

Calcolatori Elettronici

Esercitazioni Assembler

M. Sonza Reorda – M. Monetti

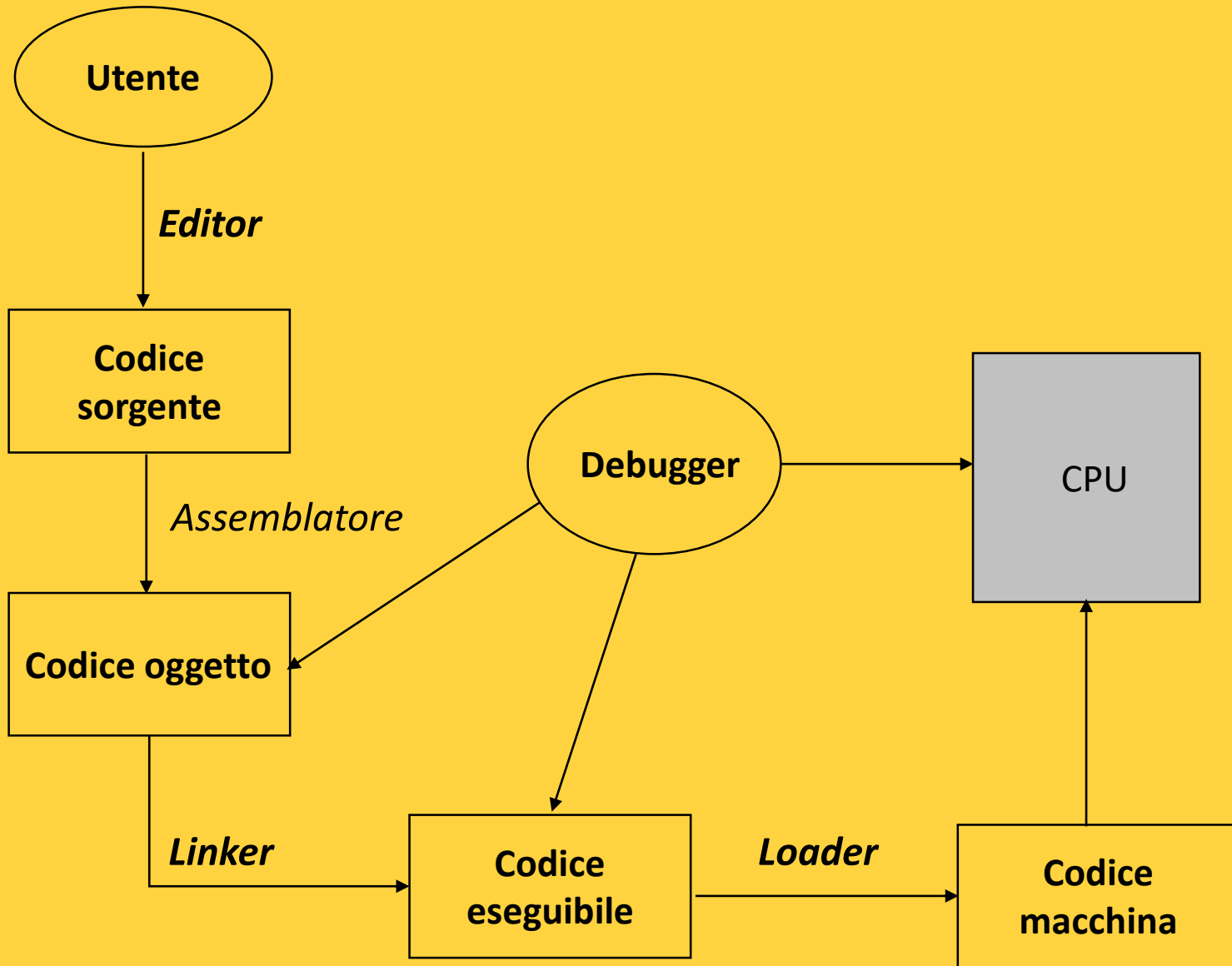
AA 2024/2025

massimo.monetti@polito.it

Politecnico di Torino

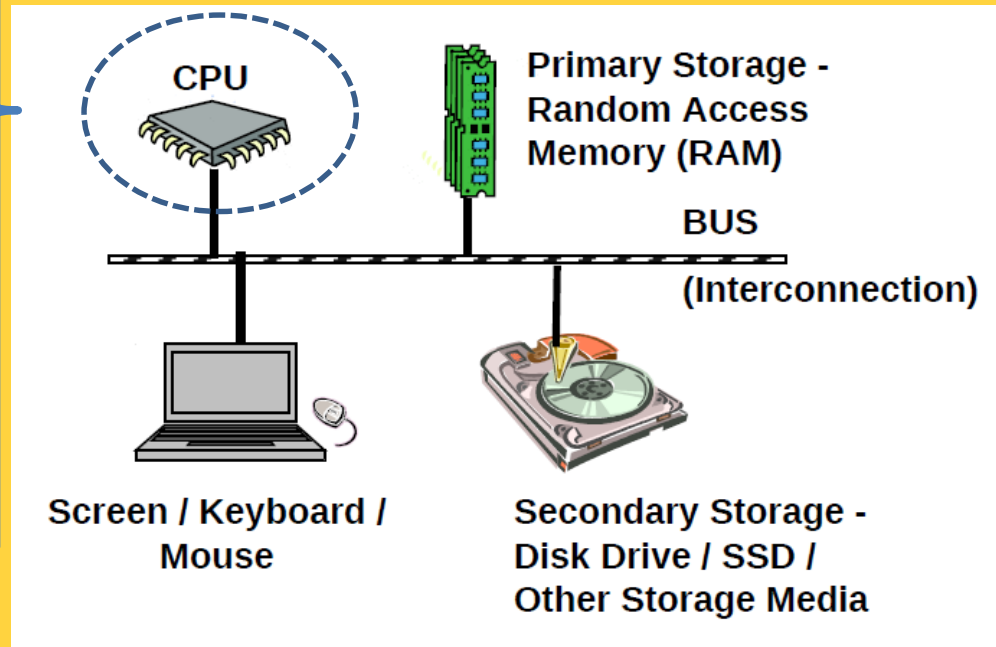
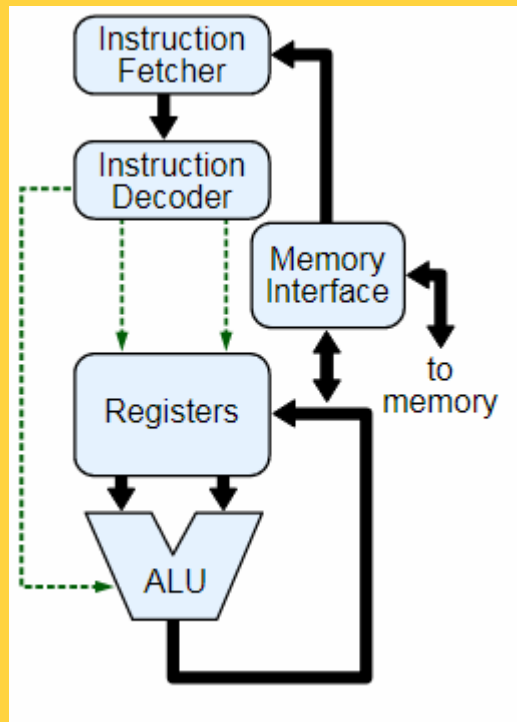
Dipartimento di Automatica e Informatica

