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

CURS 11

Limbajul de asamblare pentru procesoare ARM

Exemple

Dr. Mafteiu-Scai Liviu Octavian

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

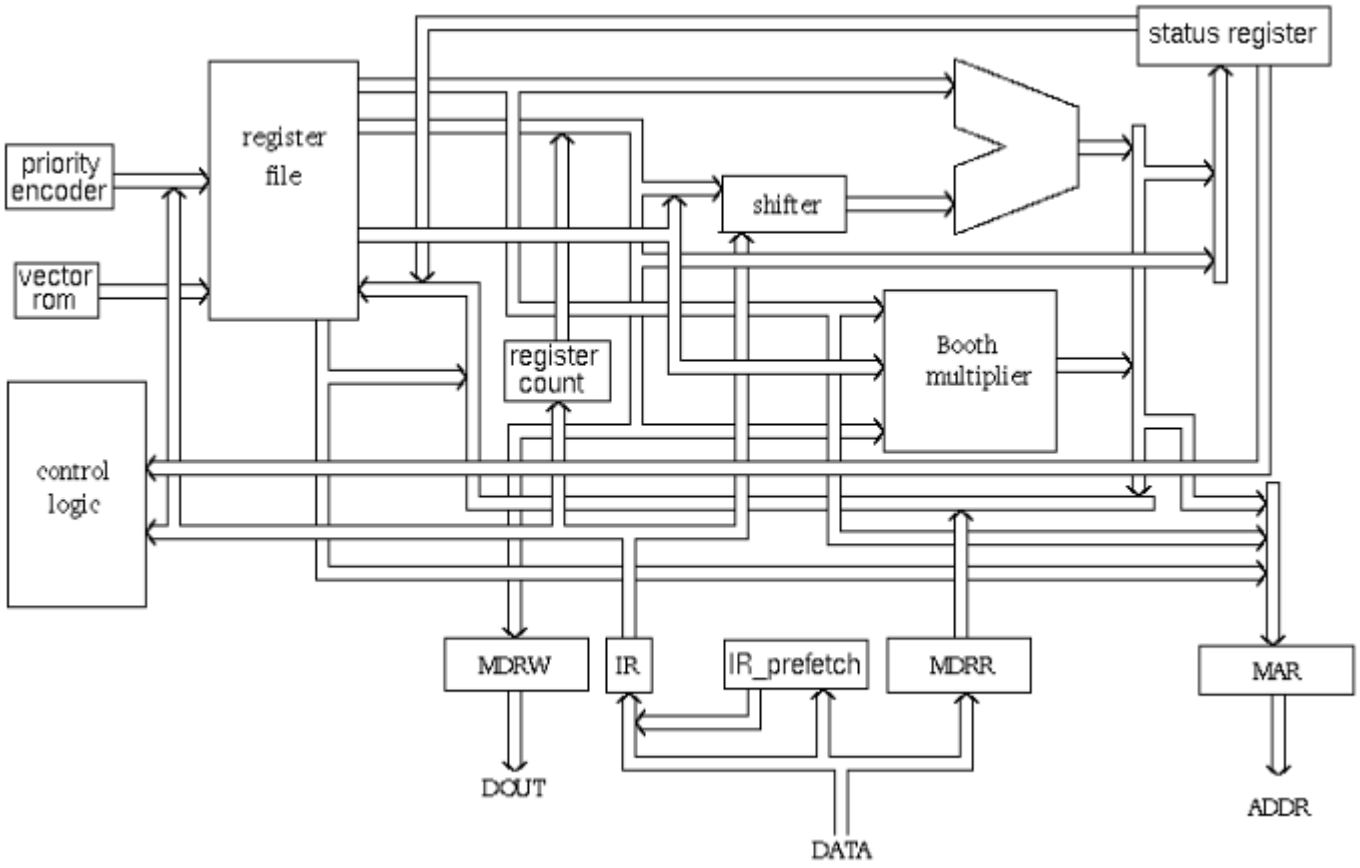
- Procesoarele RISC sunt folosite in:
  - Dispozitive mobile (tablete, telefoane mobile)
  - Sisteme incapsulate
  - Supercomputere (K, Sequoia)

  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.

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

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.

