Corso di Architettura dei Sistemi a Microprocessore

Esercitazione con simulatori Motorola 68K e ARM

Luigi Coppolino

Contact info

Prof. Luigi Coppolino
luigi.coppolino@uniparthenope.it

Università degli Studi di Napoli "Parthenope"
Dipartimento di Ingegneria

Centro Direzionale di Napoli, Isola C4
V Piano lato SUD - Stanza n. 512

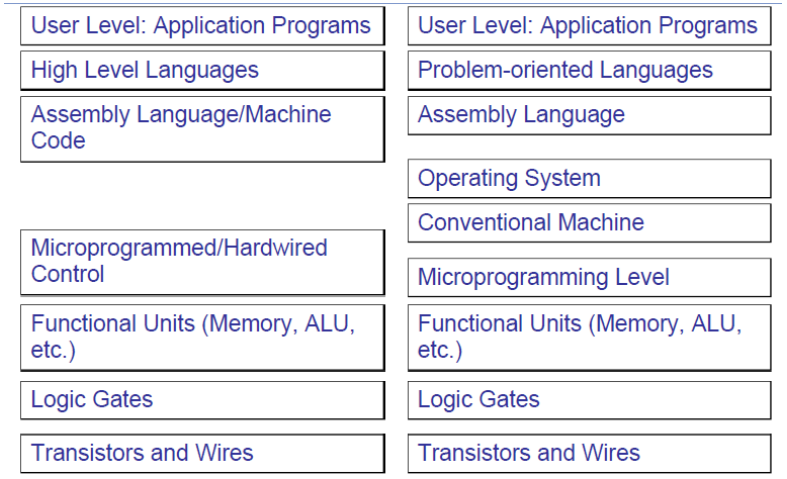
Tel: +39-081-5476702
Fax: +39-081-5476777

References

- Textbook (chapter 2)
- Manuale Freescale (http://www.freescale.com/files/archives/doc/ref_manual/M68000PRM.pdf)(http://www.freescale.com/files/dsp/doc/ref_manual/CFPRM.pdf)
- Manuale ARM (http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/DUI0204J_rvct_assembler_guide.pdf)
- Quick Guides:
  - ARM: http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf
  - Coldfire (m68000): http://home.anadolu.edu.tr/~sgorgulu/micro2/2008/68KISx1.pdf