Ciclo di vita di un programma



Il calcolatore

Schema dal punto di vista del programmatore in linguaggio Assembly



Architettura RISC-V - Registri

Register Name= x + valore alfanumerico

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

Codice di esempio

- Il codice può essere introdotto con un qualsiasi editor di testo, e salvato in un file con estensione **.a**, **.s** oppure **.asm**
 - Editor consigliato: notepad++

```
.data
# dichiarazione dati
op1: .byte 3
op2: .byte 2
res: .zero 1
# allocazione spazio in memoria per risultato
.text
main:
    lb t1,op1          # caricamento dati
    la s11,op2
    lb t2,0(s11)
    add t1,t1,t2        # esecuzione somma
    la s11,res
    sb t1,0(s11)        # salvataggio del risultato
exit:
    li a7,10
    ecall
```

data segment

- Di seguito sono riportate le dichiarazioni delle variabili

text segment

- Di seguito sono riportate le istruzioni

main procedure

- Punto di partenza del programma.

- Fine del programma

Tipi dato e dimensioni

L'architettura RISC-V utilizza le seguenti dimensioni di data/memory :

- Byte (**8 bit**)
- Halfword (semplicemente *half*) (**16 bit**)
- Word (**32 bit**)

Character ha tipicamente dimensioni di 1 byte e una stringa è una serie di byte in sequenza

.data

w1: .word 14

b1: .byte 120

h1: .half 22

string: .string "\nThis is another string"

space: .zero 32

Istruzioni e pseudo-istruzioni

Una istruzione nativa (bare-instruction) è un'istruzione che viene eseguita in modo nativo dalla CPU.

<pre>.data # dichiarazione dati op1: .byte 3 op2: .byte 2 res: .zero 1 # allocazione spazio in memoria per risultato .text main: lb t1,op1 # caricamento dati la s11,op2 lb t2,0(s11) add t1,t1,t2 # esecuzione somma la s11,res sb t1,0(s11) # salvataggio del risultato in memoria exit: li a7,10 ecall</pre>	<pre>00000000 <main>: 0: 10000317 auipc x6 0x10000 4: 00030303 lb x6 0 x6 8: 10000d97 auipc x27 0x10000 c: ff9d8d93 addi x27 x27 -7 10: 000d8383 lb x7 0 x27 14: 00730333 add x6 x6 x7 18: 10000d97 auipc x27 0x10000 1c: fead8d93 addi x27 x27 -22 20: 006d8023 sb x6 0 x27 00000024 <exit>: 24: 00a00893 addi x17 x0 10 28: 00000073 ecall</pre>
--	--

Una pseudo-istruzione è un'istruzione che l'assemblatore, o simulatore, riconosce, ma deve poi convertire in una o più istruzioni native.

<pre>.data # dichiarazione dati op1: .byte 3 op2: .byte 2 res: .zero 1 # allocazione spazio in memoria per risultato .text main: lb t1,op1 # caricamento dati la s11,op2 lb t2,0(s11) add t1,t1,t2 # esecuzione somma la s11,res sb t1,0(s11) # salvataggio del risultato in memoria exit: li a7,10 ecall</pre>	<pre>00000000 <main>: 0: 10000317 auipc x6 0x10000 4: 00030303 lb x6 0 x6 8: 10000d97 auipc x27 0x10000 c: ff9d8d93 addi x27 x27 -7 10: 000d8383 lb x7 0 x27 14: 00730333 add x6 x6 x7 18: 10000d97 auipc x27 0x10000 1c: fead8d93 addi x27 x27 -22 20: 006d8023 sb x6 0 x27 00000024 <exit>: 24: 00a00893 addi x17 x0 10 28: 00000073 ecall</pre>
--	--

Istruzioni Data Transfer

Load Upper Immediate - LUI (*Rdest, imm*)

Load Immediate - **LI** (*Rdest, imm*)

PSEUDO-ISTRUZIONE

Move - **MV** (*Rdest, Rsrc*)

PSEUDO-ISTRUZIONE

lui	Load Upper Imm	U	0110111			rd = imm << 12
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)

Pseudoinstruction	Base Instruction(s)	Meaning
li rd, immediate	addi rd, x0, imm	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register

Istruzioni Memory Access

Load Word/Byte - (LW,LB) (*Rdest*, *n(rs)*)

Store Word/Byte - (SW,SB) (*Rsrc*, *n(rs)*)

Load Address - LA (*Rdest*, *mem address*)