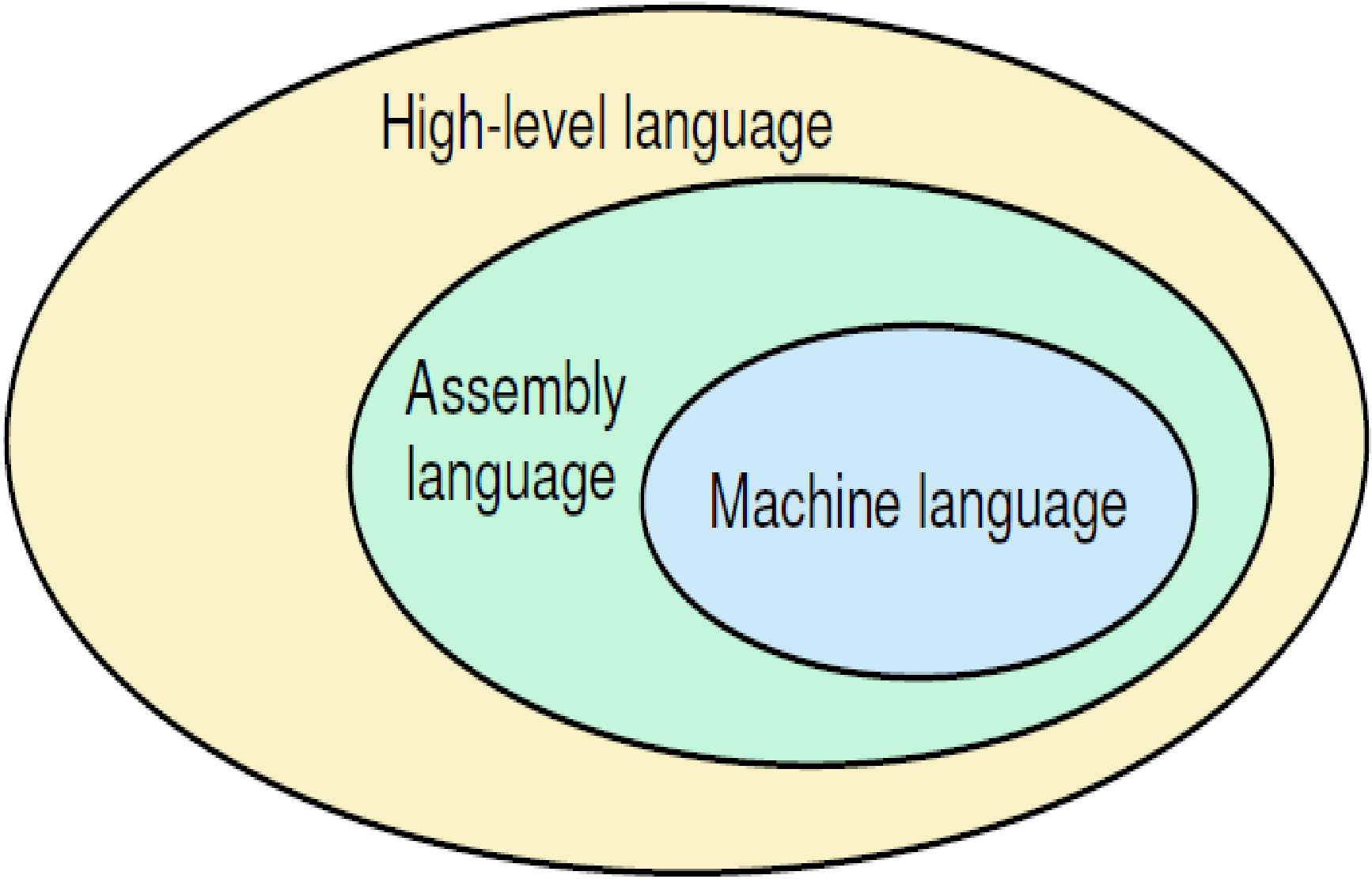
Limbajul de asamblare este un limbaj de nivel scazut (low level), el fiind foarte apropiat de codul masina.