Linguaggio Assembly

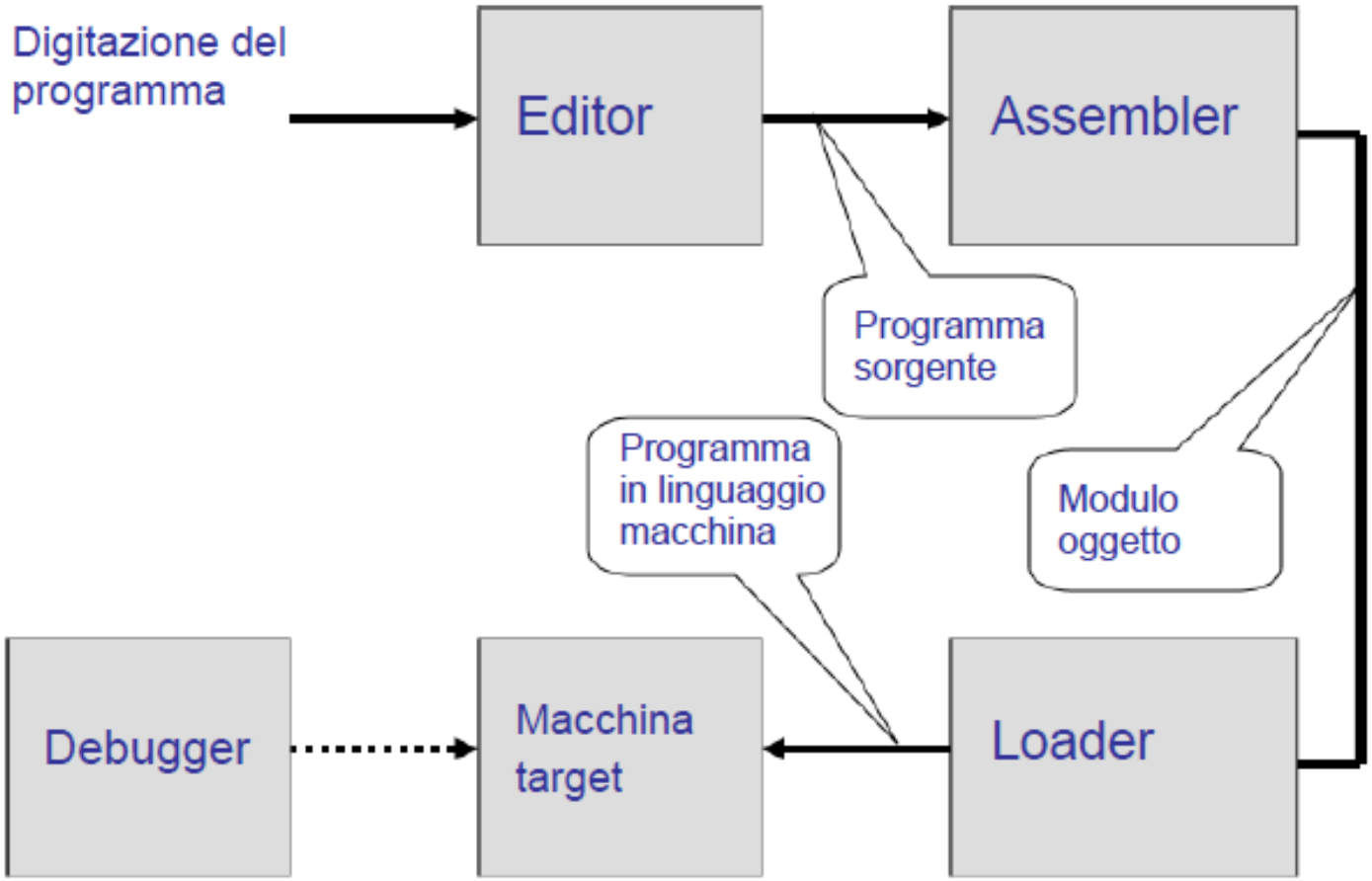
- È funzionalmente equivalente al linguaggio macchina, ma usa “nomi” più intuitivi (mnemonics)
- Definisce l’Instruction Set Architecture (ISA) della macchina
- Un compilatore traduce un linguaggio di alto livello, che è indipendente dall’architettura, in linguaggio assembly, che è dipendente dall’architettura
- Un assemblatore traduce programmi in linguaggio assembly in codice binario eseguibile
- Nel caso di linguaggi compilati (es. C) il codice binario viene eseguito direttamente dalla macchina target
- Nel caso di linguaggi interpretati (es. Java) il bytecode viene interpretato dalla Java Virtual Machine, che è al livello Assembly language



Il linguaggio Assembly

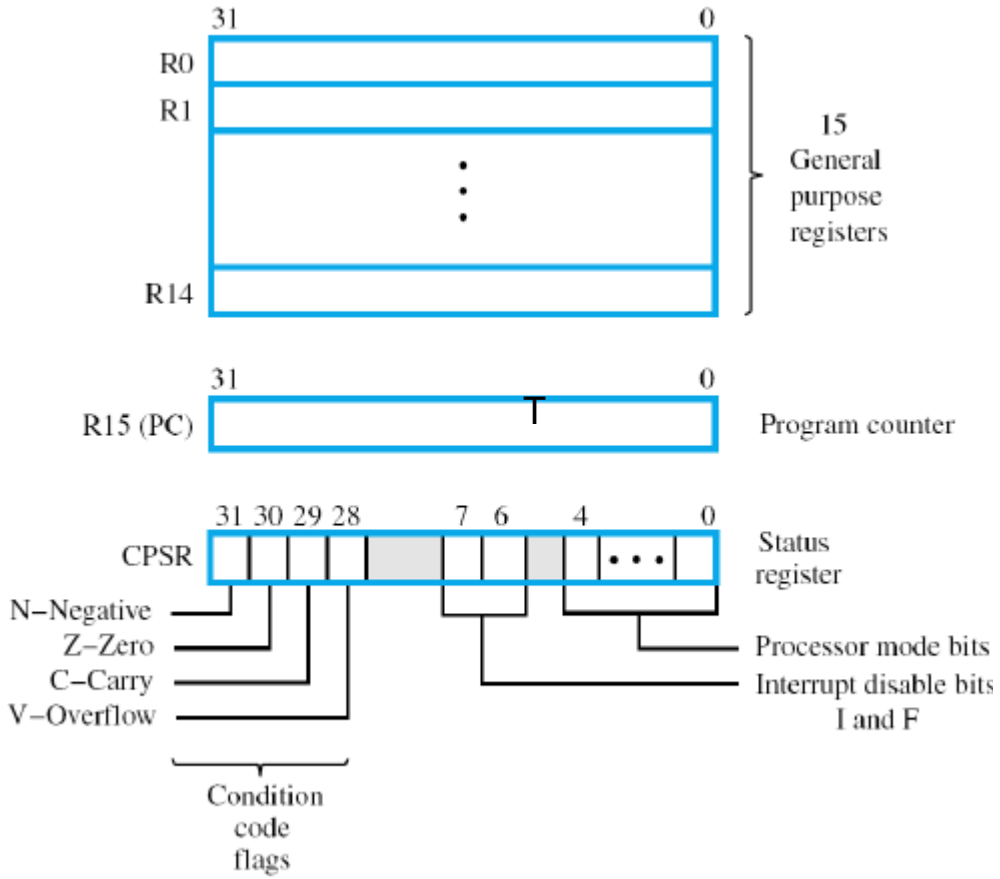
- È funzionalmente equivalente al linguaggio macchina, ma usa “nomi” più intuitivi (mnemonics)
- Definisce l’Instruction Set Architecture (ISA) della macchina
- Un compilatore traduce un linguaggio di alto livello, che è indipendente dall’architettura, in linguaggio assembly, che è dipendente dall’architettura
- Un assemblatore traduce programmi in linguaggio assembly in codice binario eseguibile
- Nel caso di linguaggi compilati (es. C) il codice binario viene eseguito direttamente dalla macchina target
- Nel caso di linguaggi interpretati (es. Java) il bytecode viene interpretato dalla Java Virtual Machine, che è al livello Assembly language

