mic decat $t, $d este setat/pus pe 1. Genereaza zero in caz contrar

if $s < $t $d = 1 else $d = 0;

Sintaxa: **slt $d, $s, $t**

Microarhitectura MIPS

LW – incarcare cuvant (load word)

Un cuvant de la adresa *s* de memorie este incarcat intr-un registru *t*: $t = MEM[$s + offset];

Sintaxa: **lw $t, offset($s)**

SW – stocare cuvant in memorie (store word)

Continutul registrului *t* este stocat la adresa specificata *s*:MEM[$s + offset] = $t;

Sintaxa: sw $t, offset($s)

BEQ – ramificatie/salt in caz de egalitate continut registrii (branch on equal)

Ramificare/salt la o alta instructiune daca continutul celor doi registrii este egal, adica se trece la executia unui alt set de instructiuni aflati la adresa offset: if $s == $t (offset << 2)); else …..;

Sintaxa: **beq $s, $t, offset**

Branch se mai numesc si instructiuni de salt (oarecum echivalente cu saltul conditional in programarea de nivel inalt)

Proiectare microarhitectura MIPS

Pas 1: se imparte microarhitectura in:

- *datapath*: opereaza pe cuvinte de date
- *control*: primeste instructiunea curenta de pe datapath si ii spune acesteia (datapath) cum sa execute instructiunea

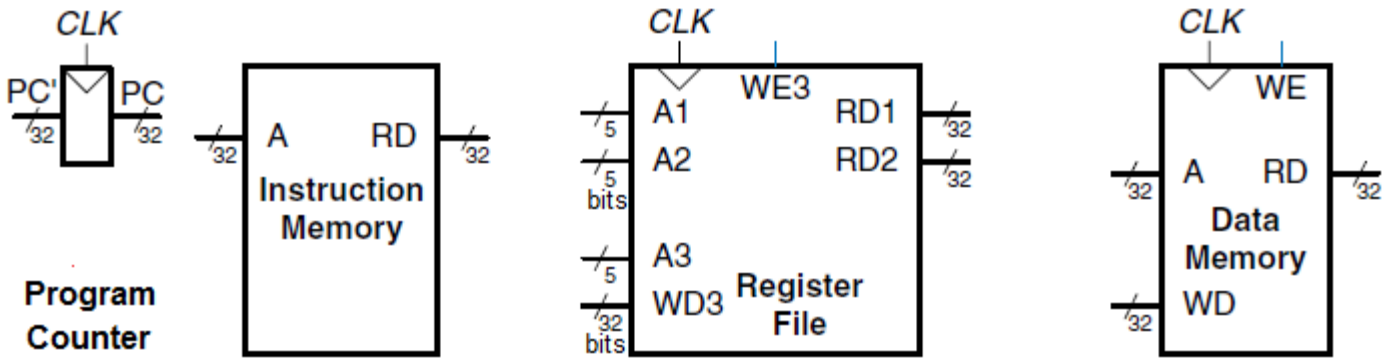
Pas 2: proiectare componenta hardware:

- memorii, contor program si registrii
- blocuri de logica combinationala: conecteaza intre ele elementele precedente

Exemplu →

Proiectare microarhitectura MIPS

Diagrama bloc: contor program + memorie instructiuni + registrii + memorie date



Program Counter:

- registru usual pe 32 biti
- PC: output, pointer ce indica instructiunea curenta (in executie)
- PC': input, pointer ce indica adresa instructiunii urmatoare

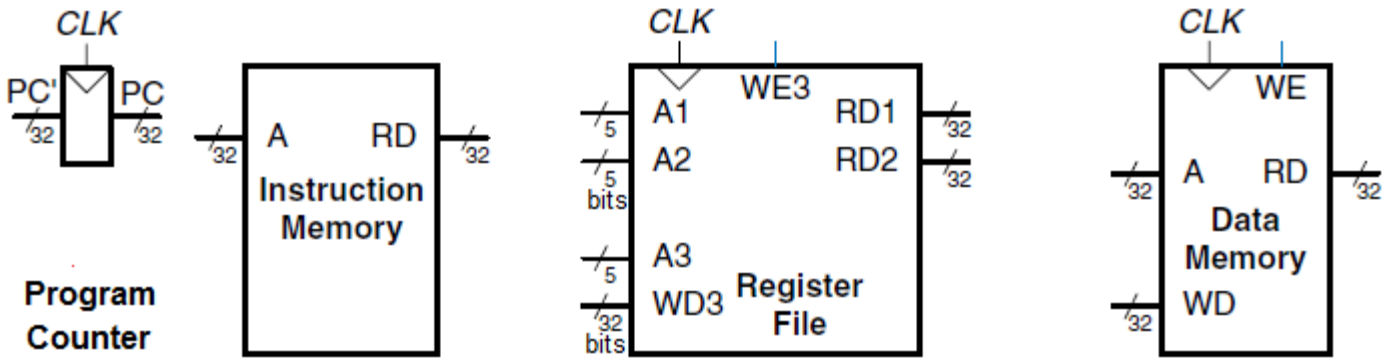
Instruction Memory:

- A, port de citire pe 32 biti
- RD, port de iesire pentru citire date

Blocul preia o adresa de instructiuni pe 32 de biti la portul A si citeste 32 biti de date (de ex. instructiuni) de la aceea adresa de citire date prin portul RD