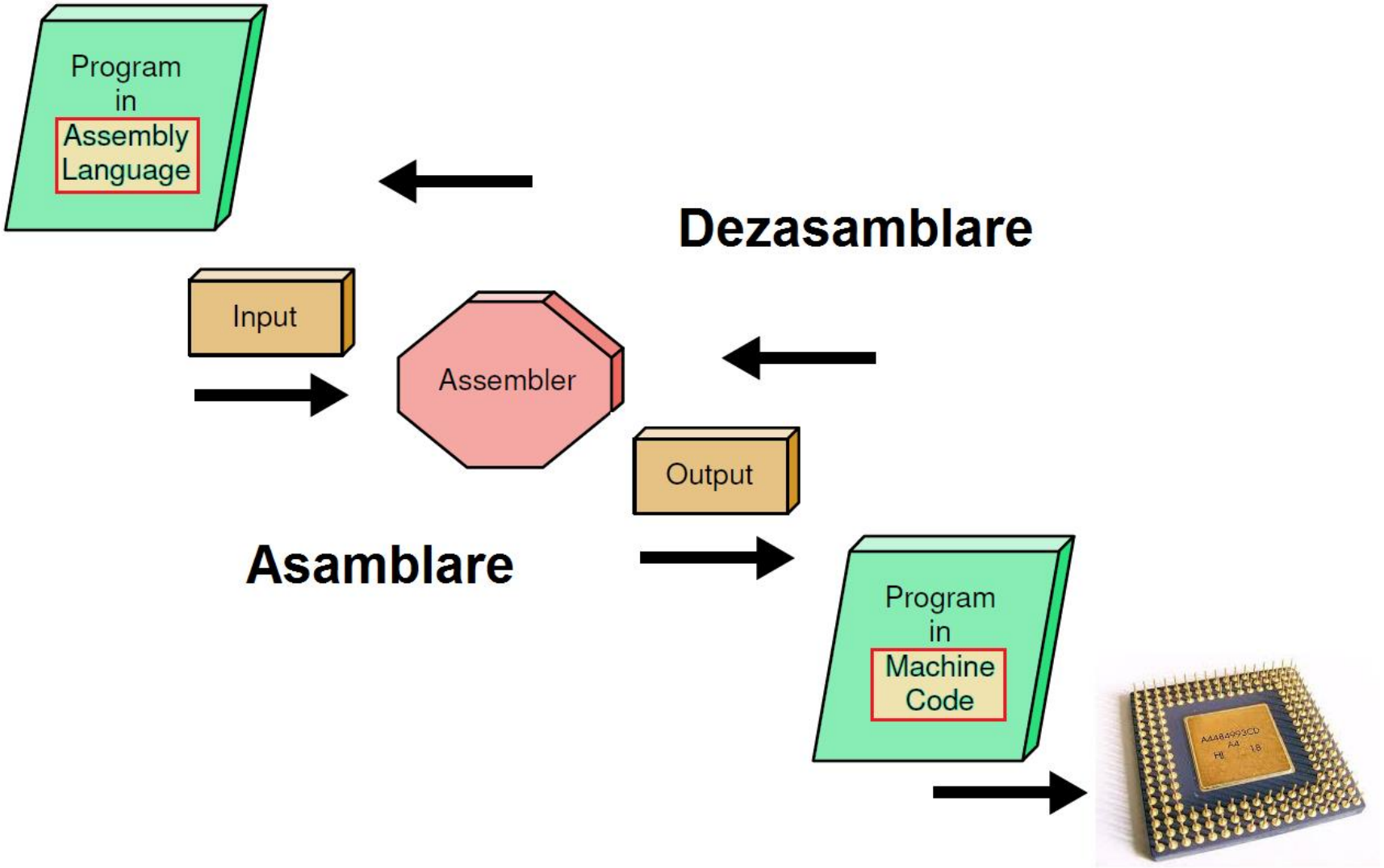
Nota:

Creste nivel limbaj => creste nivel intelegere utilizator

Scade nivel limbaj => creste nivel intelegere de catre nicroprocesor

Ce este limbajul de asamblare?




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:

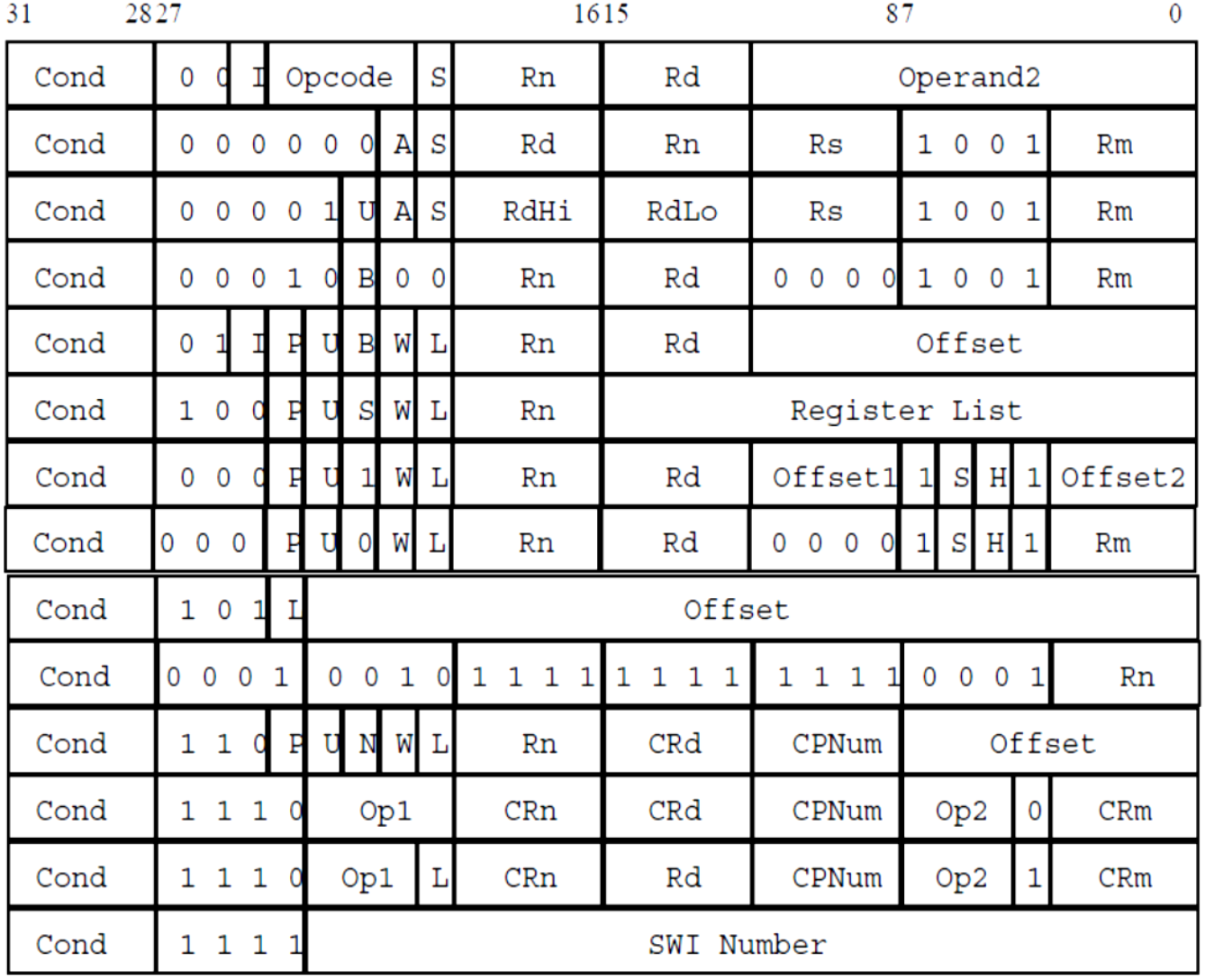
- 30 de registrii de uz general, pe 32 de biti
  - primii 15 r0...r14 sunt vizibili in functie de modul curent al procesorului
  - r13 et utilizat ca si stack pointer (sp) in limbajul de asamblare (compilatoarele C/C++ folosesc acelasi registru)
  - r14 este utilizat in modul User ca si *link register* (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 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

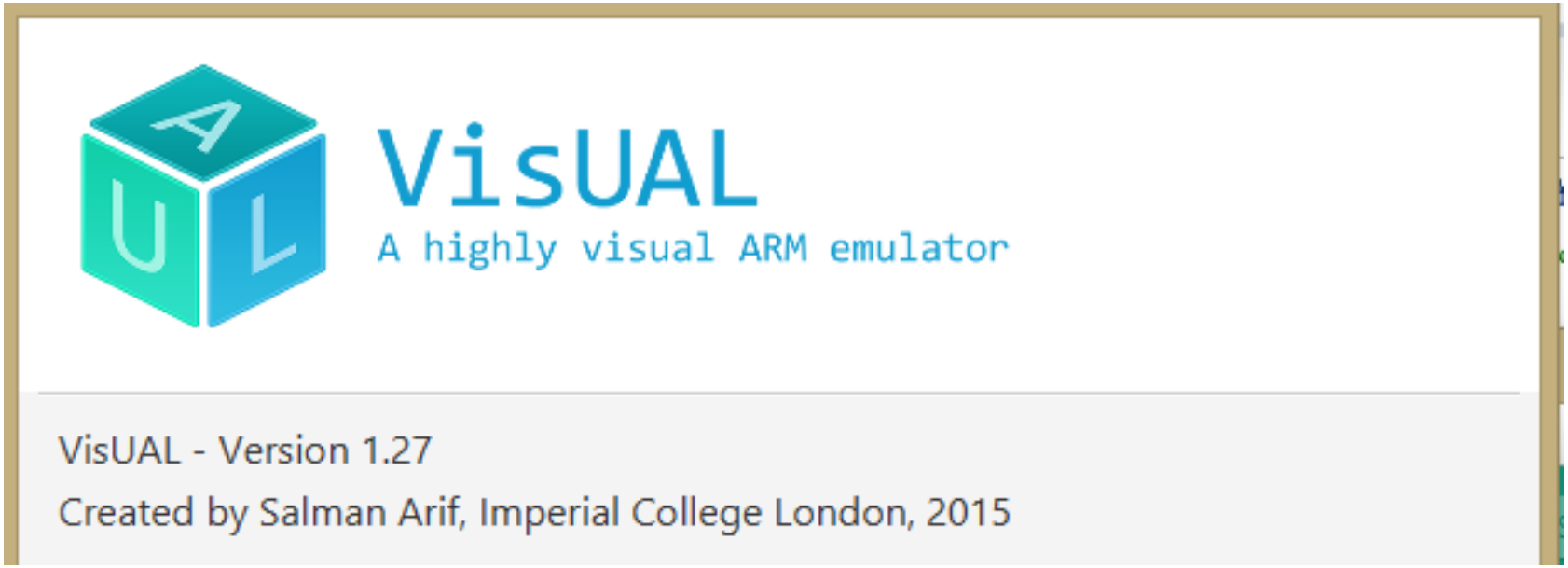
« *Learning step by step by examples*»