Ciclo di sviluppo



Programming with Advanced risc machine (ARM)

Registers



Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

ARM7TDMI Instruction Set*

Mnemonic	Instruction	Action	See Section:
ADC	Add with carry	Rd := Rn + Op2 + Carry	4.5
ADD	Add	Rd := Rn + Op2	4.5
AND	AND	Rd := Rn AND Op2	4.5
B	Branch	R15 := address	4.4
BIC	Bit Clear	Rd := Rn AND NOT Op2	4.5
BL	Branch with Link	R14 := R15, R15 := address	4.4
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]	4.3
CDP	Coprocesor Data Processing	(Coprocessor-specific)	4.14
CMN	Compare Negative	CPSR flags := Rn + Op2	4.5
CMP	Compare	CPSR flags := Rn - Op2	4.5
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)	4.5
LDC	Load coprocessor from memory	Coprocessor load	4.15
LDM	Load multiple registers	Stack manipulation (Pop)	4.11
LDR	Load register from memory	Rd := (address)	4.9, 4.10
MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}	4.16
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn	4.7, 4.8
MOV	Move register or constant	Rd : = Op2	4.5
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}	4.16
MRS	Move PSR status/flags to register	Rn := PSR	4.6
MSR	Move register to PSR status/flags	PSR := Rm	4.6
MUL	Multiply	Rd := Rm * Rs	4.7, 4.8
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2	4.5
ORR	OR	Rd := Rn OR Op2	4.5