PSEUDO-ISTRUZIONE

lb	Load Byte	I	0000011	0x0		$rd = M[rs1+imm][0:7]$
lh	Load Half	I	0000011	0x1		$rd = M[rs1+imm][0:15]$
lw	Load Word	I	0000011	0x2		$rd = M[rs1+imm][0:31]$
lbu	Load Byte (U)	I	0000011	0x4		$rd = M[rs1+imm][0:7]$
lhu	Load Half (U)	I	0000011	0x5		$rd = M[rs1+imm][0:15]$
sb	Store Byte	S	0100011	0x0		$M[rs1+imm][0:7] = rs2[0:7]$
sh	Store Half	S	0100011	0x1		$M[rs1+imm][0:15] = rs2[0:15]$
sw	Store Word	S	0100011	0x2		$M[rs1+imm][0:31] = rs2[0:31]$

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address

Istruzioni di Branch

Branch - **BEQ-BNE-BLT-BLE** *Rs1, Rs2, label*

beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm

```
ciclo2: lb  t3, (t0)
        bgt t3, t2, salta      # salta se NON deve aggiornare MIN
        lb  t2, (t0)           # aggiorna MIN
salta:  addi t1, t1, 1
        addi t0, t0, 1
        li  s11, 15
        bne t1, s11, ciclo2
```

Istruzioni di Branch

Pseudoinstruction	Base Instruction(s)	Meaning
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if \neq zero
blez rs, offset	bge x0, rs, offset	Branch if \leq zero
bgez rs, offset	bge rs, x0, offset	Branch if \geq zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if \leq
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if \leq , unsigned

```
ciclo2: lb  t3, (t0)
        bgt t3, t2, salta      # salta se NON deve aggiornare MIN
        lb  t2, (t0)          # aggiorna MIN
salta:  addi t1, t1, 1
        addi t0, t0, 1
        li  s11, 15
        bne t1, s11, ciclo2
```

Istruzioni Logical

**SLLI (Shift Left Logical Immediate),
SRLI (Shift Right Logical Immediate),
SLL, SRL
SRAI (Shift Right Arithmetic Immediate)**

xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]

Istruzioni Aritmetiche

Somma - ADD *Rdest, Rs2, Rs1*

Sottrazione - SUB *Rdest, Rs2, Rs1*

add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

Istruzioni Aritmetiche

Nel caso della moltiplicazione su 64 bit, la parte alta si gestisce con l'istruzione **mulh rd, rs1, rs2**

```
        li    t1, 0
ciclo1: add   t1, t1, 1
        add   t0, t0, 1
        sub   t2, t1, t0
        mul   s3, t1, t0
        mulh  s4, t1, t0
        li    s11, 15
        bne   t1, s11, ciclo1      # itera 15 volte
```

Metodi di Indirizzamento

Register only. Gli operandi nei Registri

```
add s0, t2, t3
```

Immediate. Immediato in base 10 o base Hex

```
addi s4, t5, -73
```

```
addi s4, t5, 0x14
```

Metodi di indirizzamento

Base Addressing. L'operando è un Base Address + Immediate

```
op1: .word 5
```

```
la    t1,op1
```

```
lw    s4,0(t1)
```

oppure

```
lw    s4,op1
```

```
op2: .word 5
```

```
la    t1,op2
```

```
sw    s4,0(t1)
```


Metodi di Indirizzamento

PC-Relative Addressing. Branches e JAL

0x354	L1:	addi s1, s1, 1
0x358		sub t0, t1, s7
...		...
0xEB0		bne s8, s9, L1

Pseudo-istruzione LI

LI (Load Immediate) ci consente di analizzare una particolarità di RISC-V

Se **Immediate** è un numero che richiede un numero di bit superiore a 12 l'istruzione LI (una pseudo- istruzione) si spezza in due istruzioni

li t0, 0xFEDC8EAB	0:	fedc92b7	lui x5 0xfedc9
	4:	eab28293	addi x5 x5 -341

Se **Immediate** è un numero che richiede un numero di bit minore o eguale a 12 l'istruzione LI (una pseudo- istruzione) diventa

li a7,10	2c:	00a00893	addi x17 x0 10
----------	-----	----------	----------------

Pseudo-istruzione LI

Esempi

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

- Use load upper immediate (lui) and addi
- lui: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9    # s0 = 0xFEDC9000
addi s0, s0, -341   # s0 = 0xFEDC9000 + 0xFFFFFEAB
                        #      = 0xFEDC8EAB
```

Pseudo-istruzione LI

Esempi

C Code

```
int a = 0xFEDC8EAB;
```

Note: -341 = 0xEAB

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC9      # s0 = 0xFEDC9000
addi s0, s0, -341     # s0 = 0xFEDC9000 + 0xFFFFFEAB
                        #      = 0xFEDC8EAB
```

If **bit 11** of the constant is **1**, increment upper 20 bits by **1** in `lui`

Ripes v2.2.6

File Edit View Help

100 ms

Source code Input type: ☒ Assembly ☐ Executable code View mode: ☐ Binary ☒ Disassembled

gpr

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7fffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000
x18	s2	0x00000000
x19	s3	0x00000000
x20	s4	0x00000000
x21	s5	0x00000000
x22	s6	0x00000000
x23	s7	0x00000000

Console Memory

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x1000000c	X	X	X	X	X
0x10000008	X	X	X	X	X
0x10000004	X	X	X	X	X
0x10000000	X	X	X	X	X
0x0ffffffc	X	X	X	X	X
0x0ffffff8	X	X	X	X	X

Display type: Hex Go to register: Go to section:

Processor: Single-cycle processor ISA: RV32

Ripes version: v2.2.6-57-g8c3783f

<https://ripes.me>

Ripes v2.2.6 - desktop





Release build (v.2.2.6) Latest

Ripes v2.2.6

Bug fixes and minor stuff

- Fixed crash when changing processor layout due to incorrect processor label IDs.
- Changed the terminology for caches, substituting "Word" for "Block". Most textbooks use the term "cache block" interchangeably with "cache line", so using "Block" for words stored in a cache line was confusing.

