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

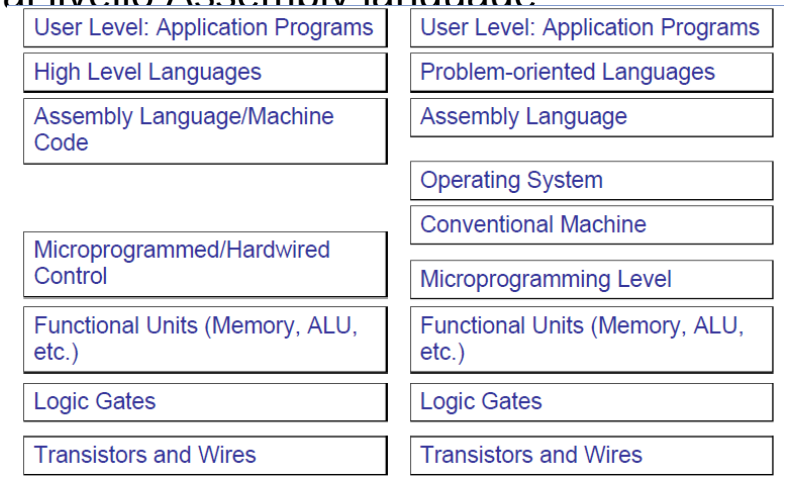
Tel: +39-081-5476702
Fax: +39-081-5476777

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

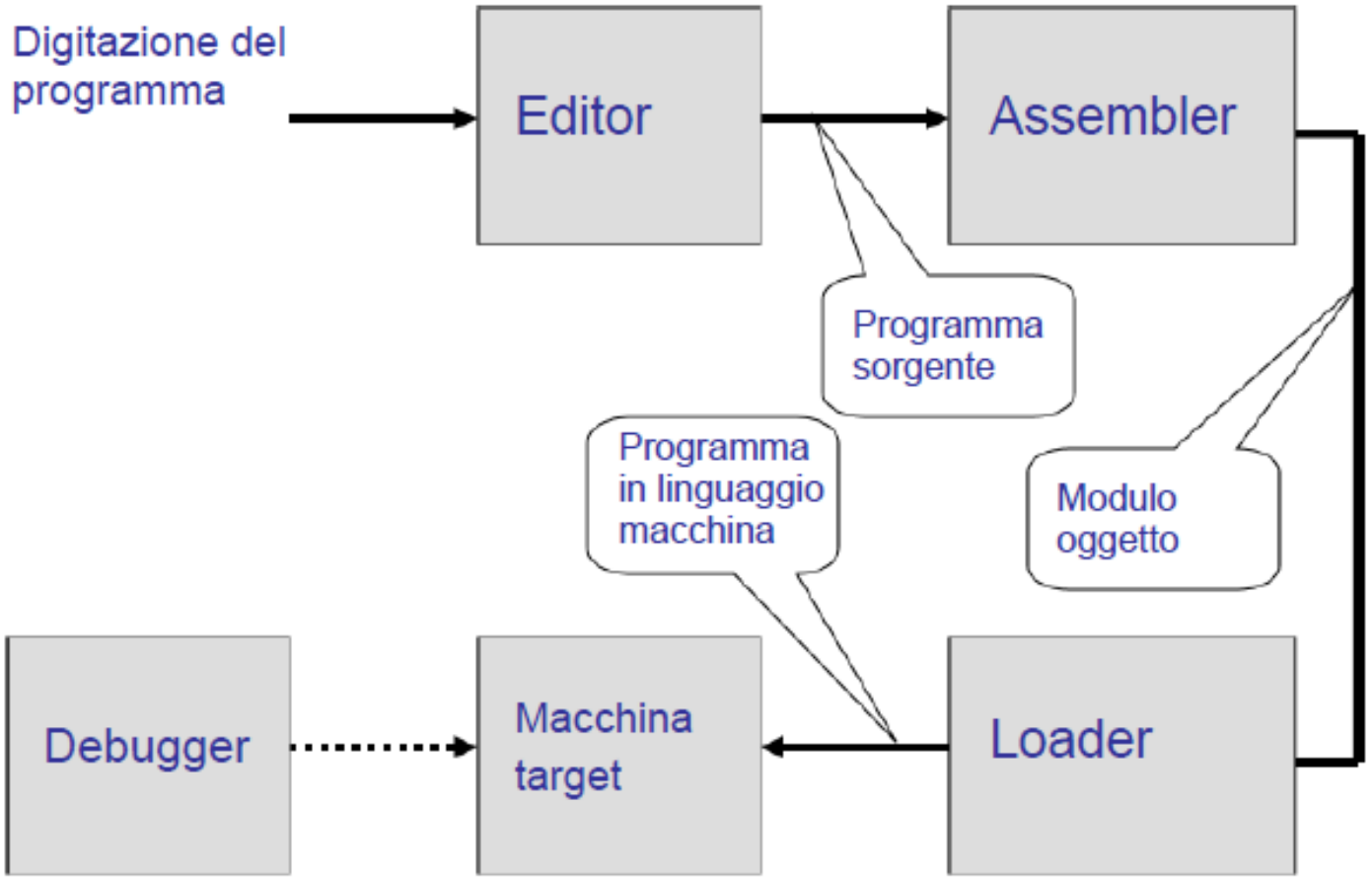
- È funzionalmente equivalente al linguaggio macchina, ma usa “nomi” più intuitivi (mnemonics)
- Definisce l’Instruction Set Architecture (ISA) della macchina
- Un compilatore traduce un linguaggio di alto livello, che è indipendente dall’architettura, in linguaggio assembly, che è dipendente dall’architettura
- 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