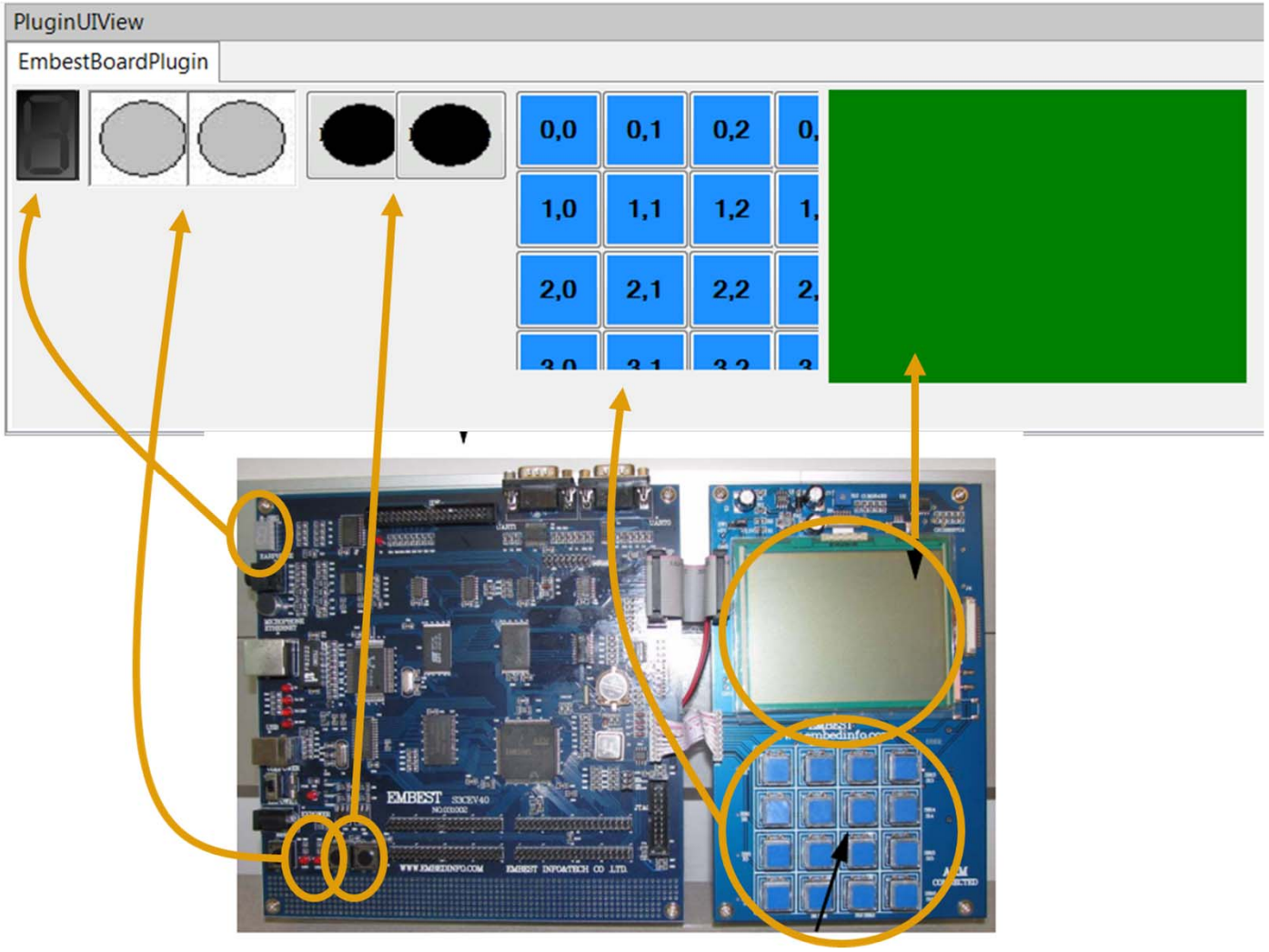
Mnemonic	Instruction	Action	See Section:
RSB	Reverse Subtract	Rd := Op2 - Rn	4.5
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry	4.5
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry	4.5
STC	Store coprocessor register to memory	address := CRn	4.15
STM	Store Multiple	Stack manipulation (Push)	4.11
STR	Store register to memory	<address> := Rd	4.9, 4.10
SUB	Subtract	Rd := Rn - Op2	4.5
SWI	Software Interrupt	OS call	4.13
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm	4.12
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2	4.5
TST	Test bits	CPSR flags := Rn AND Op2	4.5