▼ Assets 5

 Ripes-v2.2.6-linux-x86_64.ApplImage	29.6 MB	Jan 7, 2023
 Ripes-v2.2.6-mac-x86_64.zip	30.4 MB	Jan 7, 2023
 Ripes-v2.2.6-win-x86_64.zip	15 MB	Jan 7, 2023
 Source code (zip)		Jan 2, 2023
 Source code (tar.gz)		Jan 2, 2023

 2  11  10  18  1  2 34 people reacted

<https://github.com/mortbopet/ripes/releases>

Setup

Impostare Single-cycle processor

The screenshot shows the 101010 Editor interface. The top menu bar includes File, Edit, View, and Help. Below the menu bar is a toolbar with icons for a processor, cache, memory, and I/O, along with a red arrow pointing to the 'Single-cycle processor' option in the 'Select Processor' dialog. The main window is titled 'Source code' and contains a text area for input. The 'Input type' is set to 'Assembly' and the 'View mode' is set to 'Disasm'. The 'Select Processor' dialog is open, showing a list of processors under the 'RISC-V' and '32-bit' categories. The 'Single-cycle processor' is selected. The dialog also shows fields for Name, ISA, ISA Extensions, Layout, and Description. The 'Register initialization' section shows 'x2 (sp)' set to '0x7fffffff' and 'x3 (gp)' set to '0x10000000'. The 'OK' and 'Cancel' buttons are at the bottom right.

File Edit View Help

101010 Editor

Processor

Cache

Memory

I/O

Source code

Input type: ☒ Assembly ☐ Executable code

View mode: ☐ Binary ☒ Disasm

100 ms

Select Processor

- ▼ RISC-V
 - ▼ 32-bit
 - Single-cycle processor**
 - 5-stage processor w/o forward...
 - 5-stage processor w/o hazard ...
 - 5-Stage processor w/o forward...
 - 5-stage processor
 - 6-stage dual-issue processor
 - 64-bit
 - MIPS

Name: Single-cycle processor

ISA: RV32I

ISA Exts. ☒ M ☐ C

Layout: Standard

Description: A single cycle processor

Register initialization

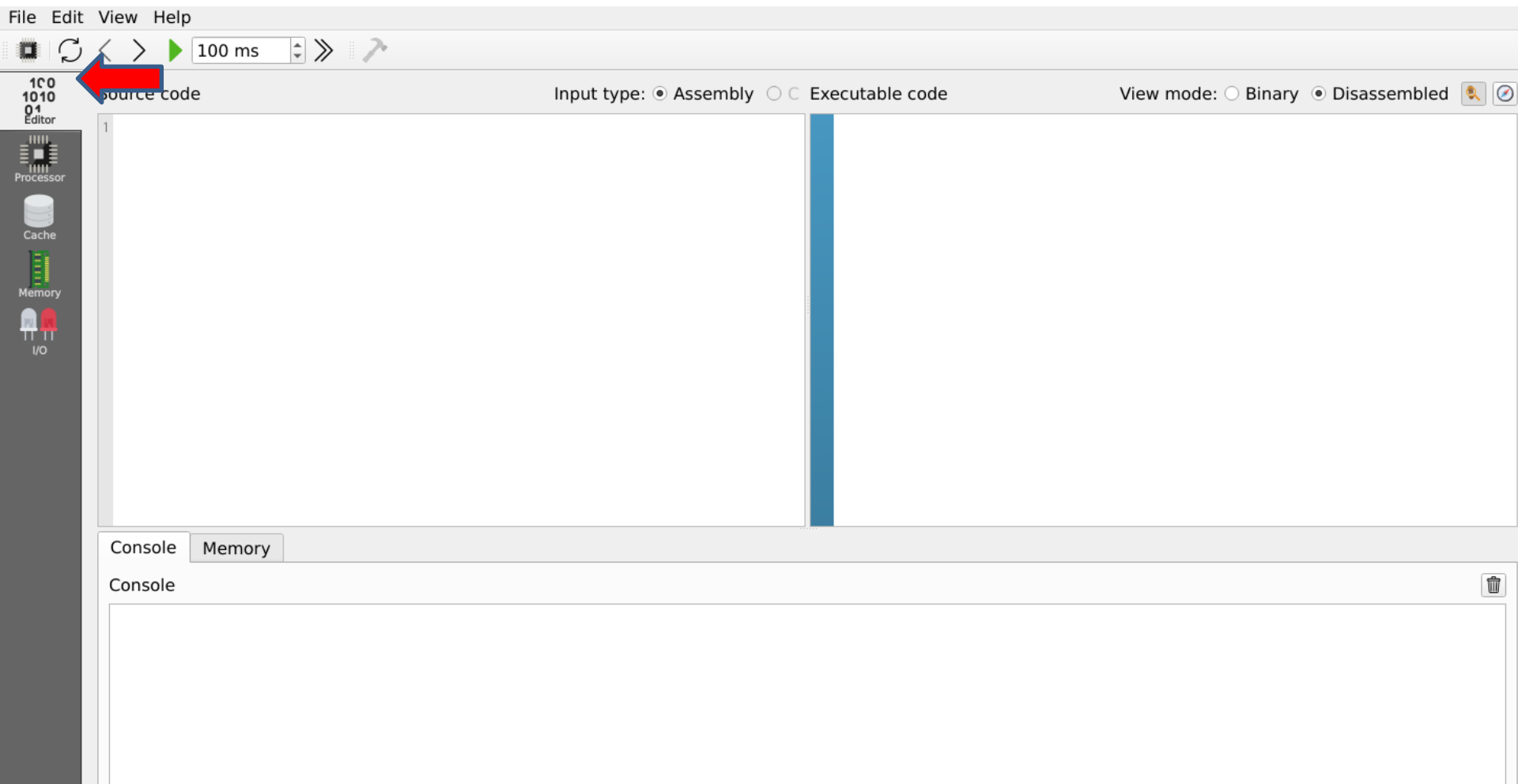
x2 (sp)	0x7fffffff0	✗
x3 (gp)	0x10000000	✗

+

OK Cancel

Source code

Click sull'icona a sx e inserire (con cut&paste) codice nella finestra di sx **Source Code**



Source code

Il source code viene analizzato sintatticamente, se tutto ok appare il codice disassemblato a dx

The screenshot shows a software development environment with a menu bar (File, Edit, View, Help) and a toolbar with icons for running, stepping through, and searching. On the left is a sidebar with icons for the Editor, Processor, Cache, Memory, and I/O. The main window is split into two panes. The left pane, titled 'Source code', shows assembly code with line numbers 1 through 18. The right pane, titled 'Disassembled', shows the corresponding machine code with addresses and instructions. The 'Input type' is set to 'Assembly' and the 'View mode' is set to 'Disassembled'.