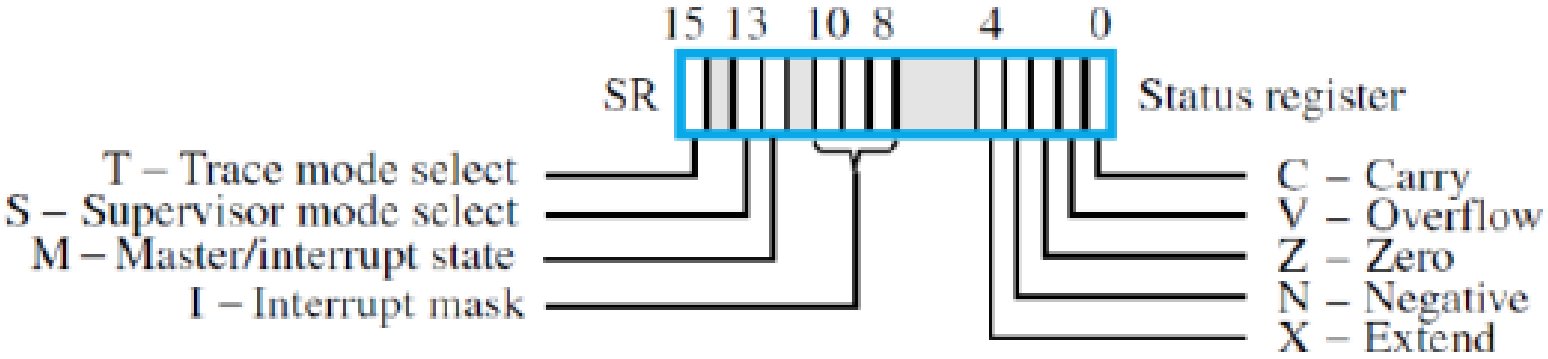
Ciclo di sviluppo



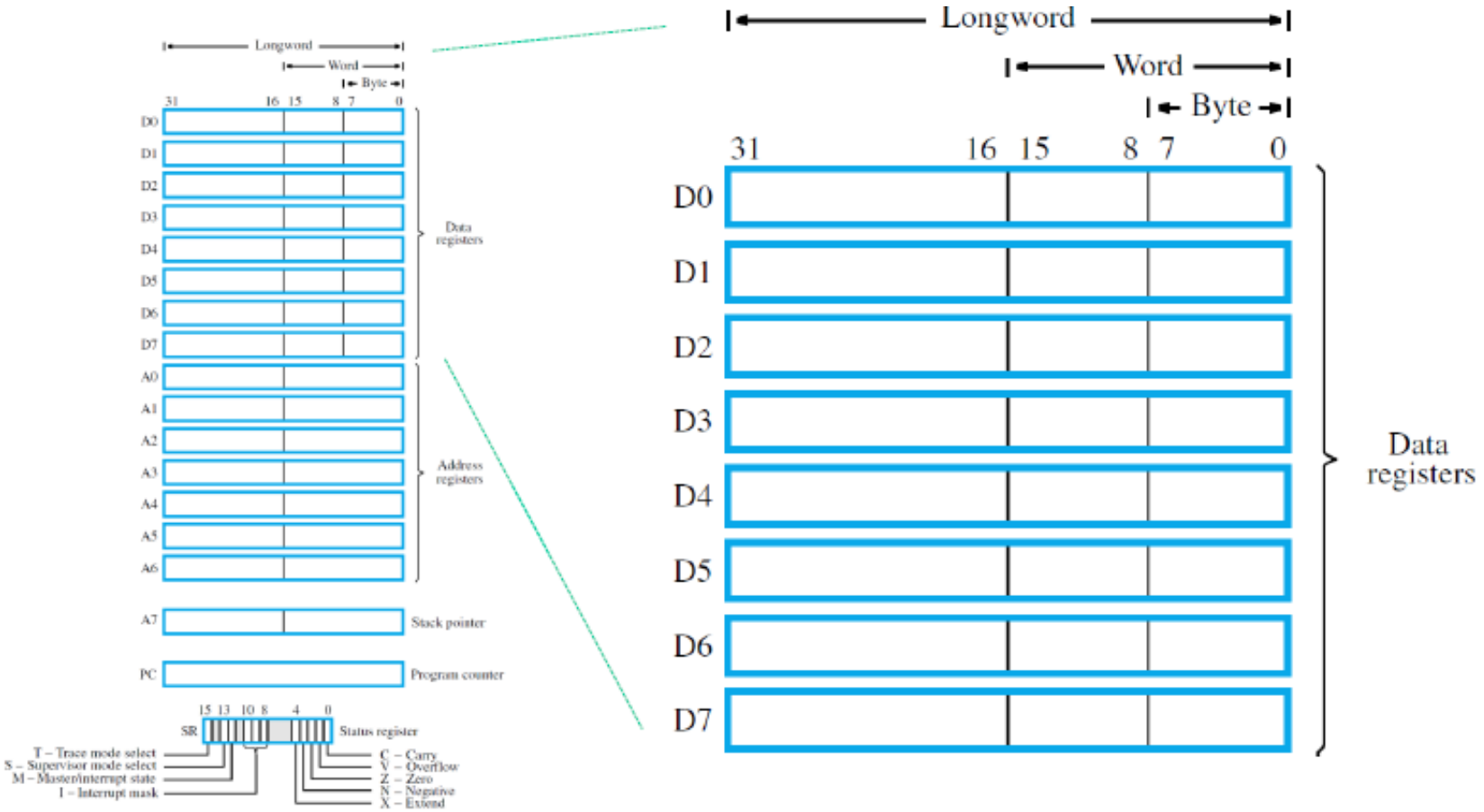
Programming with Motorola 68000

Registers and memory organization

- Byte-addressable, 32-bit address space
- Big-endian addressing scheme
- Longword (32-bit), word (16-bit), and byte (8-bit) sizes for integer data (.L, .W, .B)
- Eight data registers, D0 to D7
- Eight address registers, A0 to A7, and register A7 is the stack pointer (SP)
- Status register (SR) with condition codes
- Program Counter (PC)



Register Structure



Easy68k Simulator

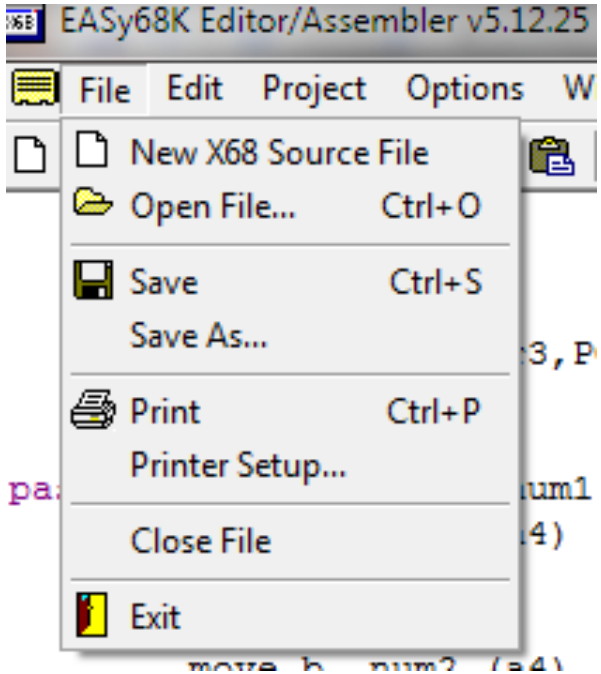
- EASy68K is an open source 68000 Structured Assembly Language IDE, GNU public licensed.

http://www.easy68k.com/

- Components:
  - Edit68K Editor/Assembler
  - Sim68K Simulator
  - EASyBIN binary editor

Code editing

- Tool: Edit68K
- Create new source code or open existing source files (.x68,.s68,.m68,.l68)
- It is possible defining the Starting code template: Options-> Assembler Option
- The editor provides code coloring
- Support:
  - Help
  - Easy68k Quick Ref (pdf)
  - Summary Instruction Set 68000 (pdf)



Formato codice sorgente

Una linea di codice sorgente Assembly è costituita da quattro campi:

- **LABEL**
  - Stringa alfanumerica
  - Definisce un nome simbolico per il corrispondente indirizzo
- **OPCODE**
  - Codice mnemonico o pseudo-operatore
  - Determina la generazione di un'istruzione in linguaggio macchina o la modifica del valore corrente del Program Location Counter
- **OPERANDS**
  - Oggetti dell'azione specificata dall'OPCODE
  - Variano a seconda dell'OPCODE e del modo di indirizzamento
- **COMMENTS**
  - Testo arbit

```
*-----------------------------------------------------------
* Program Number: Final Project
* Written by    : Alan Swanson
* Date Created  : 1-29-05
* Description   : Enter a string of no more than 80 charaters
*       lower or upper case and convert to upper case
*       and scroll the string on the hardware display
*-----------------------------------------------------------
START   ORG     $1000

*Title Message
                move.b  #14,d0          escape code to print a string
                lea     (TitleMsg,PC),a1        point to address of message
                trap    #15             print to screen
```

Convenzioni

- Gli spazi bianchi tra i diversi campi fungono esclusivamente da separatori (vengono ignorati dall'assemblatore)
- Una linea che inizi con un asterisco (*) è una linea di commento
- Nelle espressioni assembly, gli argomenti di tipo numerico si intendono espressi
  - In notazione decimale, se non diversamente specificato
  - In notazione esadecimale, se preceduti dal simbolo "$"
  - In notazione binaria (0,1) se preceduti da %
- Nell'indicazione degli operandi, il simbolo "#" denota un indirizzamento immediato

Program Location Counter (PLC)

- E’ una variabile interna dell’assemblatore
- Punta alla locazione di memoria in cui andrà caricata l’istruzione assemblata
- Viene inizializzato dallo pseudo-operatore “origin” (ORG)
- Durante il processo di assemblaggio, il suo valore è aggiornato sia in funzione degli operatori, sia in funzione degli pseudo-operatori
- E’ possibile, all’interno di un programma, fare riferimento al suo valore corrente, mediante il simbolo “*”