***ARM 7TDMI Data Sheet –**
Copyright Advanced RISC Machines Ltd (ARM) 1995

ARMSim# simulator

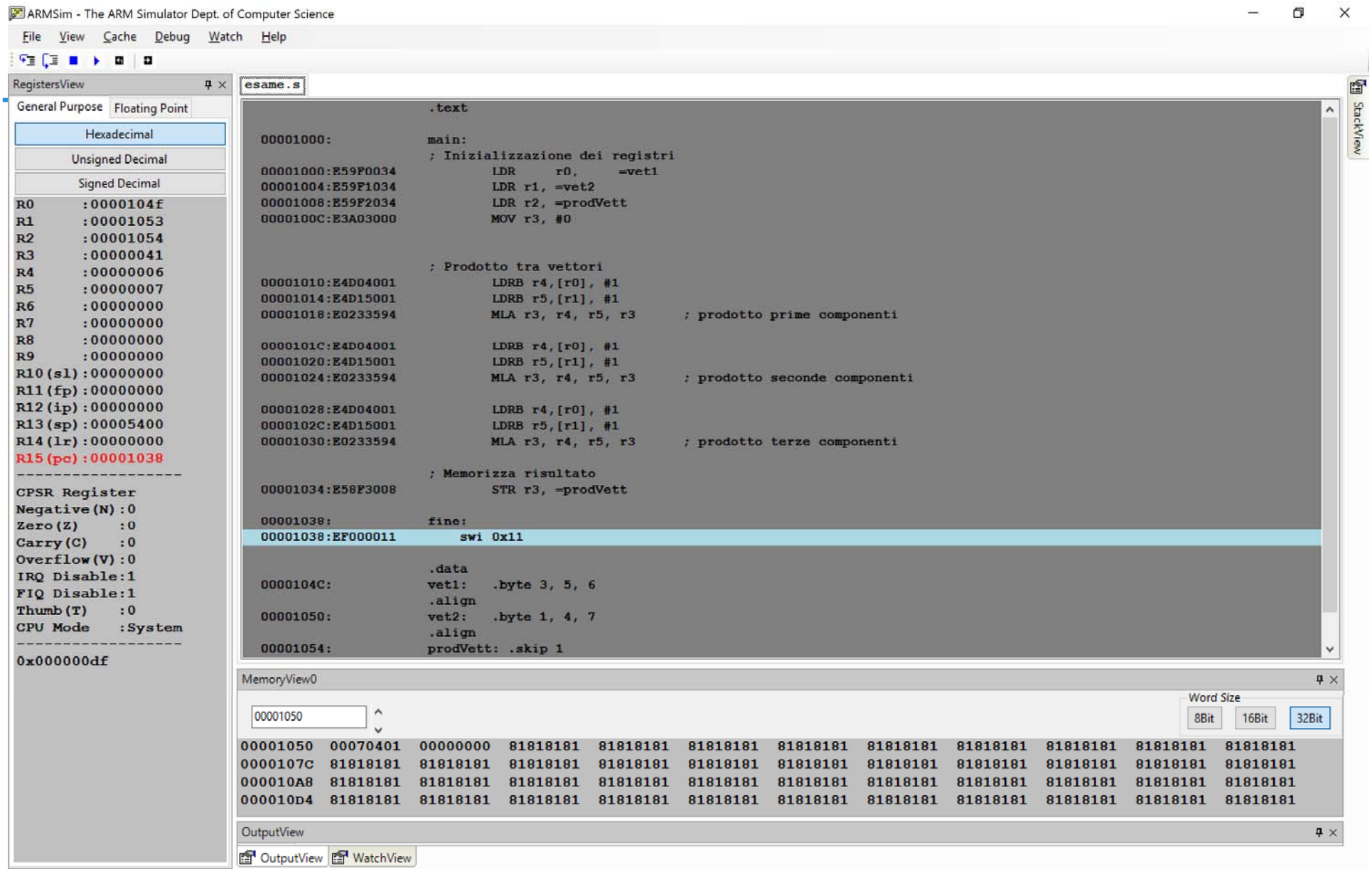
- ARMSim# sviluppato dal Department of Computer Science at the University of Victoria, in Victoria, British Columbia, Canada
  - http://armsim.cs.uvic.ca/
  - Non include un editor
  - Include la possibilità di simuare una scheda HW
  - Include una guida pdf