s-a dovedit in timp una din cele mai eficiente si rapide metode de invatare

asa ca...

Algoritmul va fi reprezentat sub forma de schema logica (organigrama), semnificatia simbolurilor grafice fiind cea cunoscuta/standard.

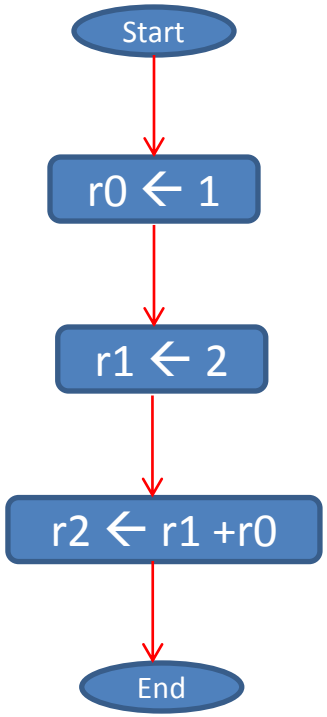
Pentru invatare limbaj asamblare vom folosi un emulator simplu ca utilizare si usor de instalat) adica doar se dezarhiveaza fisierul descarcat de la adresa **http://salmanarif.bitbucket.org/visual/**



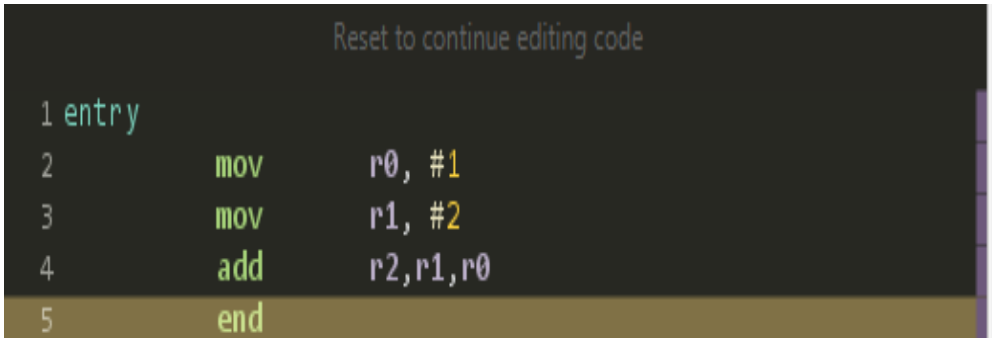
Move	**MOV**	MOV{S}{cond} dest, op1 {, SHIFT_op #expression}
Move Negated	**MVN**	MVN{S}{cond} dest, op1 {, SHIFT_op #expression}
Address Load	**ADR**	ADR{S}{cond} dest, expression
LDR Psuedo-Instruction	**LDR**	LDR{S}{cond} dest, =expression
Add	**ADD**	ADD{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Add with Carry	**ADC**	ADC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Subtract	**SUB**	SUB{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Subtract with Carry	**SBC**	SBC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Reverse Subtract	**RSB**	RSB{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Reverse Subtract with Carry	**RSC**	RSC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise And	**AND**	AND{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise Exclusive Or	**EOR**	EOR{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise Clear	**BIC**	BIC{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Bitwise Or	**ORR**	ORR{S}{cond} dest, op1, op2 {, SHIFT_op #expression}
Logical Shift Left	**LSL**	LSL{S}{cond} dest, op1, op2
Logical Shift Right	**LSR**	LSR{S}{cond} dest, op1, op2
Arithmetic Shift Right	**ASR**	ASR{S}{cond} dest, op1, op2
Rotate Right	**ROR**	ROR{S}{cond} dest, op1, op2
Rotate Right and Extend	**RRX**	RRX{S}{cond} op1, op2
Compare	**CMP**	CMP{cond} op1, op2 {, SHIFT_op #expression}
Compare Negated	**CMN**	CMN{cond} op1, op2 {, SHIFT_op #expression}
Test Bit(s) Set	**TST**	TST{cond} op1, op2 {, SHIFT_op #expression}

