OBIETTIVI

1) **Imparare ad eseguire semplici programmi RISCV all'interno del simulatore RARS**
2) **Imparare a scrivere ed interpretare semplici programmi RISCV**

---

1) Imparare ad eseguire semplici programmi RISCV all'interno del simulatore RARS

1. Scaricare il simulatore RARS a questo link: https://github.com/TheThirdOne/rars/releases/tag/v1.5
2. Aprire il simulatore con doppio click su rars.jar (oppure java -jar rars.jar da linea di comando linux/MacOsx)
3. Scaricare il programma fattoriale.s (v. Sito del corso di Architettura dei Calcolatori)
4. Avviare il simulatore RARS con il comando: java -jar cartella_rars/rars.jar
5. MENU: File/Open → fattoriale.s
6. Compilare cliccando su: MENU: Run→Assemble (Oppure premere F3)
7. Impostare "main" come label di partenza: MENU: Settings→Initialize Program Counter to global main
8. Eseguire il programma: MENU: Run → Go (Oppure premere F5)
9. Inserire il numero (es. 5)
10. Verificare il risultato (120)

Programma FATTORIALE RICORSIVORISCV

```
.data
    messaggio:    .asciz "Calcolo di n!\n"
    insdati:      .asciz "Inserisci n = "
    visris:       .asciz "n! = "
    RTN:          .asciz "\n"
```

```
.globl main
.text

# Funzione per il calcolo del fattoriale (versione per RV32IM)
# Parametri:   n  : a0 (x10)
# Risultato:   n! : a0 (x10)

fact:
# crea il call frame sullo stack (12 byte)
# lo stack cresce verso il basso
    addi sp, sp, -12      # allocazione del call frame nello stack
    sw   a0, 8(sp)        # salvataggio di n nel call frame
    sw   ra, 4(sp)        # salvataggio dell'indirizzo di ritorno
    sw   s0, 0(sp)        # salvataggio del precedente frame pointer
    add  s0, sp, zero     # aggiornamento del frame pointer

# calcolo del fattoriale
    bne  a0, zero,  Ric   # test fine ricorsione n!=0
    addi a0, zero, 1      # 0! = 1
    j    Fine

Ric:                      # chiamata ricorsiva per il calcolo di (n-1)!
    addi a0, a0, -1       # a0 <- (n   1)  passaggio del parametro in a0 per fact(n-1)
    jal  fact             # chiama fact(n-1) -> risultato in a0
    lw   t0, 8(s0)        # t0 <- n
    mul  a0, a0, t0       # n! = (n-1)! x n

# uscita dalla funzione
Fine:
    lw   s0, 0(sp)        # recupera il frame pointer
    lw   ra, 4(sp)        # recupera l'indirizzo di ritorno
    addi sp, sp, 12       # elimina il call frame dallo stack
    ret                   # ritorna al chiamante
```

```
# Programma principale
main:
# Stampa intestazione
    la   a0, messaggio
    addi a7, zero, 4
    ecall

# Stampa richiesta
    la   a0, insdati
    addi a7, zero,  4
    ecall

# legge n (valore in a0)
    addi a7, zero,  5
    ecall

# chiama fact(n)
    jal  fact          # parametro n in a0
    add  s0, a0, zero   # salva il risultato in s0
```

```
# stampa messaggio per il risultato
    la   a0, visris
    addi a7, zero,  4
    ecall

# stampa n!
    add  a0, s0, zero
    addi a7, zero,  1
    ecall

# stampa \n
    la   a0, RTN
    addi a7, zero,  4
    ecall

# exit
    addi a7, zero,  10
    ecall
```

2) Imparare a scrivere ed interpretare semplici programmi RISCV

Programma SOMMATORIA.S:

1. Scrivere direttamente in assembly RISCV un programma per CALCOLARE E STAMPARE IL VALORE DELLA SOMMA PER k CHE VA DA 1 A 10 DEI TERMINI k*(k+1) E STAMPARE IL RISULTATO (il risultato deve venire 440)
2. Focalizzare l'attenzione sulle parti esterne (es. cicli) completandone la scrittura e poi scrivere la parte centrale del ciclo:

```
.data                          #SEZIONE DATI
outstr:
  .asciz "Il risultato e': "# dichiara una stringa terminante con zero

.text                          #SEZIONE CODICE
main:                          # indica l'inizio del codice
     addi t0, x0, 1            # t0 e' un contatore inizializzato a 1 (n)
     addi t1, x0, 0            # t1 contiene la somma
     addi t2, x0, 10           # t2 contiene il limite del loop

loop_ini:
     slt  t4, t2, t0
     bne  t4, x0 loop_end      # esce dal loop se t0 > 10

     addi t3, t0, 1            # da n calcola (n+1) in t3
     mul  t3, t0, t3           # t3 = Lo (=t0*t3) (non uso Hi (mulh)...valori piccoli)
     add  t1, t1, t3           # t1 = t1 + t3
     addi t0, t0, 1            # incremento il contatore
     j    loop_ini             # salta a loop_ini

loop_end:                      # stampa la stringa di introduzione al risultato
     la   a0, out_str          # carica l'indirizzo della string da stampare in a0
     addi a7, x0,4             # system call per il servizio 4 (stampa stringa)
     ecall                     # chiama il sistema operativo per effettuare il servizio

     add  a0, x0, t1           # mette dentro a0 l'intero che e' in t1
     addi a7, x0,1             # system call per il servizio 1 (stampa un intero)
     ecall                     # chiama il sistema operativo per effettuare il servizio

     addi a7, x0,10            # system call per il servizio 1 (exit)
     ecall                     # chiama il sistema operativo per effettuare il servizio
```