EmbestBoardPlugin



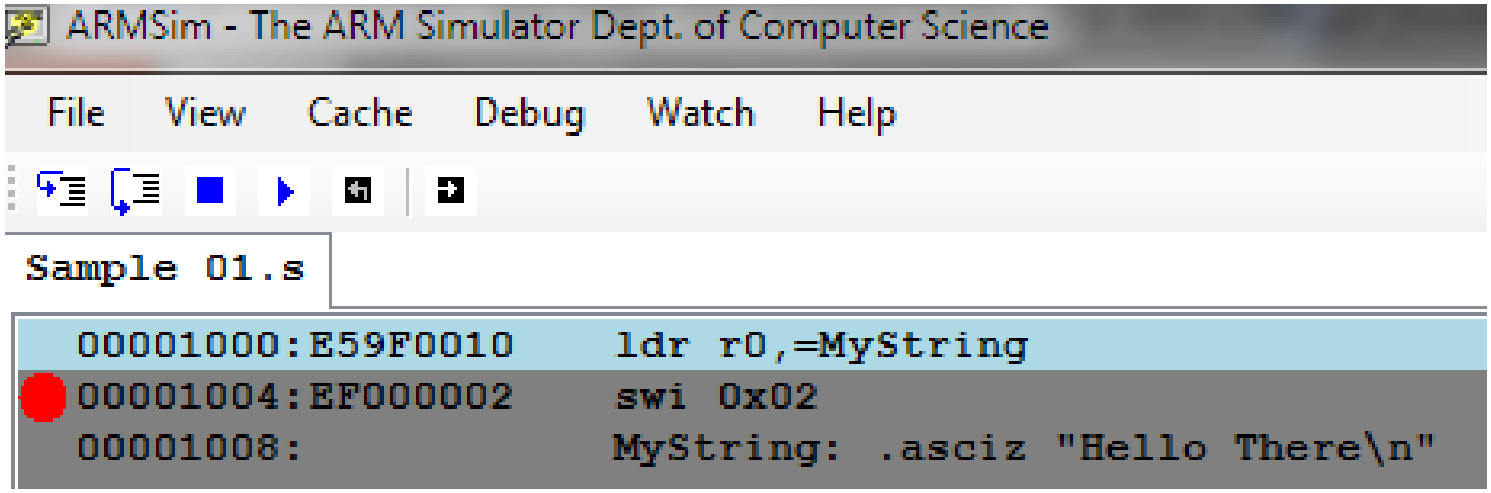
ARMSim#

- Preparazione del codice:
  - Utilizzare un editor esterno
  - Scrivere il codice ARM7tdmi
  - Salvare con estensione ".s"
- Lanciare ARMSim#
  - Aprire il file ".s"
  - Il programma viene assemblato e caricato
  - Procedere all'esecuzione del codice


ARMSim - The ARM Simulator Dept. of Computer Science
File View Cache Debug Watch Help
RegistersView
General Purpose Floating Point
Hexadecimal
Unsigned Decimal
Signed Decimal
R0 :0000104f
R1 :00001053
R2 :00001054
R3 :00000041
R4 :00000006
R5 :00000007
R6 :00000000
R7 :00000000
R8 :00000000
R9 :00000000
R10(sl):00000000
R11(fp):00000000
R12(ip):00000000
R13(sp):00005400
R14(lr):00000000
R15(pc):00001038
CPSR Register
Negative(N):0
Zero(Z) :0
Carry(C) :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T) :0
CPU Mode :System
0x000000df
esame.s
.text
00001000: main:
; Inizializzazione dei registri
00001000:E59F0034 LDR r0, =vet1
00001004:E59F1034 LDR r1, =vet2
00001008:E59F2034 LDR r2, =prodVett
0000100C:E3A03000 MOV r3, #0
; Prodotto tra vettori
00001010:E4D04001 LDRB r4,[r0], #1
00001014:E4D15001 LDRB r5,[r1], #1
00001018:E0233594 MLA r3, r4, r5, r3 ; prodotto prime componenti
0000101C:E4D04001 LDRB r4,[r0], #1
00001020:E4D15001 LDRB r5,[r1], #1
00001024:E0233594 MLA r3, r4, r5, r3 ; prodotto seconde componenti
00001028:E4D04001 LDRB r4,[r0], #1
0000102C:E4D15001 LDRB r5,[r1], #1
00001030:E0233594 MLA r3, r4, r5, r3 ; prodotto terze componenti
; Memorizza risultato
00001034:E58F3008 STR r3, =prodVett
00001038: fine:
00001038:EF000011 swi 0x11
.data
0000104C: vet1: .byte 3, 5, 6
.align
00001050: vet2: .byte 1, 4, 7
.align
00001054: prodVett: .skip 1
StackView
MemoryView0
00001050
Word Size
8Bit 16Bit 32Bit
00001050 00070401 00000000 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181
0000107C 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181
000010A8 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181
000010D4 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181 81818181
OutputView
OutputView WatchView

Simulation Run

- La prima icona **Step Into** consente di eseguire una singola istruzione, entrando in eventuali subroutines
- La seconda icona **Step Over** consente di eseguire una singola istruzione, senza entrare in eventuali subroutines
- **Breakpoint:** cliccare due volte su una riga di codice (viene aggiunto un pallino rosso). L'esecuzione del codice si interrompe all'istruzione del breakpoint



Simulation Views

Code View	It displays the assembly language instructions of the program that is currently open. This view is always visible and cannot be closed.
Registers View	It displays the contents of the 16 general-purpose user registers available in the ARM processor, as well as the status of the Current Program Status Register (CPSR) and the condition code flags. The contents of the registers can be displayed in hexadecimal, unsigned decimal, or signed decimal formats. Additionally the contents of the Vector Floating Point Coprocessor (VFP) registers can be displayed. They include the overlapped Single Precision Registers (s0-s31) and the Double Precision Floating Point Registers (d0-d15).
Output View: Console	It displays any automatic success and error messages produced by the simulator.
Output View: Stdin/Stdout/Stderr	It displays any text printed to standard output, Stdout.
Stack View	It displays the contents of the system stack. In this view, the top word in the stack is highlighted.
Watch View	It displays the values of variables that the user has added to the watch list, that is, the list of variables that the user wishes to monitor during the execution of a program.
Cache Views	They display the contents of the L1 cache. This cache can consist of either a unified data and instruction cache, displayed in the **Unified Cache View**, or separate data and instruction caches, displayed in the **Data Cache** and **Instruction Cache Views**, respectively, depending on the cache properties selected by the user.
Board Controls View	It displays the user interfaces of any loaded plug-ins. If no plug-ins were loaded at application start, this view is disabled.
Memory View	It displays the contents of main memory, as 8-bit, 16-bit, or 32-bit words. There can be multiple memory views, each displaying a different region of memory.