Assembled code

```
00001000 Starting Address
Assembler used: EASy68K Editor/Assembler v5.12.25
Created On: 02/04/2013 10:52:28

PLC        Content      Line   Label  Opcode   Operands          Comment
00000000                 1  *-----------------------------------------------------------
00000000                 2  * Program Number: Final Project
00000000                 3  * Written by    : Alan Swanson
00000000                 4  * Date Created  : 1-29-05
00000000                 5  * Description   : Enter a string of no more than 80 charaters
00000000                 6  *       lower or upper case and convert to upper case
00000000                 7  *       and scroll the string on the hardware display
00000000                 8  *-----------------------------------------------------------
00001000                 9  START   ORG     $1000
00001000                 10
00001000                 11  *Title Message
00001000  103C 000E     12              move.b   #14,d0              escape code to print a string
00001004  43FA 0433     13              lea      (TitleMsg,PC),a1    point to address of message
00001008  4E4F          14              trap     #15                 print to screen
```

Instruction format

- Un’istruzione si compone di:
  - un codice operativo (OPCODE)
  - zero, uno o più operandi

Tipi di operandi:

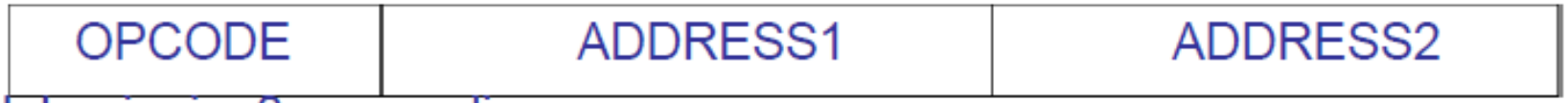
- operando costante
  - esplicito (immediato)
  - implicito (es. 0)
- operando memoria
- operando registro
  - esplicito (es. R1)
  - implicito (es. accumulatore)
- E’ un **set di istruzioni ortogonali**: il metodo per specificare l’indirizzo dell’operando è indipendente dall’opcode

Instruction format

Istruzioni a 0 operandi

Istruzioni a 1 operando

Istruzioni a 2 operandi



Istruzioni a 3 operandi

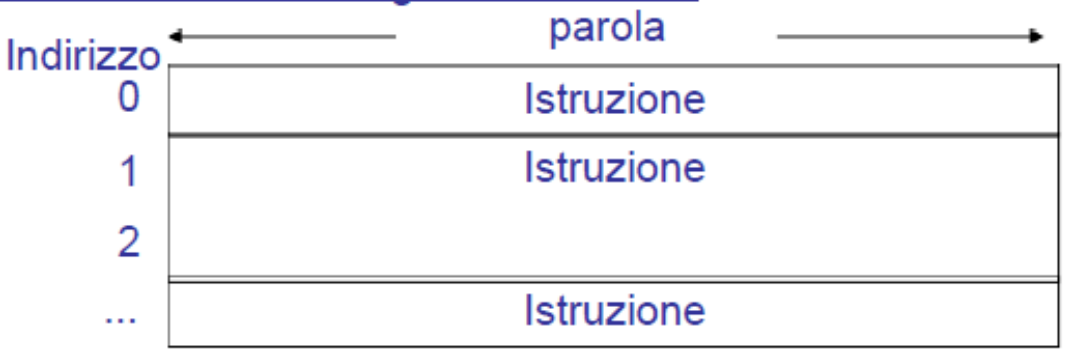
Le istruzioni possono essere tutte della stessa lunghezza in bit (codifica a lunghezza fissa) oppure possono essere di lunghezze differenti (codifica a lunghezza variabile)

Instruction format

- Sono possibili diverse relazioni tra la lunghezza dell'istruzione e la lunghezza della parola del processore

Codifica delle istruzioni a lunghezza fissa

Codifica delle istruzioni a lunghezza variabile



Formato delle istruzioni: MC68000

- istruzioni di lunghezza variabile da 1 a 5 parole da 16 bit
- la prima parola fornisce codice operativo, modo di indirizzamento e lunghezza dell'istruzione
- esistono differenti formati di codifica dell'opcode, di diversa lunghezza
- le parole successive contengono un operando immediato e/o un indirizzo di un operando memoria

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

MOVE

- es. istruzione MOVE src,dst
  - [src] -> dst
  - src e dst possono essere operandi registro-registro, registro-memoria, memoria-registro o memoria-memoria
  - op = 00
  - size = 01 => byte size = 11 => word size = 10 => long
  - le word che cominciano con 0000 sono utilizzate per altre istruzioni
  - in definitiva 3/16 dello spazio dei codici operativi è usato da MOVE

Pseudo-operators

ORG

- Initialize the Program Location Counter (PLC)
  - Syntax: ORG $HEXADDR

END

- The end directive tells the assembler that the end of a program has been reached. The operand specifies the starting address of the program (e.g. a label START).
  - Syntax: END start_address

Pseudo-operators

DS – Define Storage

- Increment the Program Location Counter (PLC) to reserve memory space for additional variables. The Define Storage directive reserves the specified amount of memory at the current location. DS is qualified by .B, .W, or .L and defaults to .W if no size is specified. Unlike DC, no data is stored in the reserved memory

```
00001310                        267  buffer1         ds.b    80              first string
00001360                        268                  ds.w    0
00001360                        269  buffer2         ds.b    80              second string
000013B0                        270                  ds.w    0
```

DC – Define Constant

- Initialize a variable value. The DC directive instructs the assembler to place the following values into memory at the current location. The 68000 microprocessor requires that word and long word numbers be stored in even memory addresses. The assembler will adjust the memory locations accordingly

```
00001416                        289  *Messages
00001416= 0D 0A 0D 0A 54 68 ...  290  UpMsg           dc.b    CR,LF,CR,LF,'The new uppercase string is ',CR,LF,0
00001439= 54 68 69 73 20 70 ...  291  TitleMsg        dc.b    'This program scrolls strings to the hardware device',CR,LF,CR,LF
00001470= 50 72 65 73 73 20 ...  292                  dc.b    'Press the toggle switches to control the speed',CR,LF,CR,LF,0
000014A3= 45 6E 74 65 72 20 ...  293  PromptMsg       dc.b    'Enter a string lower or upper case 80 charaters max',CR,LF,0
```

EQU

- Define an identity – Similar to defining a constant in C++

```
00001310                        264  *Constants
00001310  =0000000D             265  CR              equ     $0d             cariage return
00001310  =0000000A             266  LF              equ     $0a             line feed
```

Instructions: CLR

CLR — Clear an operand

Operation: `[destination] ← 0`

Syntax: `CLR <ea>`

Sample syntax: `CLR (A4)+`

Attributes: Size = byte, word, longword

Description: The destination is cleared — loaded with all zeros. The `CLR` instruction can't be used to clear an address register. You can use `SUBA.L A0,A0` to clear A0. Note that a side effect of `CLR`'s implementation is a *read* from the specified effective address before the clear (i.e., write) operation is executed. Under certain circumstances this might cause a problem (e.g., with write-only memory).

Condition codes:

X	N	Z	V	C
-	0	1	0	0

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Instructions: MOVE

MOVE Copy data from source to destination

Operation: [destination] ← [source]

Syntax: MOVE <ea>,<e>

Sample syntax:
```
MOVE (A5),-(A2)
MOVE -(A5),(A2)+
MOVE #$123,(A6)+
MOVE Temp1,Temp2
```

Attributes: Size = byte, word, longword

Description: Move the contents of the source to the destination location. The data is examined as it is moved and the condition codes set accordingly. Note that this is actually a *copy* command because the source is not affected by the move. The move instruction has the widest range of addressing modes of all the 68000's instructions.

Condition codes:

X	N	Z	V	C
-	*	*	0	0

Condition Code Register (CCR) values:

U The state of the bit is undefined (i.e., its value cannot be predicted)
- The bit remains unchanged by the execution of the instruction
* The bit is set or cleared according to the outcome of the instruction.