Proiectare microarhitectura MIPS

Diagrama bloc: contor program + memorie instructiuni + registrii + memorie date



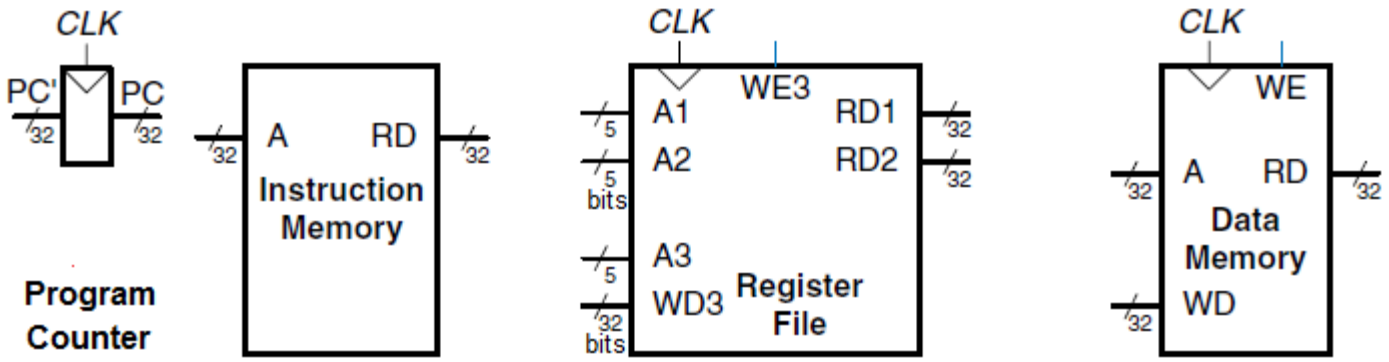
Register file:

- A1, A2: porturi de citire adrese pt. operanzi (5 biti -> 2^5=32 registri pt. operanzi)
- RD1, RD2: iesiri pt. citire date corespunzatoare la A1 respectiv A2
- WD3: port scriere pe 32 biti
- WE3: intrare pt. activare scriere
- Ceas

Daca WE3 are valoarea 1, *Register file* scrie datele in registrii specificati pe frontul crescator al ceasului sistem

Proiectare microarhitectura MIPS

Diagrama bloc: contor program + memorie instructiuni + registrii + memorie date



Data Memory:

- un singur port citire/scriere

Daca WE are valoarea 1, sunt scrise datele de la WD la adresa A pe frontul crescator al ceasului. Daca WE are valoarea 0, este citita adresa A in RD

Note finale:

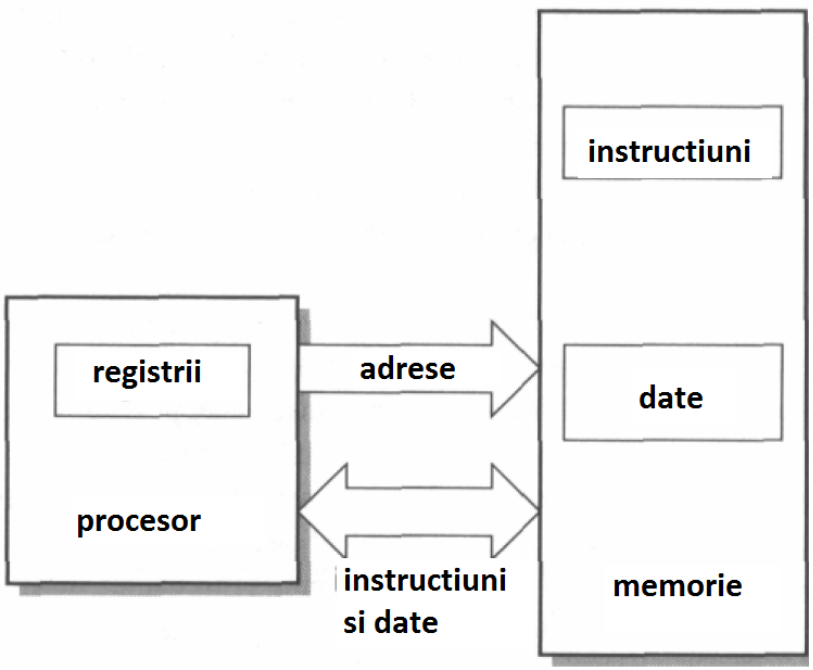
- Dupa fiecare tact de ceas, starea sistemului se schimba.
- Microprocesorul poate fi vazut ca un gigant FSM (finite state machine)

Sa recapitulam:

- La baza tuturor computerelor moderne sta principiul «stored-program» ? Cauze generatoare si cui apartine ideea ?
- Un program-stocat (stored-program) poate fi descris de catre un algoritm
- Arhitectura unui computer desemneaza modul in care utilizatorul vede un computer, adica:

setul de instructiuni, registrii, managementul memoriei, etc

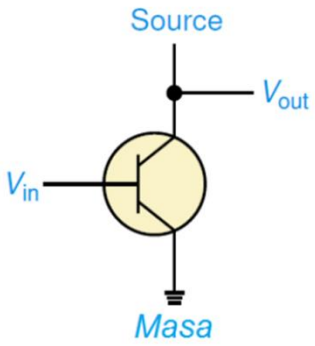
- Un procesor este un automat cu stari finite (FSA finite state automaton) care executa instructiunile continute in memorie.

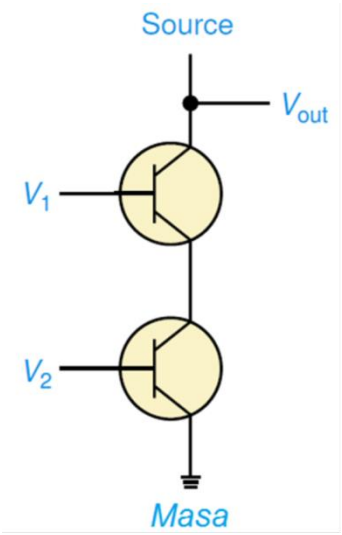


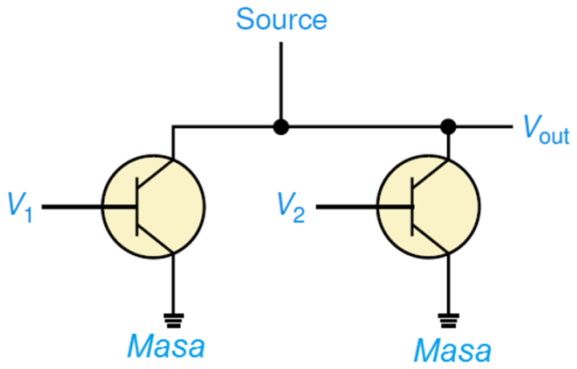
Sa recapitulam:

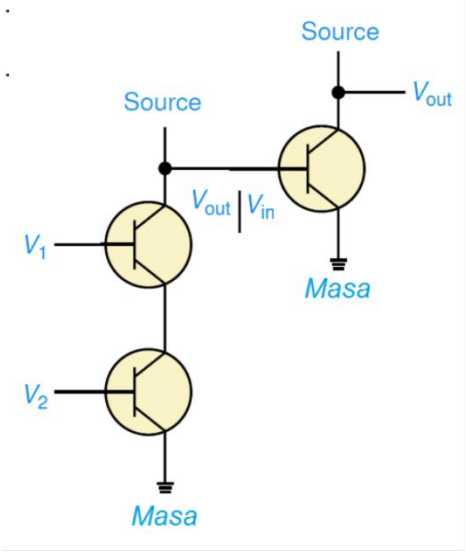
Procesele dintr-un calculator pot fi descrise in termeni hardware folosind *tranzistorii*.

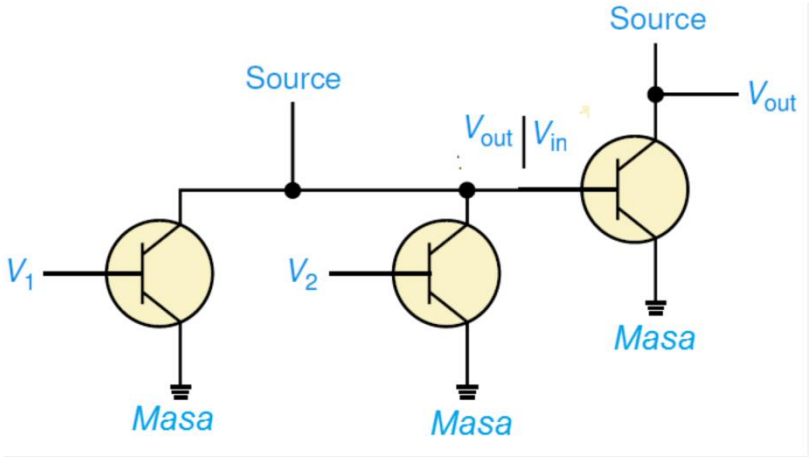
Folosind tranzistori, putem construi porti logice:











Din punct de vedere hardware avem urmatoarea ierarhie a nivelelor (de sus in jos):

- Tranzistori
- Porti logice, celule de memorie, alte circuite speciale
- Sumatoare, multiplexoare, decodoare
- Registrii, magistrale
- ALU, bancuri de registrii si bancuri de memorie
- Procesoare, memoria cache, managementul memoriei
- Circuite integrate
- Circuitele de pe placa (cablajul)
- Smartphones, PC-uri, microcontrolere

Sa analizam o alta arhitectura de procesor,

mult mai simpla decat MIPS,

elementara chiar,

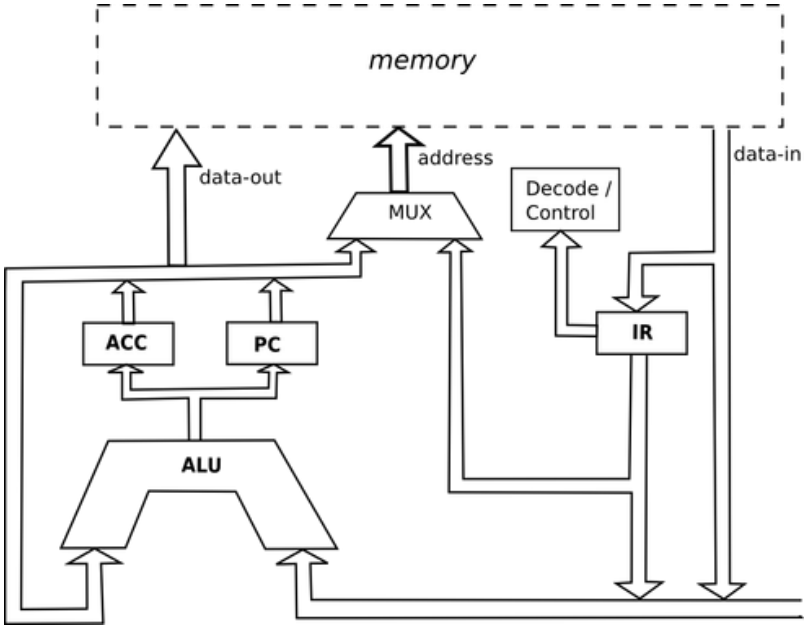
adica,

Procesorul simplu MU0

Procesorul simplu MU0

Un procesor simplu poate fi construit aband la baza:

- Un contor de program PC (program counter) care contine adresa instructiunii curente
- Un singur registru numit acumulator (ACC), care contine/mentine data in timpul prelucrarii acesteia
- O unitate ALU care.....? ..... efectueaza operatiile .....?.....care....?
- Un registru instructiune IR (instruction register) care contine instructiunea curenta
- Un decodor de instructiuni



Numarul mic de componente harware => un numar restrans de instructiuni

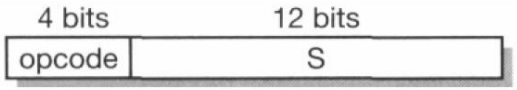
$\Rightarrow$ modelul este folosit doar in procesul de invatare

! Modelul nu poate fi folosit pentru compilatoare la limbaje de nivel inalt !