Registers written during the last instruction appear red colored in the Register view window

Esercizio

Sommare elementi di un vettore

```
    .text
ENTRY:
main:
    ADR     r1, array
    SUB     r1,r1,#4
    LDR     r0, =N
    LDR     r0,[r0]
    MOV     r4, #0
loop:
    SUBS    r0, r0, #1
    LDR     r3, [r1,#4]!
    ADD     r4, r4, r3
    BNE     loop
    SWI     0x11
array: .word 1,2,3,4,5
N:          .word 5
    .end
```

Cercare un carattere in una stringa

```
while !trovato
   if (s[i++]==c){
      trovato=true;
   }
}
if (!trovato) i=0;
```

Cercare un carattere in una stringa

```
while !trovato
   if (s[i++]==c){
      trovato=true;
   }
}
if (!trovato) i=0;
```

```
        .text
main:   ADR   r1, str
        ADR   r0, c
        LDRB  r0,[r0]
        MOV   r4, #0
cerca: LDRB  r2,[r1],#1
        CMP   r2,#0
        BEQ   nontrovato
        ADD   r4,r4,#1
        CMP   r2,r0
        BNE cerca
trovato:  ADD   r4,r4,#-1
        B     fine
nontrovato:  MOV   r4,#-1
fine:
        SWI   0x11   ; and exit
str:    .asciz "Tofadfadedfafd"
c:      .asciz "e"
        .end
```

Scrivere un programma che dato un vettore di N numeri, conti il numero di elementi pari nel vettore

```
conta = 0


for (i = 0; i< N, i++){
     if (v[i]%2=0) conta++;
}
```

Scrivere un programma che dato un vettore di N numeri, conti il numero di elementi pari nel vettore

```
conta = 0


for (i = 0; i< N, i++){
     if (v[i]%2=0) conta++;
}
```

```
        .text
ENTRY:
main:   ADR     r1, array
        LDR     r0, =N
        LDR     r0,[r0]
        MOV     r4, #0
loop:   LDR     r3, [r1],#4
        TST     r3, #1
        BEQ             pari
next:
        SUBS    r0, r0, #1
        BNE     loop
        SWI             0x11
pari:
        ADD             r4,r4,#1
        B               next
        .data
array:  .word 1,2,3,4,5
N:              .word 5
        .end
```

Esercizi

- Scrivere un programma che data una stringa in memoria inverta la stringa ponendola in una seconda area di memoria

SWI: software interrupt

- The SWI codes numbered in the range 0 to 255 inclusive are reserved for basic instructions that ARMSim# needs for I/O and should not be altered. Their list is shown in Table.The use of “EQU” is strongly advised to substitute the actual numerical code values

The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. A SWI handler should return by executing the following irrespective of the state (ARM or Thumb):

MOV PC, R14_svc

This restores the PC and CPSR, and returns to the instruction following the SWI.

NOTE: ARMSim# already provides handlers for the Interrupt codes in the table

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x00	Display Character on Stdout	r0: the character		SWI_PrChr
swi 0x02	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
swi 0x11	Halt Execution			SWI_Exit
swi 0x12	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	SWI_MeAlloc
swi 0x13	Deallocate All Heap Blocks			SWI_DAlloc
swi 0x66	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	SWI_Open
swi 0x68	Close File	r0: file handle		SWI_Close
swi 0x69	Write String to a File or to Stdout	r0: file handleor Stdout r1: address of a null terminated ASCII string		SWI_PrStr

Opcode	Description and Action	Inputs	Outputs	EQU
swi 0x6a	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	SWI_RdStr
swi 0x6b	Write Integer to a File	r0: file handle r1: integer		SWI_PrInt
swi 0x6c	Read Integer from a File	r0: file handle	r0: the integer	SWI_RdInt
swi 0x6d	Get the current time (ticks)		r0: the number of ticks (milliseconds)	SWI_Timer