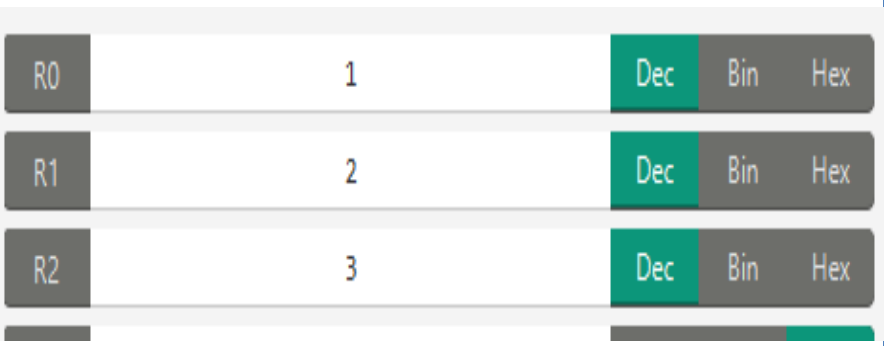
Test Equals	**TEQ**	TEQ{cond} op1, op2 {, SHIFT_op #expression}
Load Register	**LDR**	LDR{B}{cond} dest, [source {, OFFSET}] Offset addressing LDR{B}{cond} dest, [source, OFFSET]! Pre-indexed addressing LDR{B}{cond} dest, [source], OFFSET Post-indexed addressing
Store Register	**STR**	STR{B}{cond} source, [dest {, OFFSET}] Offset addressing STR{B}{cond} source, [dest, OFFSET]! Pre-indexed addressing STR{B}{cond} source, [dest], OFFSET Post-indexed addressing
Load Multiple Registers	**LDM[dir]**	LDM[dir]{cond} source, {list of registers}
Store Multiple Registers	**STM[dir]**	STM[dir]{cond} dest, {list of registers}
Branch	**B**	B{cond} target
Branch with Link	**BL**	BL{cond} target
Declare Word(s) in Memory	**DCD**	name DCD value_1, value_2, ... value_N
Declare Constant	**EQU**	name equ expression
Declare Empty Word(s) in Memory	**FILL**	{name} FILL N N must be a multiple of 4
Stop Emulation	**END**	END{cond}

Exemplul 1: X = a+b, adunarea a doua valori din 2 registrii in al 3-lea

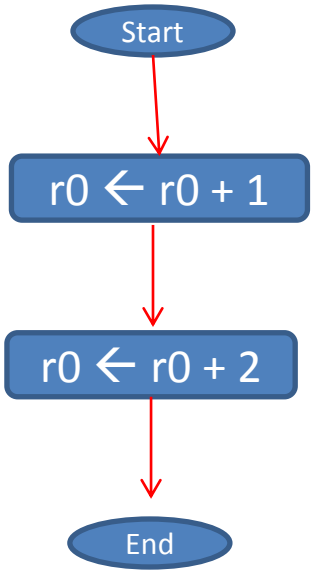


mov: copiaza o valoare intr-un registru
add: aduna continutul a doi registrii in alt registru





Exemplul 2: X = a+b, adunarea a doua valori folosind doar 1 registru

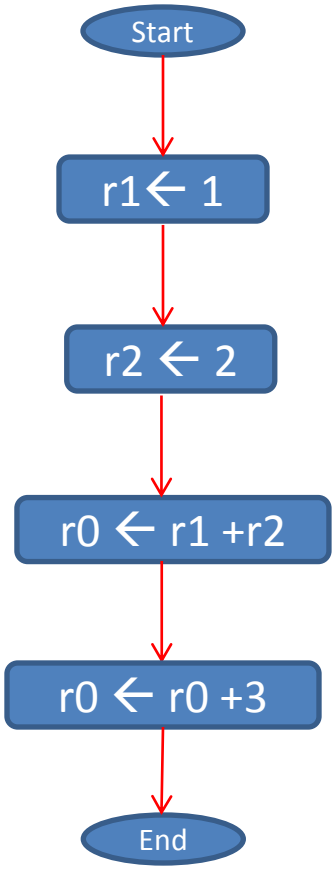


Reset to continue editing code

```
1 entry
2           add         r0, r0,#1
3           add         r0,r0,#2
4           end
```