Source code

```
1 .data
2 # dichiarazione dati
3 op1: .byte 3
4 op2: .byte 2
5 res: .zero 1
6 # allocazione spazio in memoria per risultato
7 .text
8 main:
9     lb t1,op1          # caricamento dati
10    la s11,op2
11    lb t2,0(s11)
12    add t1,t1,t2        # esecuzione somma
13    la s11,res
14    sb t1,0(s11)        # salvataggio del risultato in memoria
15
16 exit:
17     li a7,10
18     ecall
```

Disassembled

```
00000000 <main>:
0: 10000317 auipc x6,0x10000
4: 00030303 lb x6,0,x6
8: 10000d97 auipc x27,0x10000
c: ff9d8d93 addi x27,x27,-7
10: 000d8383 lb x7,0,x27
14: 00730333 add x6,x6,x7
18: 10000d97 auipc x27,0x10000
1c: fead8d93 addi x27,x27,-22
20: 006d8023 sb x6,0,x27

00000024 <exit>:
24: 00a00893 addi x17,x0,10
28: 00000073 ecall
```

Source code

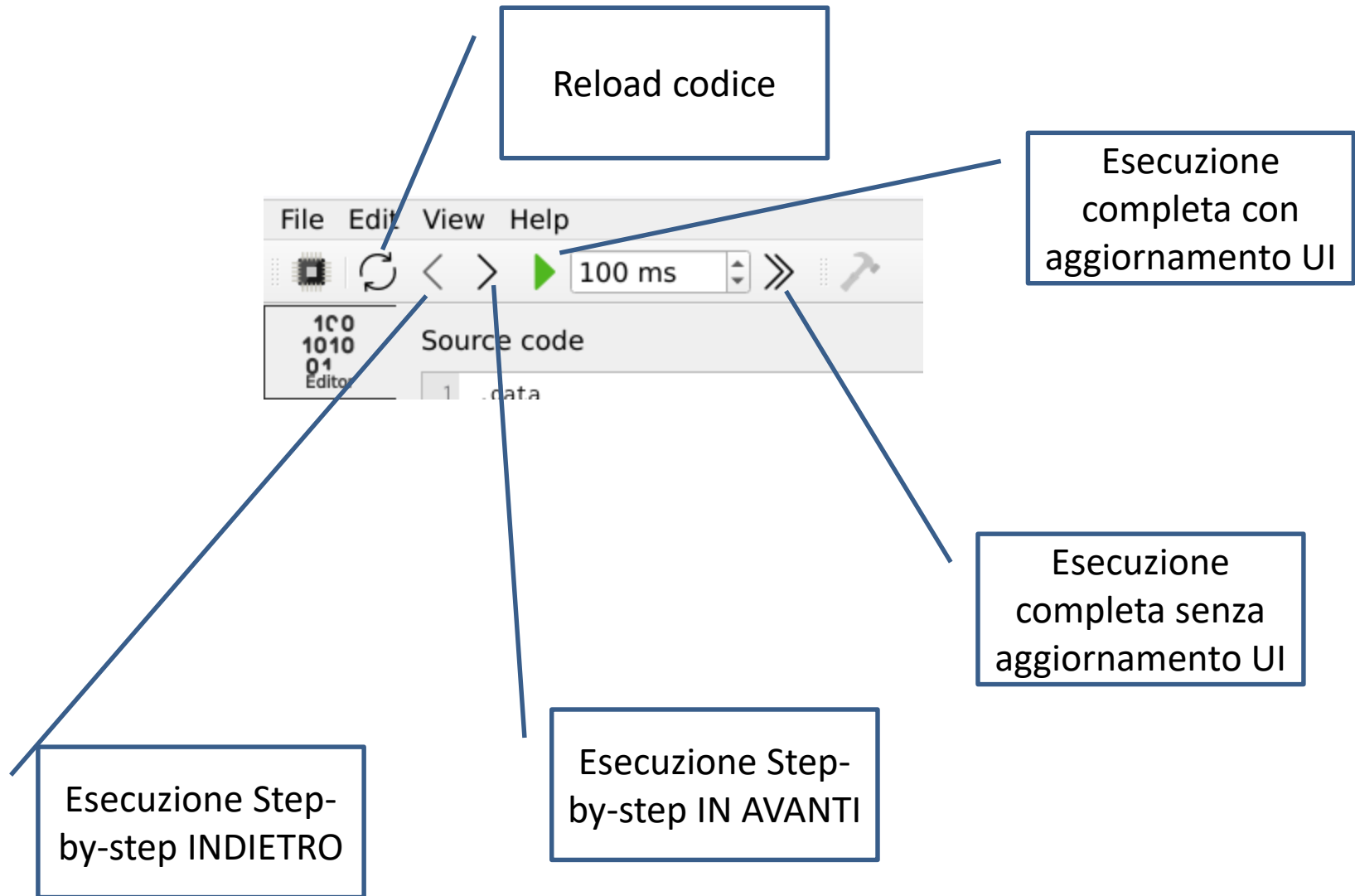
Se nel source code viene individuato un errore sintattico NON appare il codice disassemblato e la riga in errore viene sottolineata

The screenshot displays a software development environment with a menu bar (File, Edit, View, Help) and a toolbar containing icons for running, stepping through, and other debugging actions, along with a 100 ms timer. On the left, a sidebar shows system components: 1001010 Editor, Processor, Cache, Memory, and I/O. The main window is titled 'Source code' and features a tab for 'Input type' (Assembly selected) and a 'View mode' (Disassembled selected). The source code is as follows:

```
1 .data
2 # dichiarazione dati
3 op1: .byte 3
4 op2: .byte 2
5 res: .zero 1
6 # allocazione spazio in memoria per risultato
7 .text
8 main:
9     lb t1,op1      # caricamento dati
10    la s11,op2
11    lb t2,0(s11)
12    add t1,t1,t2    # esecuzione somma
13
14    sb t1,res       # salvataggio del risultato in memoria
15
16 exit:
17     li a7,10
18     ecall
```

A red arrow points to the line `sb t1,res` on line 14, which is underlined with a red dashed line, indicating a syntax error. The right pane, which would normally show disassembled code, is currently empty. At the bottom, there are tabs for 'Console' and 'Memory', with the 'Console' tab active and showing an empty output area.