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Destination operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
✔		✔	✔	✔	✔	✔	✔	✔			

Instructions: LEA

LEA — Load effective address

Operation: `[An] ← <ea>`

Syntax: `LEA <ea>,An`

Sample syntax:

```
LEA Table,A0
LEA (Table,PC),A0
LEA (-6,A0,D0.L),A6
LEA (Table,PC,D0),A6
```

Attributes: Size = longword

Description: The effective address is computed and loaded into the specified address register. For example, `LEA (-6,A0,D0.W),A1` calculates the sum of address register A0 plus data register D0.W sign-extended to 32 bits minus 6, and deposits the result in address register A1. The difference between the `LEA` and `PEA` instructions is that `LEA` calculates an effective address and puts it in an address register, while `PEA` calculates an effective address in the same way but pushes it on the stack.

Application: `LEA` is a very powerful instruction used to calculate an effective address. In particular, the use of `LEA` facilitates the writing of position independent code. For example, `LEA (TABLE,PC),A0` calculates the effective address of 'TABLE' with respect to the PC and deposits it in A0.

```
LEA  (Table,PC),A0     Compute address of Table with respect to PC
MOVE (A0),D1           Pick up the first item in the table
.                      Do something with this item
MOVE D1,(A0)           Put it back in the table
.
.
Table DS.B  100
```

Instructions: BRANCH

Bcc — Branch on condition cc

Operation: `If cc = 1 THEN [PC] ← [PC] + d`

Syntax: `Bcc <label>`

Sample syntax:
```
BEQ Loop_4
BVC *+8
```

Attributes: `BEQ` takes an 8-bit or a 16-bit offset (i.e., displacement).

Description: If the specified logical condition is met, program execution continues at location [PC] + displacement, d. The displacement is a two's complement value. The value in the PC corresponds to the current location plus two. The range of the branch is -126 to +128 bytes with an 8-bit offset, and -32K to +32K bytes with a 16-bit offset. A short branch to the next instruction is impossible, since the branch code 0 indicates a long branch with a 16-bit offset. The assembly language form `BCC *+8` means branch to the point eight bytes from the current PC if the carry bit is clear.

BCC	branch on carry clear	$\overline{C}$
BCS	branch on carry set	C
BEQ	branch on equal	Z
BGE	branch on greater than or equal	$N.V + \overline{N}.\overline{V}$
BGT	branch on greater than	$N.V.\overline{Z} + \overline{N}.\overline{V}.\overline{Z}$
BHI	branch on higher than	$\overline{C}.\overline{Z}$
BLE	branch on less than or equal	$Z + N.\overline{V} + \overline{N}.V$
BLS	branch on lower than or same	$C + Z$
BLT	branch on less than	$N.\overline{V} + \overline{N}.V$
BMI	branch on minus (i.e., negative)	N
BNE	branch on not equal	$\overline{Z}$
BPL	branch on plus (i.e., positive)	$\overline{N}$
BVC	branch on overflow clear	$\overline{V}$
BVS	branch on overflow set	V

BRANCH

- Numbers can be interpreted as signed or unsigned:

The signed comparisons are:

BGE		branch on greater than or equal
BGT		branch on greater than
BLE		branch on lower than or equal
BLT		branch on less than

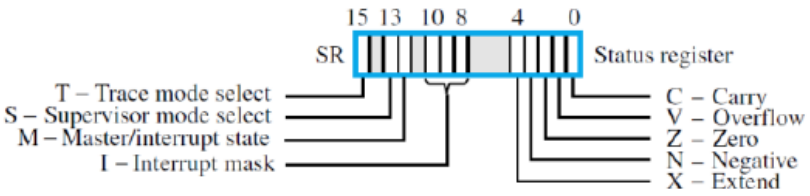
The unsigned comparisons are:

BHS	BCC	branch on higher than or same
BHI		branch on higher than
BLS		branch on lower than or same
BLO	BCS	branch on less than

E.g. #FF is higher than $10 if values are unsigned (255>16), but lower if signed (because signed #FF is -1)
Note that the Status Register contains the information used to verify the branch conditions which determine data representations

BRANCH conditions

Single bit	
BCS branch on carry set	C = 1
BCC branch on carry clear	C = 0
BVS branch on overflow set	V = 1
BVC branch on overflow clear	V = 0
BEQ branch on equal (zero)	Z = 1
BNE branch on not equal	Z = 0
BMI branch on minus (i.e., negative)	N = 1
BPL branch on plus (i.e., positive)	N = 0
Signed	
BLT branch on less than (zero)	N ⊕ V = 1
BGE branch on greater than or equal	N ⊕ V = 0
BLE branch on less than or equal	(N ⊕ V) + Z = 1
BGT branch on greater than	(N ⊕ V) + Z = 0
Unsigned	
BLS branch on lower than or same	C + Z = 1
BHI branch on higher than	C + Z = 0



⊕: XOR

Instructions: JUMP

JMP — Jump (unconditionally)

Operation: `[PC] ← destination`

Syntax: `JMP <ea>`

Attributes: Unsized

Description: Program execution continues at the effective address specified by the instruction.

Application: Apart from a simple unconditional jump to an address fixed at compile time (i.e., `JMP label`), the `JMP` instruction is useful for the calculation of *dynamic* or *computed* jumps. For example, the instruction `JMP (A0,D0.L)` jumps to the location pointed at by the contents of address register A0, offset by the contents of data register D0. Note that `JMP` provides several addressing modes, while `BRA` provides a single addressing mode (i.e., PC relative).

Condition codes:

X	N	Z	V	C
-	-	-	-	-

Source operand addressing modes

Dn	An	(An)	(An)+	-(An)	(d,An)	(d,An,Xi)	ABS.W	ABS.L	(d,PC)	(d,PC,Xn)	imm
		✔			✔	✔	✔	✔	✔	✔	

Instructions: CMP

CMP Compare

Operation: `[destination] - [source]`

Syntax: `CMP <ea>,Dn`

Sample syntax: `CMP (Test,A6,D3.W),D2`

Attributes: Size = byte, word, longword

Description: Subtract the source operand from the destination operand and set the condition codes accordingly. The destination must be a data register. The destination is not modified by this instruction.

Condition codes:

X	N	Z	V	C
-	*	*	*	*

Esercizio

Sommare elementi di un vettore

Scrivere, compilare ed eseguire il seguente codice

```
 ORG    $1000
START

* Put program code here
    MOVE.B  N,D0
    MOVE.W  #0,D1
    MOVE.B  #0,D3
    LEA     V,A0
loop
    ADD.B  (A0,D3),D1
    ADD.B   #1,D3
    SUB.B   #1,D0
    BNE loop
    SIMHALT  ;halt simulator

    ORG $2000
N   DC.B    5
V   DC.B    3,6,1,2,8
    END    START
```

Istruzioni di Selezione (1/2)

Linguaggio di alto livello:

```
if (espressione)
  istruzione
istruzione_successiva
```

Linguaggio assembler (processore MC 68000):

```
          B(NOT condizione) labelA
          istruzione
          ...
labelA    istruzione_successiva
```

Esempio:

```
if(D0==5)
   D1++;
D2=D0;
```

```
        CMPI.L  #5,D0
        BNE     SKIP
        ADDQ.L  #1,D1
SKIP    MOVE.L D0,D2
```

This approach is consequence of branching instruction mechanisms which require the interruption of program sequence

Istruzioni di Selezione (2/2)

Linguaggio di alto livello:

```
if (espressione)
      istruzione1
else  istruzione2
istruzione_successiva
```

Linguaggio assembler (processore MC 68000):

```
        B(NOT condizione) labelA
        istruzione1
        ...
        BRA labelB
labelA  istruzione2
     ...
labelB  istruzione_successiva
```

Costrutti iterativi (1/2)

Linguaggio di alto livello:

do
    istruzione
while (condizione == TRUE);
istruzione_successiva

Linguaggio assembler (processore MC 68000):

labelA istruzione
...
Bcc labelA
istruzione_successiva

Esempio: calcola 3^N (N>0)

```
D0 = 1; D1 = 1;
do {
        D0 = D0 * 3;
        D1++;
} while (D1 <= N);
```

```
        MOVE.B  #N,D2
        MOVE.B  #1,D1
        MOVE.W  #1,D0
LOOP    MULU.W  #3,D0
        ADDQ.B  #1,D1
CMP.B   D2,D1
        BLE     LOOP
```