Procesorul simplu MU0

MU0 este o masina pe 16 biti ce poate adresa 8Kb de memorie

Instructiunile au 16 biti lungime, din care 4 pentru codul operatiei (opcode) iar 12 pentru campul adresa (S)



Cel mai simplu set de instructiuni foloseste doar 8 din cele 16 coduri de instructiuni disponibile:

Instructiune	Cod operatie	Efect
LDA S	0000	ACC := mem_{16}[S]
STO S	0001	mem_{16}[S] := ACC
ADD S	0010	ACC := ACC + mem_{16}[S]
SUB S	0011	ACC := ACC – mem_{16}[S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop

De exemplu, instructiunea **ACC:=ACC + mem_{16}[S]** inseamna: adauga continutul de la adresa de memorie S in acumulator.

Instructiunile sunt preluate/aduse de la adrese consecutive de memorie, incepand cu adresa zero, pana cand o instructiune de salt (jump) modifica adresa curenta de lucru (PC), aceasta devenind adresa specificata in instructiunea de salt.

Procesorul simplu MU0

Implementarea unei instructiuni pe microprocesor presupune impartirea arhitecturii in:

- **datapath**: componentele care transporta, stochaza sau prelucreaza biti + acumulator, contor program, ALU, registrii pentru instructiuni

O instructiune starteaza cand ajunge in registrul de instructiuni.

Executia unei instructiuni:

- Accesarea operandului/operanzilor din memorie si executarea operatiei;
- Aducerea (fetch) urmatoarei instructiuni de executat in PC (prin incrementare adresa curenta in ALU)

Initializarea procesorului: prin semnalul *reset* procesorul starteaza dintr-o stare/adresa cunoscuta: in cazul procesorului MU0 de la adresa 00016

- **control logic**: componentele care nu pot fi incluse in datapath si pot fi descrise folosind FSM

Rol principal: decodare instructiune curenta.

Procesorul simplu MU0

Procesorul MU0 poate fi transformat intr-un procesor «util» prin:

- Extinderea spatiului de adrese
- Adaugarea de moduri noi de adresare
- Permiterea ca contorul de program (PC) sa stocheze si adresa de revenire dintr-o subrutina
- Adaugarea de noi registrii
- Adaugarea modului de lucru cu intreruperi
- ...

Proiectarea setului de instructiuni – generalitati-

O instructiune este compusa din:

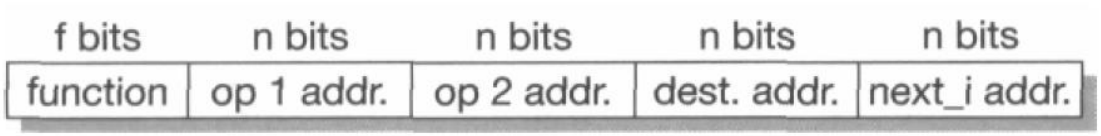
- Cativa biti necesari pentru a o diferentia fata de alte instructiuni
- Cativa biti pentru specificarea adresei operanzilor
- Cativa biti pentru a specifica locul (destinatia) unde sa fie depus rezultatul
- Cativa biti pentru a specifica adresa urmatoarei instructiuni de executat

Exemplu: adunarea a doua numere

In asamblare:

```
ADD     d,  s1,  s2,  next_i   ;  d  :=  s1  +  s2
```

In binar:



Instru

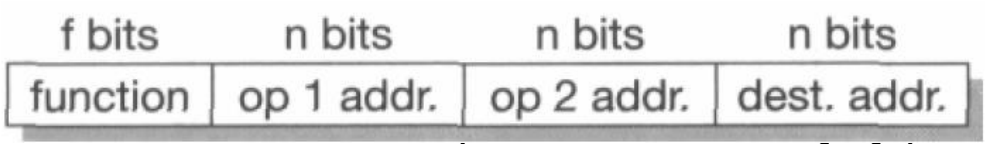
Formatul necesita 4n + f biti per instructiune, adica n biti per operand + f biti pentru numele instructiunii (ADD)

Proiectarea setului de instructiuni –generalitati-

OBIECTIV GENERAL: reducerea numarului de biti pt reprezentare instructiune

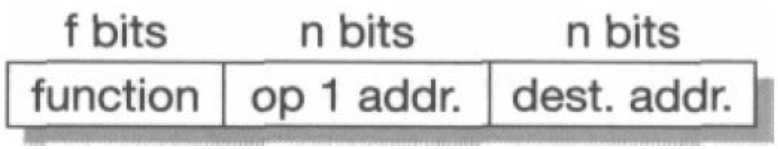
Pas 1.: renuntarea la adresa instructiunii urmatoare, aceasta putandu-se calcula prin adaugarea lungimii instructiunii curente la contorul de program (PC) (exceptand cazul cu ramificatii (branch)), obtinandu-se o instructiune in format cu 3 adrese

```
ADD     d,    s1,    s2      ;  d  :=  s1  +  s2
```



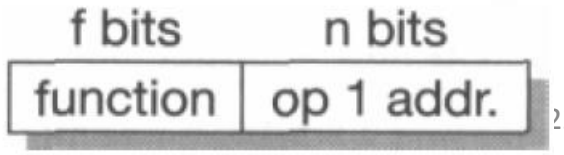
Pas 2.: ca si registru destinatie, poate fi folosit unul din registrii sursa, obtinandu-se o instructiune in format cu 2 adrese:

```
ADD     d,  s1   ;  d  :=  d  +  s1
```



Pas 3. : Daca folosim drept destinatie registrul acumulator (cazul procesorului MU0) o instructiune de adunare trebuie sa specifice doar un operand, obtinandu-se o instructiune in format cu o adresa:

```
ADD   s1     ; accumulator := accumulator + s1
```



Proiectarea setului de instructiuni - generalitati-

OBIECTIV GENERAL: reducerea numarului de biti pt reprezentare instructiune-continuare-

Pas 4.: daca se foloseste pentru adunare o stiva de evaluare se ajunge la o instructiune in format cu 0 adrese:

ADD ; top_of_stack := top_of_stack + next_on_stack

Pas 5. : Ar fi prea de tot, sa facem ceva fara sa specificam ceva anume !!!!!!!!!!!!!!!!!!!