Operatività



Debug

- L'esecuzione passo-passo è fondamentale per il *debug*
 - Osservare il valore di memoria e registri al termine di ogni istruzione
- È possibile inserire un **breakpoint** facendo click sulla corrispondente riga del codice disassemblato

Source code

Input type: ☒ Assembly ☐ Executable code

View mode: ☐ Bi

```
1 .data
2 # dichiarazione dati
3 op1: .byte 3
4 op2: .byte 2
5 res: .zero 1
6 # allocazione spazio in memoria per risultato
7 .text
8 main:
9     lb t1,op1      # caricamento dati
10    la s11,op2
11    lb t2,0(s11)
12    add t1,t1,t2    # esecuzione somma
13    la s11,res
14    sb t1,0(s11)    # salvataggio del risultato in memoria
15
16 exit:
17     li a7,10
18     ecall
```

```
00000000 <main>:
0:      10000317      auipc x6 0x10000
4:      00030303      lb x6 0 x6
8:      10000d97      auipc x27 0x10000
c:      ff9d8d93      addi x27 x27 -7
10:     000d8383      lb x7 0 x27
14:     00730333      add x6 x6 x7
18:     10000d97      auipc x27 0x10000
1c:     fead8d93      addi x27 x27 -22
20:     006d8023      sb x6 0 x27

00000024 <exit>:
24:     00a00893      addi x17 x0 10
28:     00000073      ecall
```

Debug

Colonna DX valore dei REGISTRI

File Edit View Help

100 ms

Source code

```
1 .data
2 # dichiarazione dati
3 op1: .byte 3
4 op2: .byte 2
5 res: .zero 1
6 # allocazione spazio in memoria per risultato
7 .text
8 main:
9     lb t1,op1      # caricamento dati
10    la s11,op2
11    lb t2,0(s11)
12    add t1,t1,t2    # esecuzione somma
13    la s11,res
14    sb t1,0(s11)    # salvataggio del risultato in memoria
15
16 exit:
17     li a7,10
18     ecall
```

Input type: ☒ Assembly ☐ Executable code

View mode: ☐ Binary ☒ Disassembled

gpr

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7fffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000000
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000
x18	s2	0x00000000
x19	s3	0x00000000
x20	s4	0x00000000
x21	s5	0x00000000
x22	s6	0x00000000

Console

Memory

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x00000010	0x000d8383	0x83	0x83	0x0d	0x00
0x0000000c	0xff9d8d93	0x93	0x8d	0x9d	0xff
0x00000008	0x10000d97	0x97	0x0d	0x00	0x10
0x00000004	0x00030303	0x03	0x03	0x03	0x00
0x00000000	0x10000317	0x17	0x03	0x00	0x10
-	-	-	-	-	-
-	-	-	-	-	-
-	-	-	-	-	-

Display type: Hex

Go to register:

Go to section:

Processor: Single-cycle processore

Debug [cont.]

The screenshot displays the Ripes debugger interface with the following components:

- Source code window:** Shows assembly code for a program. Line 17 is highlighted in red, corresponding to the instruction `li a7,10`.
- Symbol navigator:** A pop-up window listing symbols and their addresses:

Address	Symbol
0x00000000	main
0x00000024	exit
0x10000000	op1
0x10000001	op2
0x10000002	res
- Disassembled code window:** Shows the instruction `addi x17 x0 10` at address 0x10000000, which is highlighted in red.
- Register window (gpr):** A table showing register names, aliases, and values:

Name	Alias	Value
x9	s1	0x00000000
x10	a0	0x00000000
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000
x18	a8	0x00000000
x19	a9	0x00000000
x20	a10	0x00000000
x21	s5	0x00000000
x22	s6	0x00000000
x23	s7	0x00000000
x24	s8	0x00000000
x25	s9	0x00000000
x26	s10	0x00000000
x27	s11	0x10000002
x28	t3	0x00000000
x29	t4	0x00000000
x30	t5	0x00000000
x31	t6	0x00000000
- Memory window:** A table showing memory addresses and their contents in hex:

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000010	X	X	X	X	X
0x1000000c	X	X	X	X	X
0x10000008	0x00000000	0x00	0x00	0x00	0x00
0x10000004	0x00000000	0x00	0x00	0x00	0x00
0x10000000	0x00050203	0x03	0x02	0x05	0x00
0x0ffffffc	X	X	X	X	X
0x0fffffb8	X	X	X	X	X
0x0fffffb4	X	X	X	X	X

A yellow callout box with the text "Apri SYMBOL NAVIGATOR" points to the Symbol navigator window. A blue oval highlights the memory address 0x10000000 in the Memory window, which corresponds to the register s11 value 0x10000002 in the Register window.

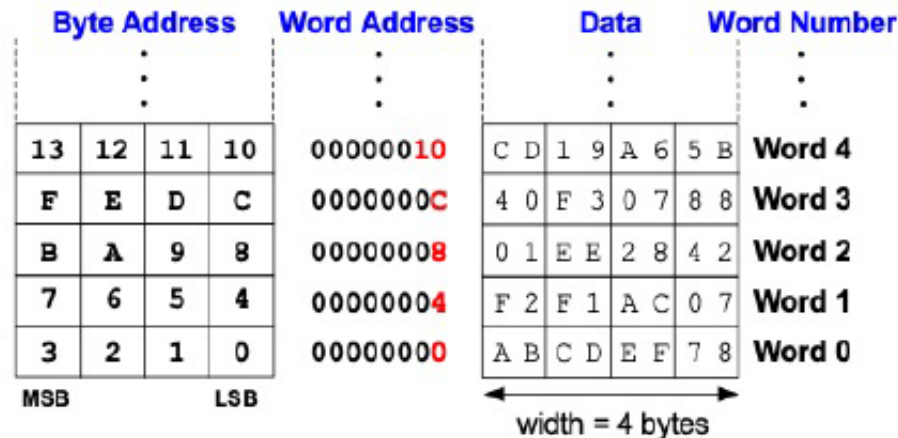
La memoria

LITTLE ENDIAN

La memorizzazione parte dal byte meno significativo per finire col più significativo, per indirizzo di memoria crescente.