Costrutti iterativi (2/2)

Linguaggio di alto livello:

```
while (condizione == TRUE)
        istruzione;
istruzione_successiva
```

Linguaggio assembler (processore MC 68000):

```
                BRA labelB
labelA istruzione
                ...
labelB Bcc labelA
```

Esempio: calcola 3^N (N>=0)

```
D0 = 1; D1 = 1;
while (D1 <= N) {
        D0 = D0 * 3;
        D1++;
};
```

```
MOVE.B  #N,D2
        MOVE.B  #1,D1
        MOVE.W  #1,D0
BRA     TEST
 LOOP   MULU.W  #3,D0
        ADDQ.B  #1,D1
 TEST   CMP.B   D2,D1
        BLE     LOOP
```

Cercare un carattere in una stringa

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

Cercare un carattere in una stringa

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

```
LOOP
   CMP.B   #1,D3
   BEQ     FINELOOP

   MOVE.B (A0)+,D2
   BEQ     FINELOOP ; se fine string => non trovato

   CMP.B   D0,D2
   BNE     NONTROV
   MOVE    #1,D3

NONTROV ADDQ.B #1,D1
   BRA     LOOP

FINELOOP
   CMP     #0,D3
   BNE     fine
   MOVE    #0,D1

fine
```

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++;
}
```

```
        MOVE.B  N,D0
        MOVE.B  #0,D1  ;contatore
        LEA     V,A0

loop
        MOVE.B  (A0)+,D2
        AND.B   #%00000001,D2
        BNE disp
        ADDQ.B  #1,D1
disp    ADD.B   #-1,D0
        BGT     loop

        SIMHALT             ; halt simulator
        ORG $2000
N       DC.B    5
V       DC.B    3,6,1,2,8
```

Esercizi

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

TRAP

- **TRAP:** software interrupt. It's synchronous since called by the program at a certain point of instruction sequence.
- TRAPs call routines pointed in the memory. The operation (task) performed by the routine is not specified and is typically demanded to OS or system BIOS implementation.
- Motorola 68000 provides 16 TRAPs (i.e. there is space for 16 predefined routines in each system)
- Easy68k simulator implements a set of tasks such as in the table below

Commonly Used Simulator Input/Output Tasks TRAP #15 is used to run simulator tasks. Place the task number in register D0. See Help for a complete description of available tasks. (cstring is null terminated)

0	Display n characters of string at (A1), n=D1.W (stops on NULL or max 255) with CR,LF	1	Display n characters of string at (A1), n=D1.W (stops on NULL or max 255) without CR,LF	2	Read characters from keyboard. Store at (A1). Null terminated. D1.W = length (max 80)	3	Display D1.L as signed decimal number
4	Read number from keyboard into D1.L	5	Read single character from keyboard in D1.B	6	Display D1.B as ASCII character	7	Set D1.B to 1 if keyboard input pending else set to 0
8	time in 1/100 second since midnight →D1.L	9	Terminate the program. (Halts the simulator)	10	Print cstring at (A1) on default printer.	11	Position cursor at row,col D1.W=ccrr, $FF00 clears
13	Display cstring at (A1) with CR,LF	14	Display cstring at (A1) without CR,LF	15	Display unsigned number in D1.L in D2.B base	17	Display cstring at (A1) , then display number in D1.L
18	Display cstring at (A1), read number into D1.L	19	Return state of keys or scan code. See help	20	Display ± number in D1.L, field D2.B columns wide	21	Set font properties. See help for details

TRAP example: printing to STDOUT

- TRAP #15 provides simulation tasks:
  - It executes the task number indicated in the D0 register
  - Each task performs operations on specific registers
- To print a string on Standard Output:
  - Task #14 prints a string pointed in A1 without CR and LF
  - Task #13 prints a string pointed in A1 with CR and LF

```
move      #14,d0                  task number 14 (display null string)
lea       textHello,a1            address of string
trap      #15
...
textHello dc.b    'Hello World',0 null terminated string
```

Esercizio

- Eseguire il programma helloWorld

TRAP example: reading from STDIN

- To read characters from keyboard:
  - Task #2 reads from keyboard and stores into A1 (null terminated). D1.W contains the string length (max 80).
  - Task #4 for numbers -> D1.L
  - Task #5 for single char -> D1.B

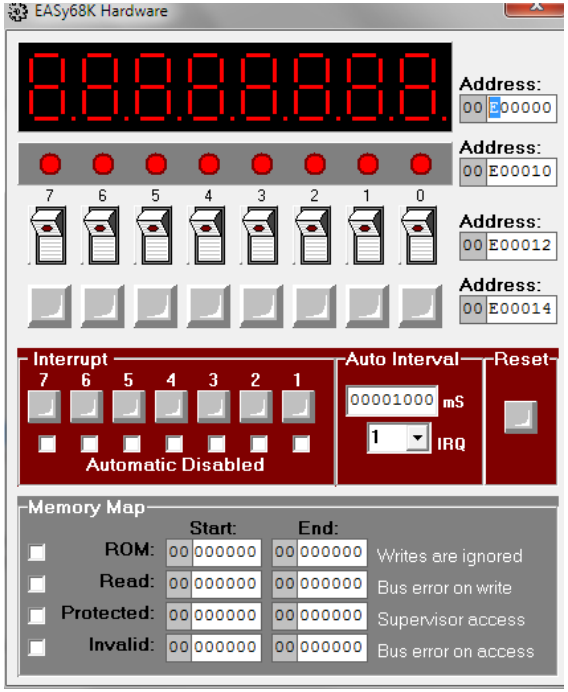
```
move.b   #2,d0              escape code to enter a string
lea      (buffer1,PC),a1    put the address of the buffer into a1
trap     #15                print to screen
move     d1,blen            length of string
...
buffer1  ds.b     80        first string
         ds.w     0
blen     ds.w     1         length of inputed string
```

Esercizio

ReadString

TRAP example: hardware control

- Task #32 provides hardware simulation through Sim68k
- Hardware window: **8 digit 7-segment display**, a bank of **8 light emitting diodes** (LED), a bank of **8 toggle switches**, a bank of **8 push button switches**.
- Each of these hardware items may be mapped to any valid 68000 address by entering the desired address in the corresponding Address: field. If the address entered causes a conflict with another device the color of the address text will change to red.



Task	
30	Clear the Cycle Counter
31	Return the Cycle Counter in D1.L Zero is returned if the cycle count exceeds 32 bits.
32	Hardware/Simulator D1.B = 00, Display hardware window D1.B = 01, Return address of 7-segment display in D1.L D1.B = 02, Return address of LEDs in D1.L D1.B = 03, Return address of toggle switches in D1.L D1.B = 04, Return Sim68K version number in D1.L Version 3.9.10 is returned as 0003090A D1.B = 05, Enable exception processing. Exceptions will be directed to the appropriate 68000 exception vector. This has the same effect as checking Enable Exceptions in the Options menu. D1.B = 06, Set Auto IRQ D2.B = 00, disable all Auto IRQs or Bit 7 = 0, disable individual IRQ Bit 7 = 1, enable individual IRQ Bits 6-0, IRQ number 1 through 7 D3.L, Auto Interval in milliseconds, 1000 = 1 second D1.B = 07, Return address of push button switches in D1.L

TRAP example: 7-seg display and controls

- The 7-segment display has each digit mapped to a successive word address beginning with the left-most digit.
- The toggle switches write a 1 to the corresponding bit in memory when the switch is on "Up" and write a 0 when the switch is off "Down".
- The push button switches are normally high. They write a 0 to the corresponding bit in memory while the switch is pressed and write a 1 when the switch is released.

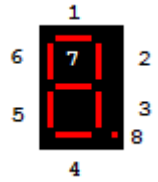
Position/Address

DIGIT	ADDRESS
1	E00000
2	E00002
3	E00004
4	E00006
5	E00008
6	E0000A
7	E0000C
8	E0000E