NOTA: Arhitectura ARM standard foloseste formatul de instructiune cu 3 adrese

Tipuri de instructiuni:

- De calcul/procesare: adunare, scadere, inmultire
- De mutare date: intre registrii, intre registrii si memorie, etc
- De control al fluxului program/instructiuni
- Speciale, de control al procesorului

Moduri de adresare: atunci cand este accesat un operand pentru procesare sau mutare, exista cateva moduri/tehnici standard de a specifica locatia unde se afla acest operand:

- Adresare imediata/directa: printr-o valoare in binar continuta in instructiune
- Adresare indirecta: instructiunea contine valoarea binara a adresei de memorie care contine adresa in binar a valorii respective
- Adresare prin registru: valoara este continuta intr-un registru iar instructiunea contine «numele» registrului respectiv
- Adresare indirecta prin registru: instructiunea contine registrul care contine o adresa de memorie la care se afla valoarea
- Adresarea baza+offset: instructiunea specifica un registru (baza) si o valoare offset care se adauga la baza pentru a obtine adresa de memorie a operandului
- Adresarea baza+index: spre deosebire de precedenta, offsetul este continut intr-un alt registru

Instructiunile de control / ramificatii (branches)/(jumps)

Permit modificarea explicita a valorii din contorul de program (PC)

Au in general forma:

B

...

...

LABEL

si forteaza PC (Program Counter) sa indice/sara la instructiunea aflata la LABEL

Ponderea operatiilor efectuate de un procesor ARM: (valori statistice)

Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

Limbajul de asamblare pentru procesoarele ARM

Procesoare ARM

(Acorn RISC Machine)original

(Advanced RISC Machine)

(RISC⇔Reduced Instruction Set Computing)

De ce ARM?

- Setul redus de instructiuni face mai usoara initierea in limbajul de asamblare

- Procesoarele RISC sunt folosite in:
  - Dispozitive mobile (tablete, telefoane mobile)
  - Sisteme incapsulate
  - Supercomputere (K, Sequoia)

  Deoarece necesita mai putini tranzistori in comparatie cu arhitecturile CISC

  ⇔

  costuri mai mici, grad mai mare de integrare, mai putina caldura degajata si consum mai mic de energie raportat la alte arhitecturi.

Ce se afla la baza arhitecturii ARM?

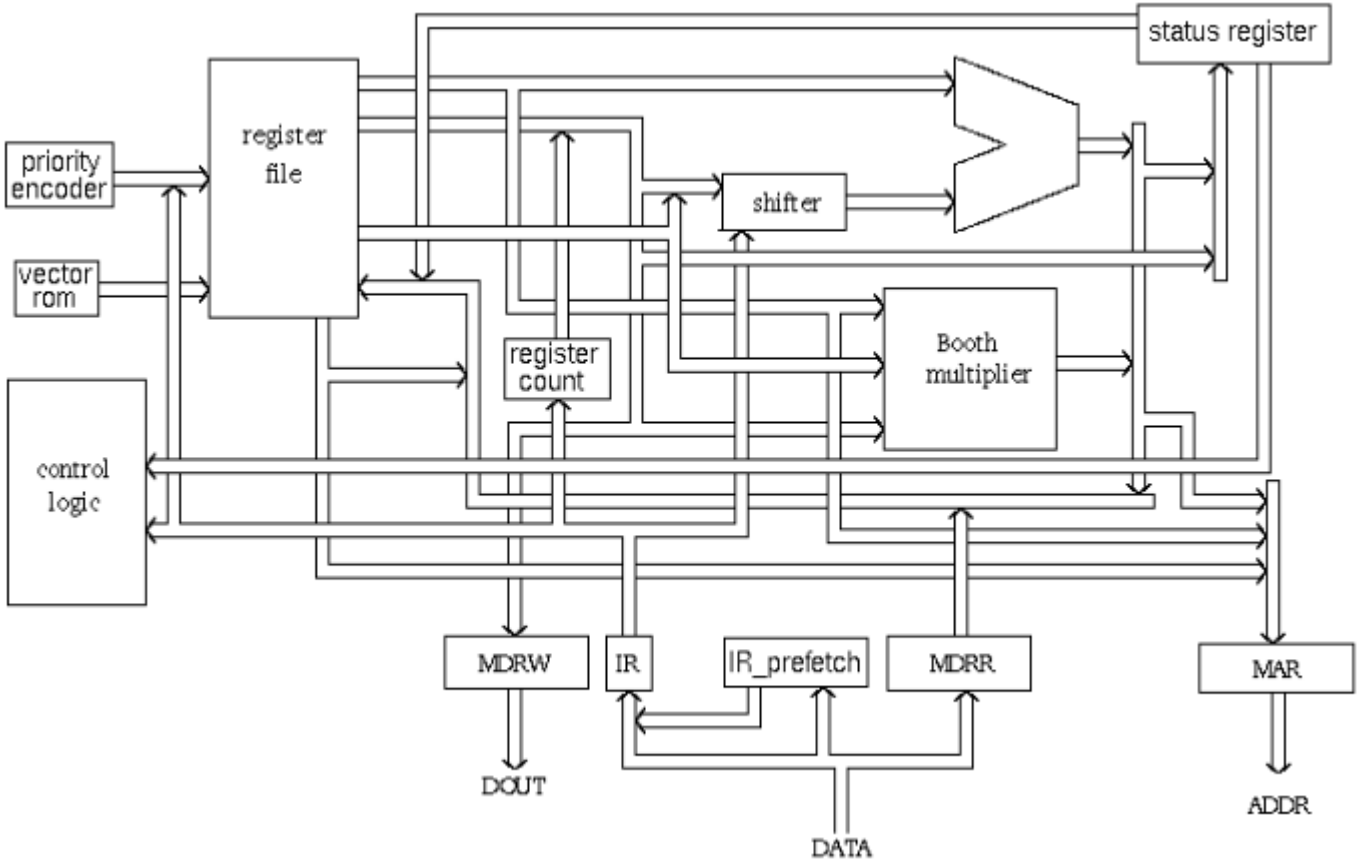
Princeton + Harvard => a arhitectura hibrida

adica simplificat:

1 memorie si 2 magistrale (date si instructiuni)

Un exemplu diagrama bloc a unui procesor ARM:

1. Register file: 37 registrii: 31 registrii generali pe 32 biti, 6 registrii de stare
2. Booth Multiplier: inmultire 2 nr. binare in complement fata de 2 cf. algor. A.D. Booth (1950)
3. Barrel shifter : operatii de shiftare si rotatie pe vectori de biti
4. Arithmetic Logic Unit (ALU)
5. Control Unit.



Ce este limbajul de asamblare?

Limbajul de asamblare este un limbaj de programare a calculatoarelor aflat imediat ca si nivel dupa limbajul/codul mașină (binar).

Acesta folosește o desemnare simbolică a elementelor de program, numite uneori mnemonici.

Programarea in limbaj de asamblare presupune o bună cunoaștere a structurii procesorului

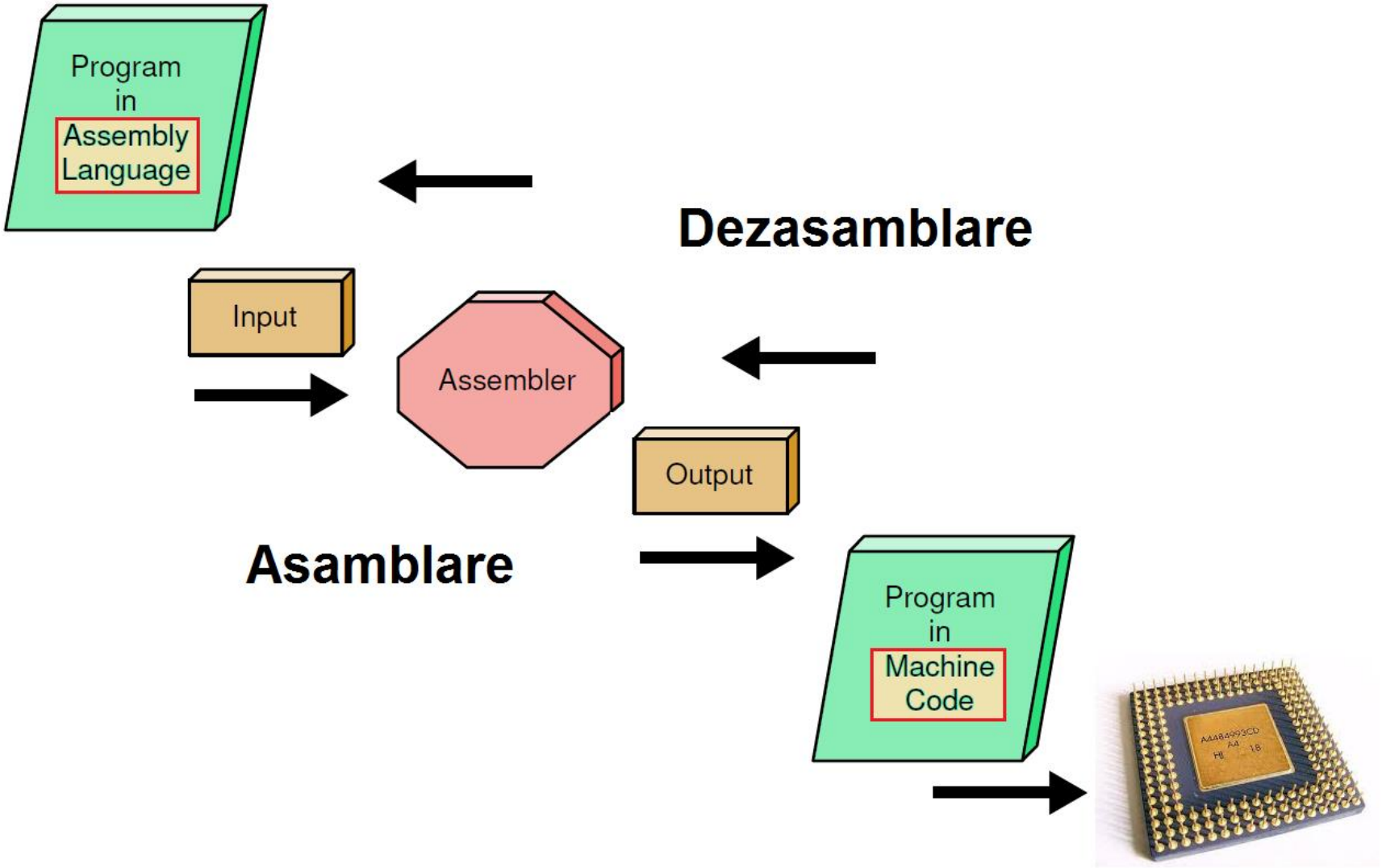
Programul rezultat va putea funcționa numai pe un anumit tip de calculator, portabilitatea putand fi asigurata (nu intotdeauna si in orice conditii) doar de catre limbajele de programare de nivel inalt.

Limbajul de asamblare este un limbaj de nivel scazut (low level), el fiind foarte apropiat de codul masina.