RISC-V Byte-addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address **increments by 4**



La memoria

Istruzione di **Load** di una word di data all'indirizzo di memoria 8 nel registro **s3**
In **s3** troveremo **0x01EE2842**

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into s3.
- s3 holds the value 0x1EE2842 after load

RISC-V assembly code

```
lw s3, 8(zero) # read word at address 8 into s3
```

Byte Address				Word Address	Data	Word Number
⋮				⋮	⋮	⋮
13	12	11	10	00000010	C D 1 9 A 6 5 B	Word 4
F	E	D	C	0000000C	4 0 F 3 0 7 8 8	Word 3
B	A	9	8	00000008	0 1 E E 2 8 4 2	Word 2
7	6	5	4	00000004	F 2 F 1 A C 0 7	Word 1
3	2	1	0	00000000	A B C D E F 7 8	Word 0
MSB					width = 4 bytes	
					LSB	

La memoria

.data

dichiarazione dati

op1: .byte 3

op2: .byte 2

res: .zero 1

The screenshot shows a memory viewer interface with a menu bar (File, Edit, View, Help) and a toolbar with icons for file operations and a 100 ms timer. On the left, there is a sidebar with icons for CPU, Memory, and I/O. The main area is divided into two panels: 'Memory viewer' and 'Memory map'.

Memory viewer

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000000	X	X	X	X	X
0x10000020	X	X	X	X	X
0x10000024	X	X	X	X	X
0x10000028	X	X	X	X	X
0x1000001c	X	X	X	X	X
0x10000018	0x00000000	0x00	0x00	0x00	0x00
0x10000014	X	X	X	X	X
0x10000010	X	X	X	X	X
0x1000000c	X	X	X	X	X
0x10000008	0x00000000	0x00	0x00	0x00	0x00
0x10000004	0x00000000	0x00	0x00	X	X
0x10000000	0x00050203	0x03	0x02	0x05	0x00
0x0ffffffc	X	X	X	X	X
0x0ffffff8	X	X	X	X	X
0x0ffffff4	X	X	X	X	X
0x0ffffff0	X	X	X	X	X
0x0fffffec	X	X	X	X	X
0x0fffffe8	X	X	X	X	X
0x0fffffe4	X	X	X	X	X
0x0ffffe0	X	X	X	X	X
0x0ffffdc	X	X	X	X	X
0x0ffffd8	X	X	X	X	X

Display type: Hex - Go to register: - Go to section: -

Memory map

Name	Size	Range
.text	44	0x00000000 - 0x0000002b
.data	3	0x10000000 - 0x10000002
.bss	0	0x11000000 - 0x10ffffff

Processor: Single-cycle processor ISA: RV32IM

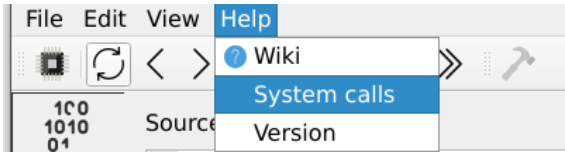
eCall

eCall con cui gestire I/O - Print

a7	a0	Name	Description
1	(integer to print)	print_int	Prints the value located in a0 as a signed integer
2	(float to print)	print_float	Prints the value located in a0 as a floating point number
4	(pointer to string)	print_string	Prints the null-terminated string located at address in a0
10	-	exit	Halts the simulator
11	(char to print)	print_char	Prints the value located in a0 as an ASCII character
34	(integer to print)	print_hex	Prints the value located in a0 as a hex number
35	(integer to print)	print_bin	Prints the value located in a0 as a binary number
36	(integer to print)	print_unsigned	Prints the value located in a0 as an unsigned integer
93	(status code)	exit	Halts the simulator and exits with status code in a0

eCall

eCall con cui gestire I/O - Read



Supported system calls

Func. (a7)	Name
11	PrintChar
17	GetCWD
30	Time_msec
31	Cycles
34	PrintIntHex
35	PrintIntBinary
36	PrintIntUnsigned
57	Close
62	LSeek
63	Read
64	Write
80	FStat
93	Exit2
214	brk
1024	Open

Read

Read from a file descriptor into a buffer

Arguments:

Arg. #	Reg.	Description
0	a0	the file descriptor
1	a1	address of the buffer
2	a2	maximum number of bytes to read

Returns:

Ret. #	Reg.	Description
0	a0	number of read bytes or -1 if an error occurred

Close

eCall

eCall con cui gestire I/O - Read

.data

buffer: .zero 255

.text

syscall 5 (read_int)

li a7, **63**

li a0,0

la a1, buffer

li a2,255

ecall

lw a0, 0(a1) # Nella prima word di BUFFER la lettura del char

andi a0,a0,255 # Mask 3 byte più significativi

li s11, 0x30 # trasformazione in INT del CHAR

sub a0,a0,s11 # trasformazione in INT del CHAR

mv t0, a0 # In T0 il risultato

CODICE di ESEMPIO

Template

Name and general description of program

Data declarations go in this section.

.data

program specific data declarations

Program code goes in this section.

.text

main:

>>>> your program code goes here.