```
move.b  #32,d0   trap task to get address of hardware
move.b  #0,d1
trap    #15          display hardware window
move.b  #1,d1
trap    #15      get address of 7-segment display
move.l  d1,a4    a4  = address of 7-segment display
...
adda.l  #14,a4   point to the right-most display
move.b  #%01100110,(a4)      show 4 on last display
```

Display bitmask

87654321
00000000

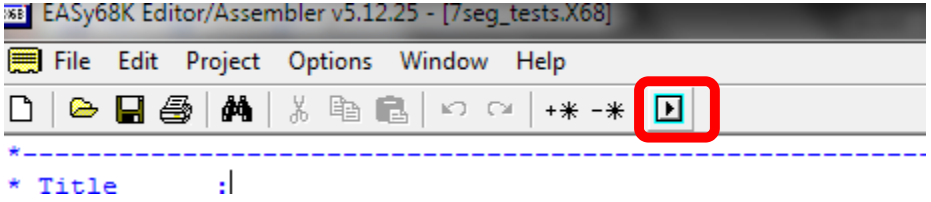


Esempio

IO_PROGR_CIAO

Easy68k: assembling and simulation

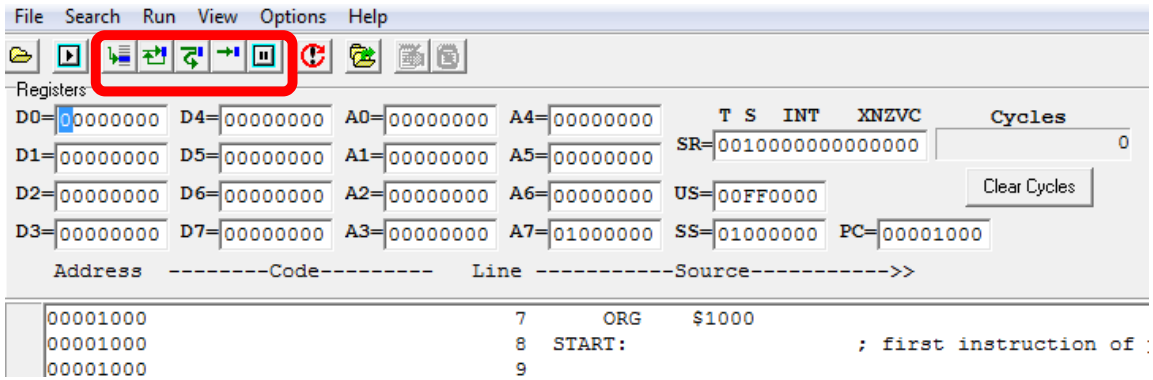
- In Edit68k press Assembly Source button



- If no errors are detected, the Sim68K simulator starts

Simulation run options:

- Run
- Run to Cursor
- Auto Trace
- Step Over
- Trace Into
- Pause
- Rewind



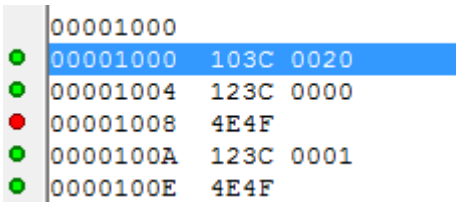
Simulation Run

Run

- To run the program, select Run from the Run Menu, press F9 or click the Run button on the toolbar. Sim68K begins executing the 68000 program at the current Program Counter location. Program execution will continue until one of the following occurs:
  - The program reaches a STOP instruction.
  - The program reaches a user placed Break-Point
  - The user Pauses the program.
  - The user Resets the simulator.
  - An exception occurs.

Step Over

- Executes the current instruction and positions the Program Counter at the instruction in the next line. If the current instruction is a JSR or BSR the subroutine is completely executed and the Program Counter is placed at the instruction following the JSR or BSR.



BreakPoint

- A breakpoint is used to halt a program. Put the cursor on the green circle and click to activate the BP (red circle).

When the program reaches the breakpoint it will halt prior to running the instruction at the breakpoint. The contents of the 68000 registers are displayed and may be modified. The program may be resumed using the Run button or using Trace and Step.

Programming with Advanced risc machine (ARM)

Registers and memory organization

- Byte addressable
- Half and full words (16 or 32 bits) can be organized as both big-endian and little-endian
- 7 operating modes (usr, fiq, irq, svc, abt, sys, und)
- 37 registers
  - 30 general purpose
  - 1 program counter (pc)
  - 1 current program status register (cpsr)
  - 5 saved program status registers (spsr)
- **Reduced Instruction Set Computers (RISC) have one-word instructions and require arithmetic operands to be in registers**
- A load/store architecture is used, meaning:
  - only Load and Store instructions are used to access memory operands
  - operands for arithmetic/logic instructions must be in registers, or one of them may be given explicitly in instruction word
  - data transfers are required before arithmetic

Operating modes

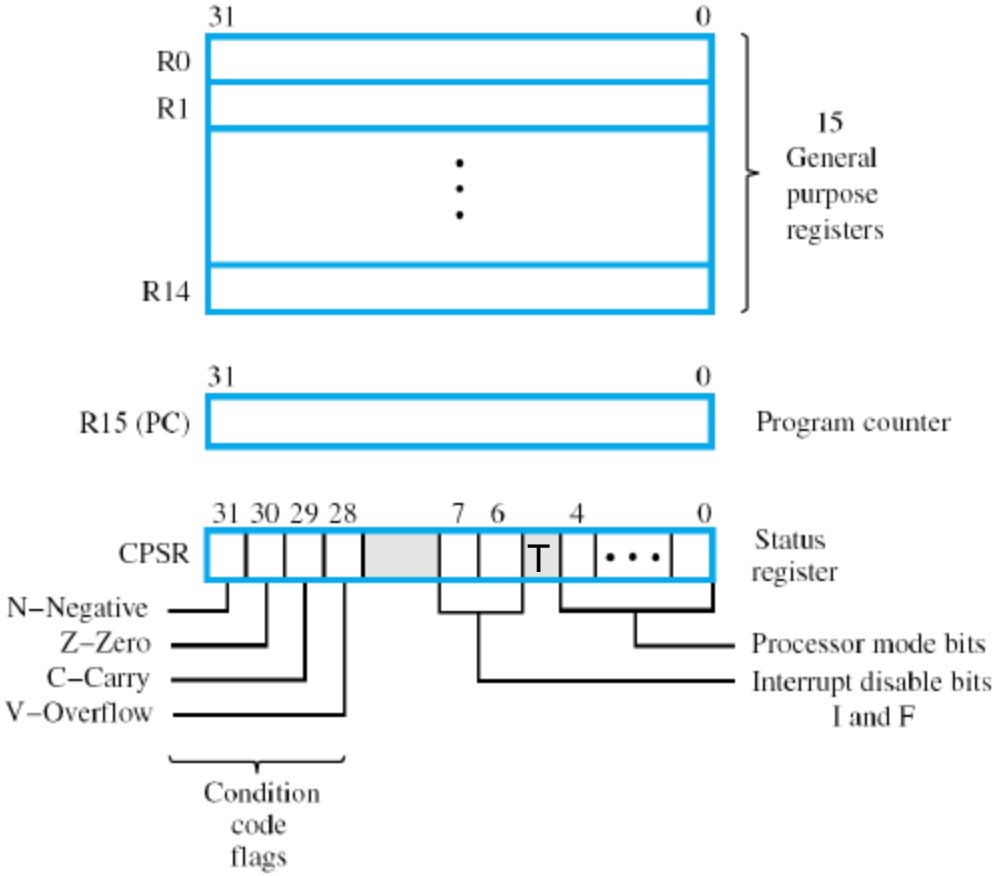
The ARM has six operating modes:

- *User* (unprivileged mode under which most tasks run)
- *FIQ* (entered when a high priority (fast) interrupt is raised)
- *IRQ* (entered when a low priority (normal) interrupt is raised)
- *Supervisor* (entered on reset and when a Software Interrupt instruction is executed)
- *Abort* (used to handle memory access violations)
- *Undef* (used to handle undefined instructions)

ARM Architecture Version 4 adds a seventh mode:

- *System* (privileged mode using the same registers as user mode)

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.

Register access

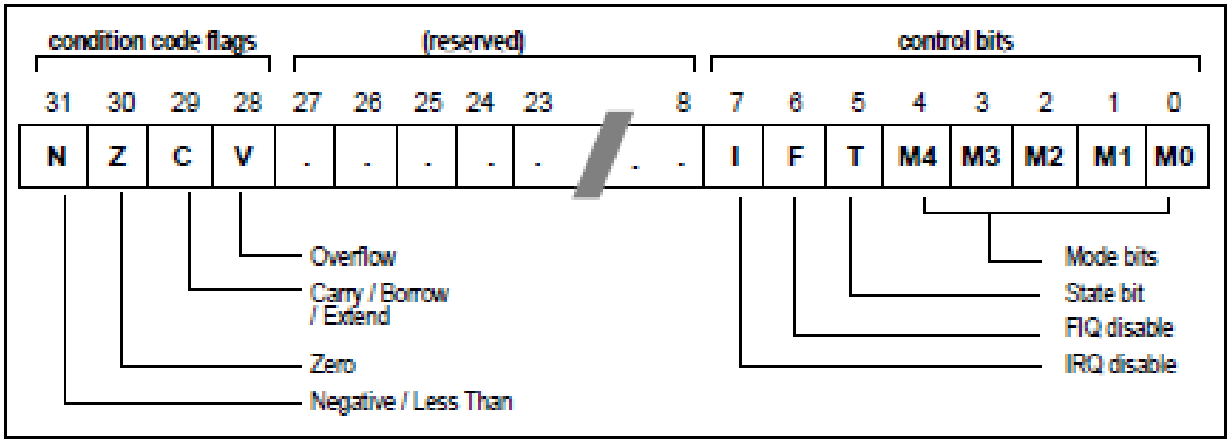
Each mode can access

- a particular set of r0-r12 registers
- a particular r13 (the stack pointer) and r14 (link register)
- r15 (the program counter)
- cpsr (the current program status register)

Privileged modes can access also to

- Spsr registers

Program Status Register

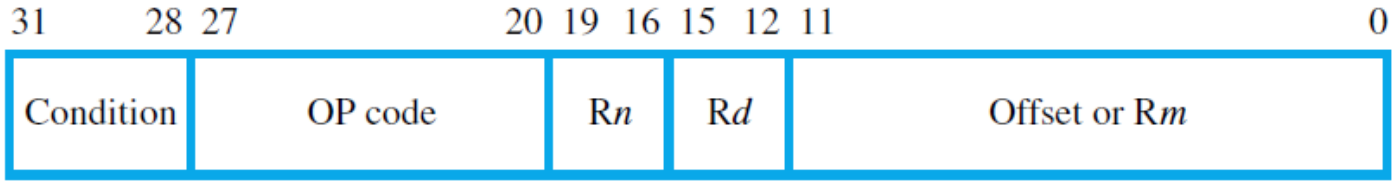


Interrupt Disable bits.
I = 1, disables the IRQ.
F = 1, disables the FIQ.

T Bit (Architecture v4T only)
T = 0, Processor in ARM state
T = 1, Processor in Thumb state

Example: Pre-indexed Load

- Indexed addressing: effective address of a memory operand is the sum of the contents of a base register Rn and a signed offset
- Offset:12-bit immediate value or Rm value



LDR Rd, [Rn, #offset]
performs Rd ← [[Rn] + offset]

LDR Rd, [Rn, Rm]
performs Rd ← [[Rn] + [Rm]]

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

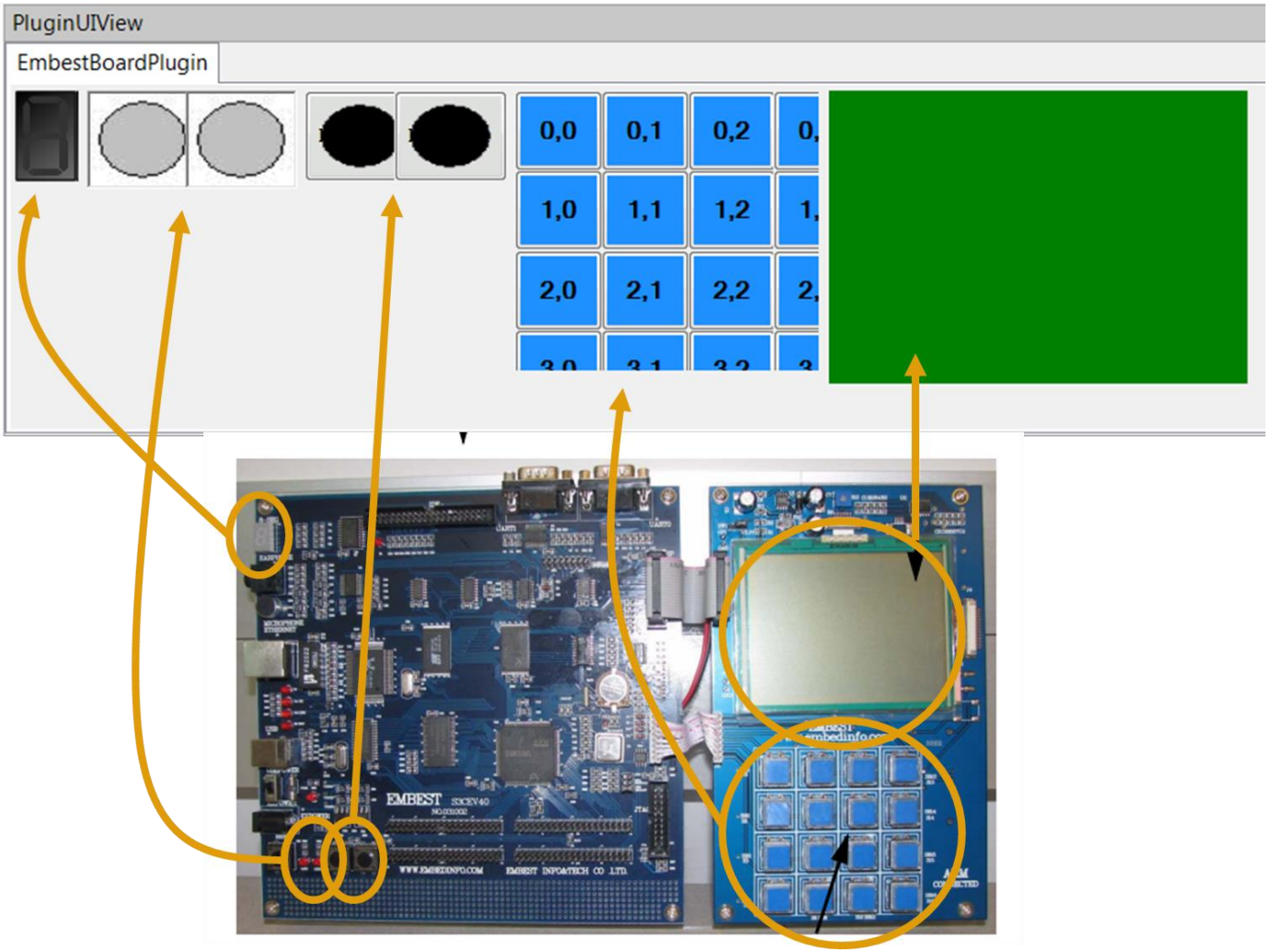
ARMSim# simulator

- Simulates ARM7TDMI processor:
  - T=thumb instruction set, D=debug unit, M=MMU, I=trace circuit is inside the core (Embedded Trace Macrocel)
  - This is basic core and all core have TDMI
  - e.g. Cortex, M excepted, have ARM7 core
- Thumb instruction set: a compact 16-bit encoding for a subset of the ARM instruction set.

ARMSim# has been developed by members of the Department of Computer Science at the University of Victoria, in Victoria, British Columbia, Canada. It is distributed for free for academic use.

- http://armsim.cs.uvic.ca/
- Does not include code editor
- Provides plugin extensions e.g. for hardware simulation
- ARMSim# user guide (pdf) available

EmbestBoardPlugin

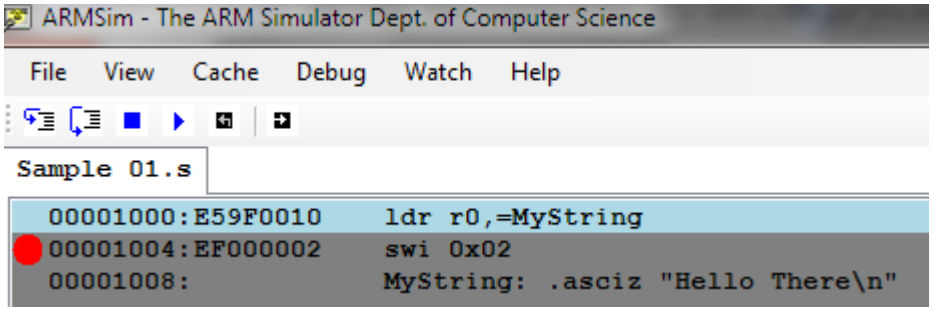


ARMSim#

- Prepare the code:
  - Use a text editor
  - Write your ARM7tdmi code
  - Save as “.s” file
- Execute ARMSim# (for Microsoft Windows OSs)
  - Open your “.s” file
  - Wait for program assembling
  - Check your program execution

Simulation Run

- The **Step Into** button causes the simulator to execute the highlighted instruction and move to the next instruction in the program. If the highlighted instruction is a subroutine call (BL or BX instruction) then the next highlighted instruction will be the first instruction of the subroutine.
- The **Step Over** button causes the simulator to execute the highlighted instruction and move to the next instruction in the current subroutine. If the highlighted instruction is a subroutine call (BL or BX instruction) then the program is run until the subroutine returns. Thus, unless a breakpoint is encountered, the next highlighted instruction will be at the return point from the subroutine call.
- **Breakpoint:** double click on instruction line to set a breakpoint (red circle) (Code view)



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

ARM Directives

- **AREA:** The AREA directive instructs the assembler to assemble a new code or data area. Areas are independent, named, indivisible chunks of code or data that are manipulated by the linker. (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0041c/CACGFDDB.html)
  - Example: The following example defines a read-only code area named Example.

    ```
    AREA Example,CODE,READONLY          ; An example code area.
                                        ; code
    ```
- **ENTRY:** The ENTRY directive declares its offset in its containing AOF area to be the unique entry point to any program containing the area.
  - You must specify one and only one ENTRY directive for a program. If ENTRY does not exist, or if more than one ENTRY exists, a error message is generated at link time.
- **END:** The END directive informs the assembler that it has reached the end of a source file.
  - Every assembly language source file must end with END on a line by itself.

ARM Directives

- **DCB**: The DCB directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. = is a synonym for DCB.
- **DCD:** The DCD directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory. & is a synonym for DCD.
- **DCW:** The DCW directive allocates one or more halfwords of memory, aligned on 2-byte boundaries, and defines the initial runtime contents of the memory.
- **EQU:** The EQU directive gives a symbolic name to a numeric constant. * is a synonym for EQU.
  - Example: num EQU 2 ; num is equivalent to 2
- **ALIGN:** By default, the ALIGN directive aligns the current location within the code to a word (4-byte) boundary. The current location is aligned to the next 2^n-byte boundary.
  - Use ALIGN to ensure that your code is correctly aligned. As a general rule it is safer to use ALIGN frequently through your code.
  - Use ALIGN when data definition directives appear in code areas. When data definition directives (DCB, DCW, DCWU, DCDU and %) are used in code areas, the program counter does not necessarily point to a word boundary. When the assembler encounters the next instruction mnemonic it inserts up to 3 bytes, if required, to ensure that the instruction is: word aligned in ARM state; halfword aligned in Thumb state.

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        DCB     1                       ; pc misaligned
        ALIGN                           ; ensures that label1 addresses
label1                                  ; the following instruction.
        MOV r5,#0x5
```