Nota:

Creste nivel limbaj => creste nivel intelegere utilizator

Scade nivel limbaj => creste nivel intelegere de catre nicroprocesor


Program in Assembly Language
Dezasamblare
Input
Assembler
Output
Asamblare
Program in Machine Code

Procesul de asamblare

- generare tabelă de simboluri, ce conține toate numele simbolice din programul sursă
- sunt contorizate instrucțiunile și datele, asociind numelor simbolice un "deplasament" față de începutul programului. Adresa de inceput este data de catre sistemul de operare si difera de valoarea 0.
- se genereaza programul obiect, prin traducerea instrucțiunilor si inlocuirea numelor simbolice cu adresele din tabela de simboluri
- se genereaza programul executabil cu ajutorul unui *linkeditor*, care in principal editeaza legaturile intre module (linkage edit).

Registrii ARM: 37 de registrii:

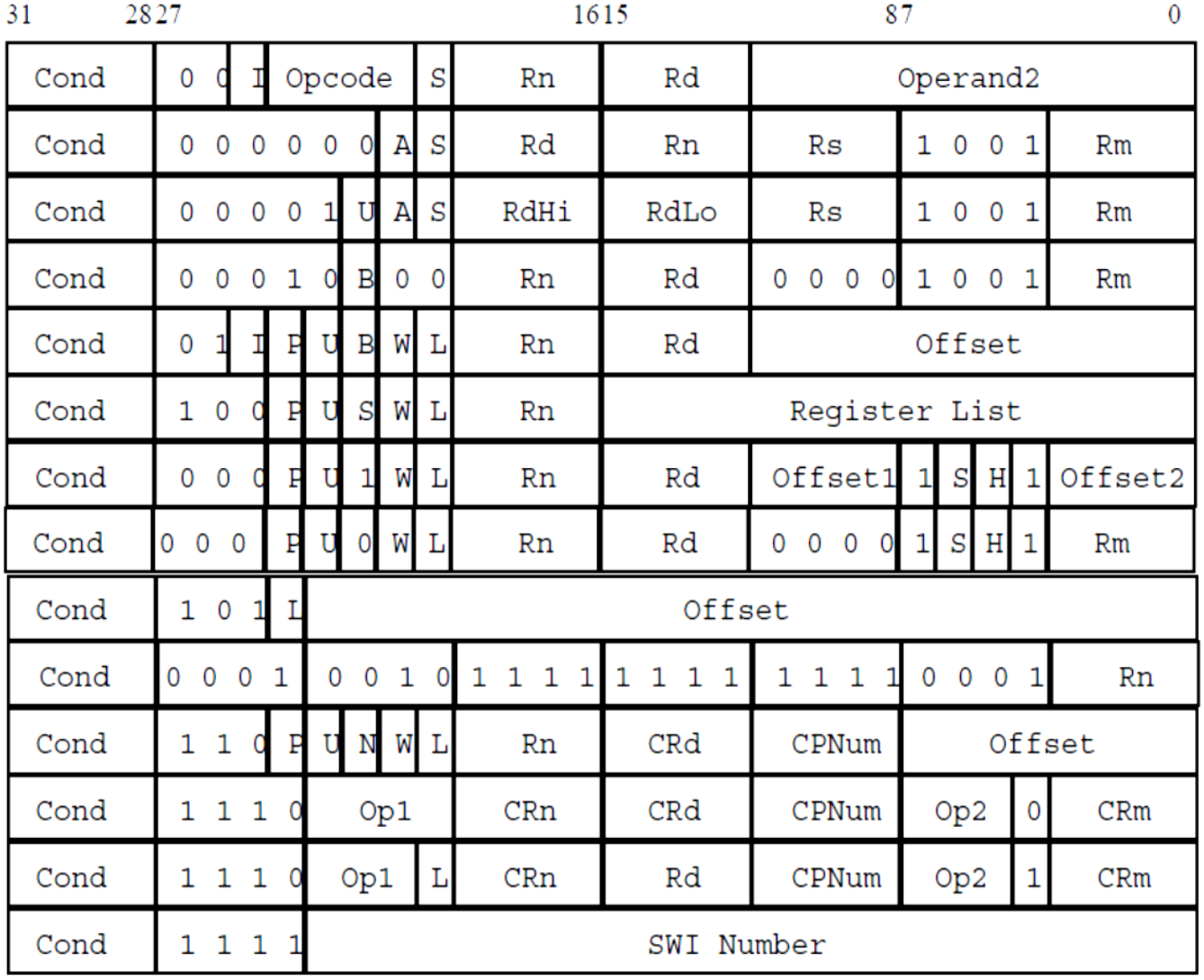
- 30 de registrii de uz general, pe 32 de biti
  - primii 15 r0...r14 sunt vizibili in functie de modul curent al procesorului
  - r13 et utilizat ca si stack pointer (sp) in limbajul de asamblare (compilatoarele C/C++ folosesc acelasi registru)
  - r14 este utilizat in modul User ca si *link register* (lr) pt. a stoca adresa de revenire dintr-o subrutina
- Un registru pentru contorul de program: r15 (PC). Acesta este incrementat cu un *word* (4 bytes) la fiecare instructiune
- Registru pt. stocarea starii curente program (CPSR-current program status register)
- 5 registrii SPSR (saved program status register) pt. stocarea CPSR in cazul unei intreruperi

Nota:

- Toti registrii r0...r14 pot fi accesati direct de catre toate instructiunile

Formatul instructiunilor ARM

Continutul celor 32 de biti:



Instruction type

- Data processing / PSR Transfer
- Multiply
- Long Multiply
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset
- Halfword transfer: Register offset
- Branch
- Branch Exchange
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

Executia conditionala a instructiunilor ARM

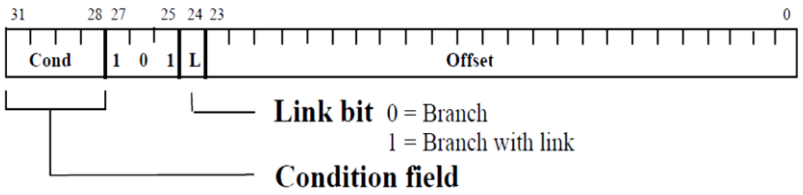
Toate instructiunile ARM au un camp (numit {cond}) care contine o conditie care determina/decide daca instructiunea va fi executata de catre CPU.