PARTE CENTRALE DEL CICLO

RISCV Instructions (RV64IMFD) **v221117**

Instruction coding (hexadecimal)			Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
funct7/imm	funct3	opcode				
00	0	33/3b	**add**	`add/addw x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
20	0	33/3b	**subtract**	`sub/subw x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
imm	0	13/1b	**add immediate**	`addi/addiw x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
01	0	33/3b	**multiply**	`mul/mulw x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
01	1	33	**multiply high**	`mulh x5,x6,x7`	x5 ← x6 * x7	Higher 64bits of 128-bits product
01	4	33/3b	**division**	`div/divw x5,x6,x7`	x5 ← x6/x7	(signed/word) division (divw**)
01	6	33/3b	**reminder**	`rem/remw x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
00	2/3	33	**set on less than**	`slt/sltu x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
imm	2/3	13	**set on less than immediate**	`slti/sltiu x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (less than)
00	7/6/4	33	**and / or / xor**	`and/or/xor x5,x6,x7`	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR
imm	7/6/4	13	**and /or / xor immediate**	`andi/ori/xori x5,x6,100`	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR register, constant
0	1	33/3b	**shift left logical**	`sll/sllw x5,x6,x7`	x5 ← x6 << x7	Shift left by register (sllw**)
imm	1	13/1b	**shift left logical immediate**	`slli/slliw x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
0	5	33/3b	**shift right logical**	`srl/srlw x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
imm	5	13/1b	**shift right logical immediate**	`srli/srliw x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
20	5	33/3b	**shift right arithmetic**	`sra/sraw x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
imm	5	13/1b	**shift right arithmetic immediate**	`srai/sraiw x5,x6,10`	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
imm	3/2/0	03	**load dword / word / byte**	`ld/lw/lb x5,100(x6)`	x5 ← MEM[x6+100]	Data from memory to register
imm	6/4	03	**load word / byte unsigned**	`lwu/lbu x5,100(x6)`	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
imm	3/2	23	**store dword / word / byte**	`sd/sw/sb x5,100(x6)`	MEM[x6+100] ← x5	Data from register to memory (sw**)
imm[31:12]	-	37	**load upper immediate**	`lui x5,0x12345`	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION			**load address**	`la x5,var`	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	`REAL: lui x5,H20(&var);ori x5, L12(&var) INST.` (H20=high 20 bits of &var; L12=low 12 bits of &var)
imm[31:12] (rd=0) imm[11:0] (rs1=rs2=0)	- 0	6f/63	**jump/branch**	`j/b label`	PC+=off (off=PC-&label) **(PS.INST.)**	`REAL INST.: jal x0,offset/beq x0,x0,offset`
imm[31:12] (rd=1)	-	6f	**jump and link (offset)**	`jal label`	x1←(PC+4); PC+=offset **(PS. INST.)**	`REAL INST.: jal x1,offset` (offset=PC-&label)
Imm (rd=0,rs=1)	0	67	**return from procedure**	`ret`	PC←x1 **(PSEUDO INST.)**	`REAL INST.: jalr x0,0(x1)`
imm	0	67	**jump and link register**	`jalr x1, 100(x5)`	x1 ← (PC + 4); PC=x5+100	Procedure return; indirect call
imm÷2	0/1	63	**branch on equal / not-equal**	`beq/bne x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
00 (rs1=0,rs2=0,rd=0)	0	73	**ecall**	`ecall`	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
08 (rs1=0,rs2=2,rd=0)	0	73	**sret**	`sret`	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION			**move**	`mv x5,x6`	x5 ← x6 **(PSEUDO INST.)**	`REAL INST.: add x5,x0,x6`
PSEUDOINSTRUCTION			**load immediate**	`li x5,100`	x5 ← 100 **(PSEUDO INST.)**	`REAL INST.: addi x5,x0,100`
PSEUDOINSTRUCTION			**no operation (nop)**	`nop`	do nothing **(PSEUDO INST.)**	`REAL INST.: addi x0,x0,0`
{0,1} / {4,5}	0	53	**floating point add/sub**	`fadd/fsub.{s,d} f0,f1,f2`	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
{8,9} / {c,d}	0	53	**floating point multiplication/divisio**	`fmul/fdiv.{s,d} f0,f1,f2`	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION			**floating point move between f-regs**	`fmv.{s,d} f0,f1`	f0←f1 **(PSEUDO INST.)**	`REAL INST.: fsgnj.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point negate**	`fneg.{s,d} f0,f1`	f0← – (f1) **(PSEUDO INST.)**	`REAL INST.: fsgnjn.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point absolute value**	`fabs.{s,d} f0,f1`	f0← | f1 | **(PSEUDO INST.)**	`REAL INST.: fsgnjx.{s,d} f0,f1,f1`
{50,51}	0/1/2	53	**floating point compare**	`fle/flt/feq.{s,d} x5,f0,f1`	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
{70,71} (rs2=0)	0	53	**move between x (integer) and f regs**	`fmv.x.{s,d} x5,f0`	x5←f0 (no conversion)	Copy (no conversion)
{78,79} (rs2=0)	0	53	**move between f and x regs**	`fmv.{s,d}.x f0,x5`	f0←x5 (no conversion)	Copy (no conversion)
imm	2	7	**load/store floating point (32bit)**	`flw/fsw f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
imm	3	7	**load/store floating point (64bit)**	`fld/fsd f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
21/20 (rs2=0)	7	53	**convert to/from double from/t single**	`fcvt.d.s/fcvt.s.d f0,f1`	f0← (double)f1 / f0← (single)f1	Type conversion
{60,61} (rs2=0)	7	53	**convert to integer fron {single,double}**	`fcvt.w.{s,d} x5,f0`	x5← (int)f0	Type conversion
{68,69} (rs2=0)	7	53	**convert to {single,double} fron integer**	`fcvt.{s,d}.w f0,x5`	f0← ({single,double})x5	Type conversion
{2c,2d} (rs2=0)	0	53	**square root**	`fsqrt.{s,d} f0,f1`	f0← square root of f1	Single or double square root
{10,11}	0/1/2	53	**sign injection**	`fsgnj/jn/jx.{s,d} f0,f1,f2`	f0←sgn(f2)|f1| / –sgn(f2)|f1| / sgn(f2)|f1|	Extract the mantissa and exp. from f1 and sign from f2

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

Appendice 1

Visualizzazione dello stato dello stack nel programma fact(3):

factorial.s – "text" (codice assembly caricato a 0x40'0000)