GNU assembly (GAS) directives

GNU assembly (GAS) is the assembler used by the GNU Project. It uses C style directives, which are architecture independent. It represents a “generic” assembler

Directives are keywords beginning with a period.

For example:

- .asciz:

The .asciz directive accepts string literals as arguments. String literal are a sequence characters in double quotes. The string literals are assembled into consecutive memory locations. The assembler automatically inserts a null character (\0 character) after each string.

  - Syntax: .asciz StringValue

- .equ:

Set the value of symbol equal to expression.

  - Syntax: .equ SEG_A,0x80

- ARMSim# accepts some of these directives
- Similar to pure ARM directives seen before

Directive comparison

Tabella A2.4 Direttive di assemblatore GAS, NIOS II, ColdFire, ARM e MASM

GAS	NIOS II	ColdFire	ARM	MASM
.org	.org	.org		ORG
.equ|=	.equ	.equ|=	EQU|=	EQU|=
.space|.skip	.skip	.space|ds.*t*	SPACE	D*t* *n* DUP(*v*)
.byte	.byte	.byte|dc.b	DCB	DB
.short|.hword|.word	.hword	.short|dc.w	DCW*a*	DW
.long|.int|.word	.word	.long|dc.l	DCD*a*	DD
.quad			DCQ*a*	DQ
.ascii	.ascii	.ascii	DCB	
.asciz|.string	.asciz	.asciz	DCB "*s*",0	
.data	.data	.data	AREA *s* DATA	.DATA
.text	.text	.text	AREA *s* CODE	.CODE
		entry	ENTRY	
		.textequ		TEXTEQU
.req			RN	
.title			TTL	TITLE
.end	.end		END	END *e*

Legenda:
t suffisso del codice mnemonico: tipo (dimensione) di ciascun elemento ColdFire: $t \in$ {b,w,l}; MASM: $t \in$ {B,W,D,Q};
n numero di elementi;
v valore iniziale di ogni elemento, "?" se dati non inizializzati;
a suffisso del codice mnemonico: nessun allineamento se *a* = U, altrimenti *a*, assente e allineamento a indirizzo pari se DCW, multiplo di 4 se DCD o DCQ;
s stringa di caratteri ASCII (nella direttiva AREA: nome del segmento);
e etichetta (opzionale): indirizzo di inizio dell'esecuzione del programma.

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;
```

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++;
}
```

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:

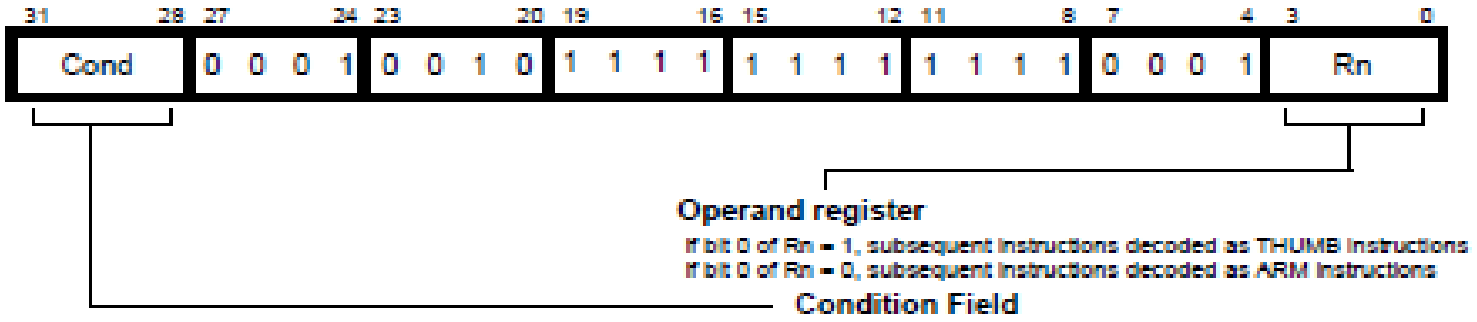
```
InFileName:      .asciz   "Infile1.txt"
InFileError:     .asciz   "Unable to open input file\n"
                 .align
InFileHandle:    .word    0

                 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
```