R0	3
R1	0
R2	0

Exemplul 3: X = a+b+c, adunarea a trei valori folosind 3 registrii



Reset to continue editing code

```
1 entry
2           mov         r1, #1
3           mov         r2, #2
4           add         r0,r1,r2
5           add         r0,r0,#3
6           end
```

R0	6	Dec
R1	1	Dec
R2	2	Dec
R3	0x0	Dec

Exemplul 4: Se declara si initilizeaza doua constante a1 si a2, se muta continutul lor in 2 registrii si se aduna in al treilea registru r2

```
1 entry
2
3 a1        EQU     1
4 a2        equ     2
5           mov     r0, #a1
6           mov     r1, #a2
7           add     r2, r0,r1
```

R0	0x1
R1	0x2
R2	0x3
R3	0x0
R4	0x0

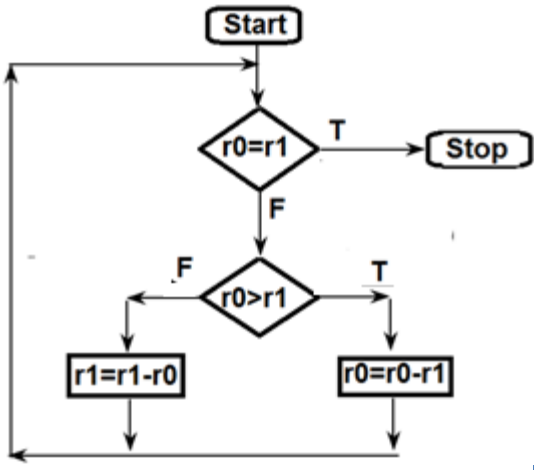
Intrebare: Este corect? Argumentati

```
1  entry
2
3  a1       EQU     1
4  a2       equ     2
5           mov     r0, #a1
6           mov     r1, #a2
7           add     r2, r0,r1
8  a1       equ     20
9           mov     r0,#a1
10          add     r3,r0,r1
11          end
```

R0	1
R1	2
R2	3
R3	3
R4	0x0
R5	0x0
R6	0x0

Exemplul 5. Algoritmul lui Euclid pt. CMMDC prin scaderi repetate

?




```
Reset to continue editing code
1 entry
2           mov     r0, #15
3           mov     r1, #50
4 cmmdc     cmp     r0,r1
5           subgt   r0,r0,r1
6           sublt   r1,r1,r0
7           b       cmmdc
8           end
9
```

Branch

Warning

Possible infinite loop detected

The instruction at line 4 has been branched to 1000 times. This may indicate unwanted behaviour, such as an infinite loop.

Branch destination: Line 4: cmmdc cmp r0,r1 — Show in Editor

Branch source: Line 7: b cmmdc — Show in Editor

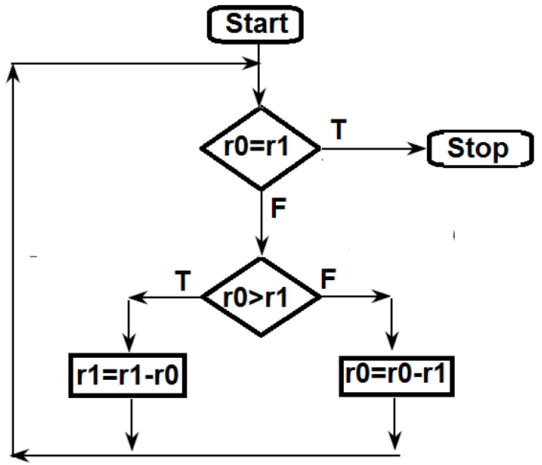
☐ Do not warn me again for this loop until I press "Reset"

Ignore and Continue | Change Threshold | Abort Execution

R0	5
R1	5
R2	0x0
R3	0x0
R4	0x0
R5	0x0
R6	0x0
R7	0x0
R8	0x0
R9	0x0
R10	0x0
R11	0x0
R12	0x0
R13	0xFF000000
LR	0x0