Codurile sunt:

EQ	Equal
NE	Not equal
CS/HS	Higher or same (unsigned >=)
CC/LO	Lower (unsigned <)
MI	Negative
PL	Positive or zero
VS	Overflow
VC	No overflow
HI	Higher (unsigned <=)
LS	Lower or same (unsigned <=)
GE	Signed >=
LT	Signed <
GT	Signed >
LE	Signed <=
AL	Always (usually omitted)

Instructiuni de ramificatie (branch instructions)

B si **BL** (Branch si Branch with link)



Sintaxa: B{cond} eticheta

BL{cond} subrutina_eticheta

- BL, in functie de o conditie (optionala) executa un salt la o eticheta. Nu afecteaza LR (link register)
- In plus, BL copiaza in r14 adresa instructiunii urmatoare. La reintoarcerea din subrutina, trebuie restaurat PC (program counter) conform continutului LR (link register) folosind instructiunea MOV pc, lr
- Instructiunile pot accesa un domeniu de ±32Mb de la adresa instructiunii curente. Totusi, aceste instructiuni pot fi folosite chiar daca se iese din domeniu.

Exemple: B eticheta1 BL eticheta2

BX {cond} Rm

In functie de indeplinirea conditiei este executat salt la instructiunea cu adresa Rm.

Exemplu: BX r7

Instructiuni de procesare a datelor

Prelucrari de date in ARM:

- Operatii aritmetice
- Operatii logice
- Comparatii
- Mutari de date intre registrii

Note:

- Prelucrarile anterioare lucreaza doar in registrii, NU in memorie
- Efectueaza operatii asupra unuia sau doi operanzi
- Intotdeauna, primul operand al acestor instructiuni este un registru

Operatii arimetice

Operatii

- ADD operand1 + operand2
- ADC operand1 + operand2 + carry
- SUB operand1 - operand2
- SBC operand1 - operand2 + carry -1
- RSB operand2 - operand1
- RSC operand2 - operand1 + carry - 1

Sintaxa

<Operation>{<cond>} Rd, Rn, Operand2

Comparatii

Operatii

- CMP operand1 - operand2
- CMN operand1 + operand2
- TST operand1 AND operand2
- TEQ operand1 XOR operand2

Sintaxa

<Operation>{<cond>} Rn, Operand2

Operatii logice

Operatii

- AND operand1 AND operand2
- XOR operand1 XOR operand2
- ORR operand1 OR operand2
- BIC operand1 AND NOT operand2

Sintaxa

<Operation>{<cond>} Rd, Rn, Operand2

Deplasari de date

Operatie

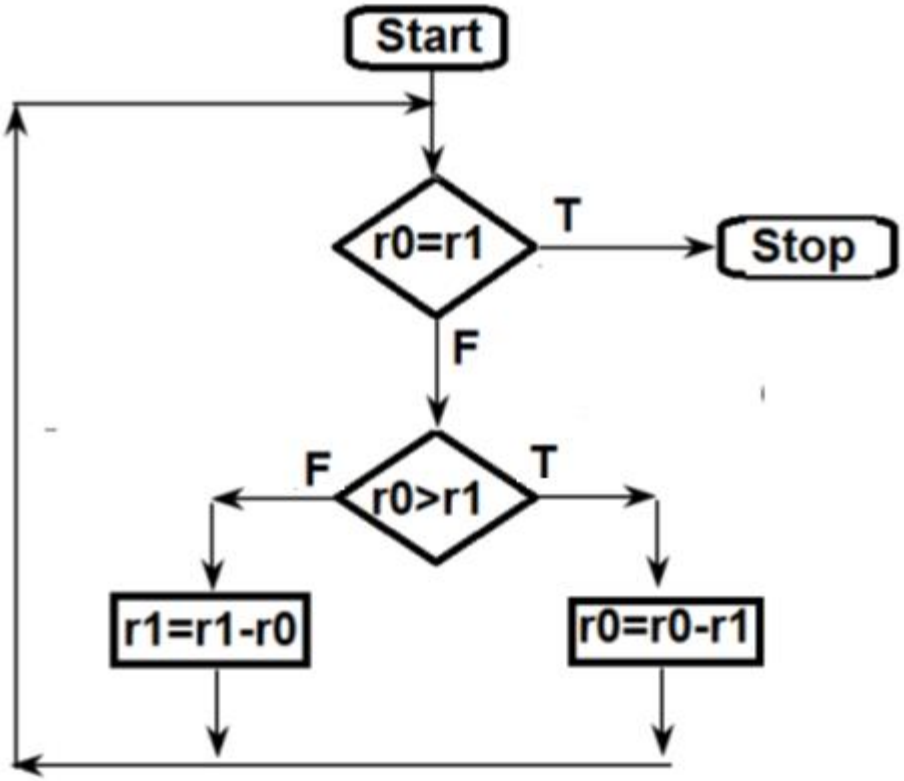
- MOV operand2

Sintaxa

<Operation>{<cond>} Rd, Operand2

Exemplu: MOV r0, r1

Implementare algoritm Euclid pt. *GCD* a 2 nr. naturale prin scaderi repetate




```
gcd     cmp   r0, r1       ;if r0 > r1
        subgt r0, r0, r1   ;subtract r1 from r0
        sublt r1, r1, r0   ;else subtract r0 from r1
        bne   gcd          ;reached the end?
```

VA URMA