SWI: standard I/O

- Output View provides a tab labelled “Stdin/Stdout/Stderr” where output from the user program is displayed as a result of using software interrupts (SWI instructions) to perform I/O.
- SWI 0x02: display a null terminated ASCII string in r0 on StdOut

```
ldr       r0,=MyString      load register r0 to MyString label
swi       0x02              execute swi 0x02
MyString: .asciz "Hello There\n"
```

- SWI 0x66, read a file:

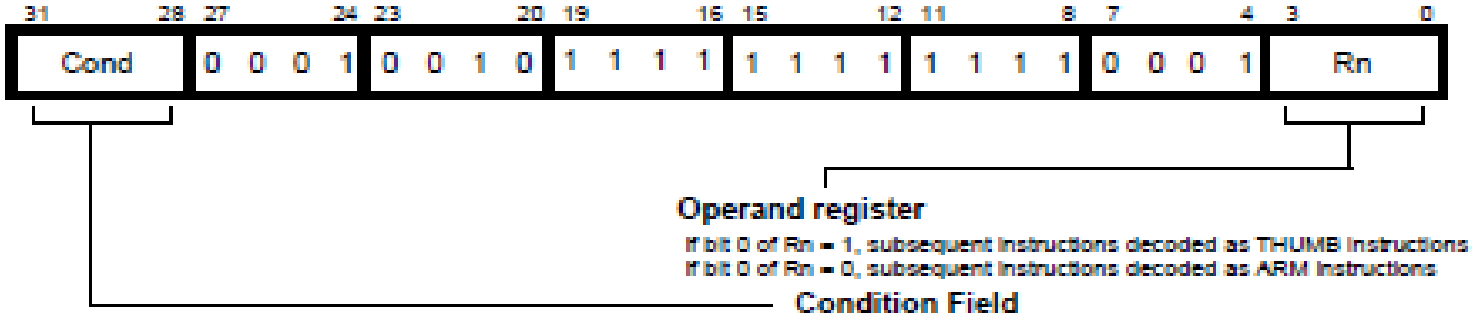
```
InFileName:     .asciz   "Infile1.txt"
InFileError:    .asciz   "Unable to open input file\n"
                .align
InFileHandle:   .word    0

                ldr      r0,=InFileName      @ set Name for input file
                mov      r1,#0               @ mode is input
                swi      SWI_Open            @ open file for input
                bcs      InFileError         @ if error?
                ldr      r1,=InFileHandle    @ load input file handle
                str      r0,[r1]             @ save the file handle
```

Esempio

HelloWorld

Conditions



Conditions remove the need for many branches:

- Stall the pipeline
- Allows very dense in-line code, without branches.
- Increase the number of instructions

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Conditions

- To execute an instruction conditionally, simply postfix it with the appropriate condition:

For example an add instruction takes the form:

– ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)

To execute this only if the zero flag is set:

– ADDEQ r0,r1,r2 ; If zero flag set then…

; ... r0 = r1 + r2

- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by post-fixing the instruction (and any condition code) with an “S”.

```
ADDS R1,#1      ; Add 1 to register 1, setting CPSR flags
                ; on the result then call subroutine if
BLCC sub        ; the C flag is clear, which will be the
                ; case unless R1 held 0xFFFFFFFF.
```

Condition codes

- **C language:**

```
if (a < b) { x = 5; y = c + d; } else x = c - d;
```

- **ARM:**

```
        ; compute and test condition
        ADR r4,a              ; get address for a
        LDR r0,[r4]           ; get value of a
        ADR r4,b              ; get address for b
        LDR r1,[r4]           ; get value for b
        CMP r0,r1             ; compare a < b
        BGE fblock            ; if a >= b, branch to false block
        …                     ; true block
        B After               ; branch after the false
                              ; block
false_block …                 ; false block instructions
After       …                 ; continue
```

- **C language:**

```
for ( i = 0 ; i < 15 ; i++){
  j = j + j;
}
```

- **ARM:**

```
        SUB    R0, R0, R0          ; i -> R0 and i = 0
start   CMP R0, #15                ; is i < 15?
        ADDLT R1, R1, R1           ; j = j + j
        ADDLT R0, R0, #1           ; i++
        BLT start
```

- **C language:**

```
if (i == 0) {
  i = i +10;
}
```

- **ARM:** (assume i in R1)

```
SUBS    R1, R1, #0
ADDEQ R1, R1, #10
```