Exemplul 5. Algoritmul lui Euclid pt. CMMDC prin scaderi repetate

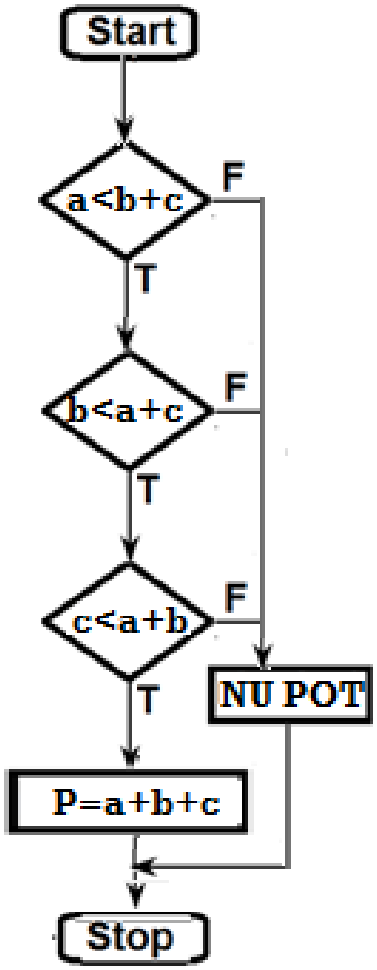
corect




```
Reset to continue editing code
1 entry
2           mov         r0, #15
3           mov         r1, #50
4 cmmdc     cmp         r0,r1
5           subgt       r0,r0,r1
6           sublt       r1,r1,r0
7           bne         cmmdc
8           end
```

R0	5
R1	5
R2	0
R3	0x0
R4	0x0

Exemplul 6 Calcul perimetru triunghi daca triunghiul exista

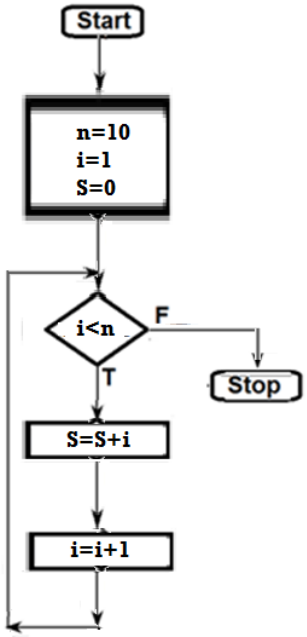


Reset to continue editing code

```
1  entry
2          mov     r0, #3
3          mov     r1, #4
4          mov     r2, #5
5          add     r3, r0,r1
6          add     r4, r0,r2
7          add     r5, r1,r2
8          cmp     r2,r3
9          addlt   r6,r2,r3
10         cmp     r0,r5
11         addlt   r7,r0,r5
12         cmp     r1, r4
13         addlt   r8,r1,r4
14         end
15
```

R0	3
R1	4
R2	5
R3	7
R4	8
R5	9
R6	12
R7	12
R8	12

Exemplul 7 Suma primelor n numere naturale – iterativ –



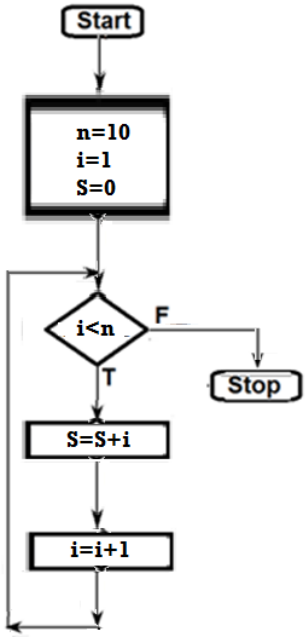
Reset to continue editing code

```
1 entry
2 enu       equ     10
3           mov     r0, #1
4           mov     r1, #0
5 for       add     r1, r1, r0
6           add     r0, r0, #1
7           cmp     r0, #enu
8           ble     for
9           end
10
```

R0	11
R1	55
R2	0x0
R3	0x0
R4	0x0
R5	0x0

? De ce avem valoarea 11 in registrul *r0* ?

Exemplul 8 Inmultirea a doua numere naturale – iterativ –




```
1  entry
2  aul      equ     10
3  beul     equ     5
4           mov     r0, #1
5           mov     r1, #0
6  for      add     r1, r1, #aul
7           add     r0, r0, #1
8           cmp     r0, #beul
9           ble     for
10          end
```

R0	6
R1	50
R2	0x0
R3	0x0
R4	0x0
R5	0x0

? De ce avem valoarea 6 in registrul *r0* ?

Teme de casa

1. Suma primelor n numere pare
2. Suma primelor numere impare
3. Al n-lea termen din sirul Fibonacci
4. Ridicarea la putere
5. Produsul primelor n numere naturale