Done, terminate program.

li a7, 10

ecall

Scrittura di un valore in una cella di memoria

```
.data
variabile:  .zero 4      # int variabile
  
.text
  
main:
    li  t0, 19591501      # variabile = 19591501 (012A F14D hex)
    la  s11, variabile
    sw  t0, 0(s11)
  
exit:  li  a7,10
    ecall
```

Input/Output da *console*

```
.data
buffer:      .zero 255
msg1:        .string "\nIntroduci il primo valore: "
msg2:        .string "\nIntroduci il secondo valore: "

.text

main:
# stampa la stringa
li  a7,4
# ecall 4 (print_str)
la  a0,msg1
# argomento: stringa
ecall
# syscall 5 (read_int)
li  a7, 63
li  a0,0
la  a1, buffer
li  a2,255
ecall
lw  a0, 0(a1)
andi a0,a0,255
li  s11, 0x30
sub  a0,a0,s11
mv   t0, a0
```

```
# primo operando
li  a7,4
la  a0,msg2
ecall

# syscall 5 (read_int)
li  a7, 63
li  a0,0
la  a1, buffer
li  a2,255
ecall
lw  a0, 0(a1)
andi a0,a0,255
li  s11, 0x30
sub  a0,a0,s11
mv   t1,a0

add  s11,t1,t0 # somma degli operandi

li  a7,1      # syscall 1 (print_int)
mv  a0,s11
ecall

exit:
li  a7,10
ecall
```


Input/Output da console [cont.]

Il programma esegue la somma dei due valori inseriti



The image shows a screenshot of a debugger's console window. At the top, there are two tabs: 'Console' and 'Memory'. The 'Console' tab is selected. Below the tabs, the word 'Console' is written. The main area of the console displays the following text: 'Introduci il primo valore: 3', 'Introduci il secondo valore: 4', '7', and 'Program exited with code: 0'. A vertical cursor is positioned at the end of the last line of text.

```
Console Memory
Console

Introduci il primo valore: 3
Introduci il secondo valore: 4
7
Program exited with code: 0
|
```

Esempio codice : Ricerca del carattere minimo

```
.data
buffer:      .zero 255
bVet:        .zero 5
bRes:        .zero 1
message_in:  .string  "Inserire caratteri : "
message_out: .string  "\nValore Minimo : "

.text

main:        la t0, bVet                # puntatore a inizio del vettore
             li  t1,0                  # contatore

             la  a0,message_in          # indirizzo della stringa
             li  a7,4                  # system call stampa stringa
             ecall
```

Esempio codice : Ricerca del carattere minimo [cont]

```
ciclo1:
    # syscall 5 (read_int)
    li a7,63
    li a0,0
    la a1, buffer
    li a2,255
    ecall
    lw a0,0(a1)
    andi a0,a0,255
    li s11,0x30
    sub a0,a0,s11
    # system call (risultato in v0)
    sb a0,0(t0)
    addi t1,t1,1
    addi t0, t0,1
    li s11,5
    bne t1,s11,ciclo1      # itera 5 volte
    la t0,bVet
    li t1,0                # contatore
    lb t2,0(t0)            # in t2 memorizzo MIN iniziale
```

Esempio codice : Ricerca del carattere minimo [cont]

```
ciclo2: lb    t3,0(t0)
        bgt   t3,t2, salta      # salta se NON deve aggiornare MIN
        lb    t2,0(t0)         # aggiorna MIN
salta:  addi   t1,t1,1
        addi   t0,t0,1
        li     s11,5
        bne    t1,s11,ciclo2
          

        la     a0,message_out
        li     a7,4
        ecall
          

        li     a7,1            # stampa 1 int
        mv     a0,t2
        ecall
          

exit:   li     a7,10
        ecall
```

ASCII Table

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	À	224	E0	à
1	01	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	â	193	C1	Á	225	E1	á
2	02	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ã	194	C2	Â	226	E2	â
3	03	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ä	195	C3	Ã	227	E3	ã
4	04	End of xmit	EOT	CTRL-D	36	24	\$	68	44	D	100	64	d	132	84	ä	164	A4	å	196	C4	Ä	228	E4	ä
5	05	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e	133	85	å	165	A5	Ä	197	C5	Å	229	E5	å
6	06	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f	134	86	ä	166	A6	Å	198	C6	Æ	230	E6	æ
7	07	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	Ö	199	C7	Ø	231	E7	ø
8	08	Backspace	BS	CTRL-H	40	28	(72	48	H	104	68	h	136	88	è	168	A8	ç	200	C8	È	232	E8	è
9	09	Horizontal tab	HT	CTRL-I	41	29)	73	49	I	105	69	i	137	89	é	169	A9	Ö	201	C9	É	233	E9	é
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	Ö	202	CA	Ê	234	EA	ê
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	¾	203	CB	Ë	235	EB	ë
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¾	204	CC	Ë	236	EC	ë
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m	141	8D	ï	173	AD	ï	205	CD	¼	237	ED	¼
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	¼	206	CE	½	238	EE	½
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o	143	8F	Ä	175	AF	½	207	CF	½	239	EF	½
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p	144	90	E	176	B0	¾	208	D0	¾	240	F0	¾
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	¾	209	D1	¾	241	F1	¾
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	¾	210	D2	¾	242	F2	¾
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s	147	93	ø	179	B3	¾	211	D3	¾	243	F3	¾
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t	148	94	ø	180	B4	¾	212	D4	¾	244	F4	¾
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u	149	95	ð	181	B5	¾	213	D5	¾	245	F5	¾
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v	150	96	ü	182	B6	¾	214	D6	¾	246	F6	¾
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w	151	97	ü	183	B7	¾	215	D7	¾	247	F7	¾
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x	152	98	ý	184	B8	¾	216	D8	¾	248	F8	¾
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y	153	99	Û	185	B9	¾	217	D9	¾	249	F9	¾
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	¾	218	DA	¾	250	FA	¾
27	1B	Escape	ESC	CTRL-[59	3B	;	91	5B	[123	7B	{	155	9B	ý	187	BB	¾	219	DB	¾	251	FB	¾
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C		156	9C	£	188	BC	¾	220	DC	¾	252	FC	¾
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D]	125	7D	}	157	9D	¥	189	BD	¾	221	DD	¾	253	FD	¾
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~	158	9E	Þ	190	BE	¾	222	DE	¾	254	FE	¾
31	1F	Unit separator	US	CTRL-`	63	3F	?	95	5F	`	127	7F	DEL	159	9F	ÿ	191	BF	¾	223	DF	¾	255	FF	¾