Text Segment

pt	Address	Code	Basic		Source
	0x00400000	0xfe810113	addi x2,x2,0xffffffe8	18:	addi sp, sp, -24 # allocazione del call frame nello stack
	0x00400004	0x00a13823	sd x10,0x00000010(x2)	19:	sd a0, 16(sp) # salvataggio di n nel call frame
	0x00400008	0x00113423	sd x1,0x00000008(x2)	20:	sd ra, 8(sp) # salvataggio dell'indirizzo di ritorno
	0x0040000c	0x00813023	sd x8,0x00000000(x2)	21:	sd s0, 0(sp) # salvataggio del precedente frame pointer
	0x00400010	0x00010433	add x8,x2,x0	22:	add s0, sp, zero # aggiornamento del frame pointer
	0x00400014	0x00051663	bne x10,x0,0x00000006	25:	bne a0, zero, Ric # test fine ricorsione n!=0
	0x00400018	0x00100513	addi x10,x0,0x00000001	26:	addi a0, zero, 1 # 0! = 1
	0x0040001c	0x0140006f	jal x0,0x0000000a	27:	j Fine
	0x00400020	0xfff50513	addi x10,x10,0xffff...	30:	addi a0, a0, -1 # a0 <- (n - 1) passaggio del parametro in a0 per fact(n-1)
	0x00400024	0xfddff0ef	jal x1,0xfffffffee	31:	jal fact # chiama fact(n-1) -> risultato in a0
	0x00400028	0x01043283	ld x5,0x00000010(x8)	32:	ld t0, 16(s0) # t0 <- n
	0x0040002c	0x02550533	mul x10,x10,x5	33:	mul a0, a0, t0 # n! = (n-1)! x n
	0x00400030	0x00013403	ld x8,0x00000000(x2)	37:	ld s0, 0(sp) # recupera il frame pointer
	0x00400034	0x00813083	ld x1,0x00000008(x2)	38:	ld ra, 8(sp) # recupera l'indirizzo di ritorno
	0x00400038	0x01810113	addi x2,x2,0x00000018	39:	addi sp, sp, 24 # elimina il call frame dallo stack
	0x0040003c	0x00008067	jalr x0,x1,0x00000000	40:	jr ra # ritorna al chiamante
	0x00400040	0x0fc10517	auipc x10,0x0000fc10	48:	la a0, messaggio
	0x00400044	0xfc050513	addi x10,x10,0xffff...		
	0x00400048	0x00400893	addi x17,x0,0x00000004	49:	addi a7, zero, 4
	0x0040004c	0x00000073	ecall	50:	ecall
	0x00400050	0x0fc10517	auipc x10,0x0000fc10	53:	la a0, insdati
	0x00400054	0xfbf50513	addi x10,x10,0xffff...		
	0x00400058	0x00400893	addi x17,x0,0x00000004	54:	addi a7, zero, 4
	0x0040005c	0x00000073	ecall	55:	ecall
	0x00400060	0x00500893	addi x17,x0,0x00000005	58:	addi a7, zero, 5
	0x00400064	0x00000073	ecall	59:	ecall
	0x00400068	0xf99ff0ef	jal x1,0xffffffcc	62:	jal fact # parametro n in a0
	0x0040006c	0x00050433	add x8,x10,x0	63:	add s0, a0, zero # salva il risultato in s0
	0x00400070	0x0fc10517	auipc x10,0x0000fc10	66:	la a0, visris
	0x00400074	0xfae50513	addi x10,x10,0xffff...		
	0x00400078	0x00400893	addi x17,x0,0x00000004	67:	addi a7, zero, 4
	0x0040007c	0x00000073	ecall	68:	ecall
	0x00400080	0x00040533	add x10,x8,x0	71:	add a0, s0, zero
	0x00400084	0x00100893	addi x17,x0,0x00000001	72:	addi a7, zero, 1
	0x00400088	0x00000073	ecall	73:	ecall
	0x0040008c	0x0fc10517	auipc x10,0x0000fc10	76:	la a0, RTN
	0x00400090	0xf9850513	addi x10,x10,0xffff...		
	0x00400094	0x00400893	addi x17,x0,0x00000004	77:	addi a7, zero, 4
	0x00400098	0x00000073	ecall	78:	ecall
	0x0040009c	0x00a00893	addi x17,x0,0x0000000a	81:	addi a7, zero, 10
	0x004000a0	0x00000073	ecall	82:	ecall

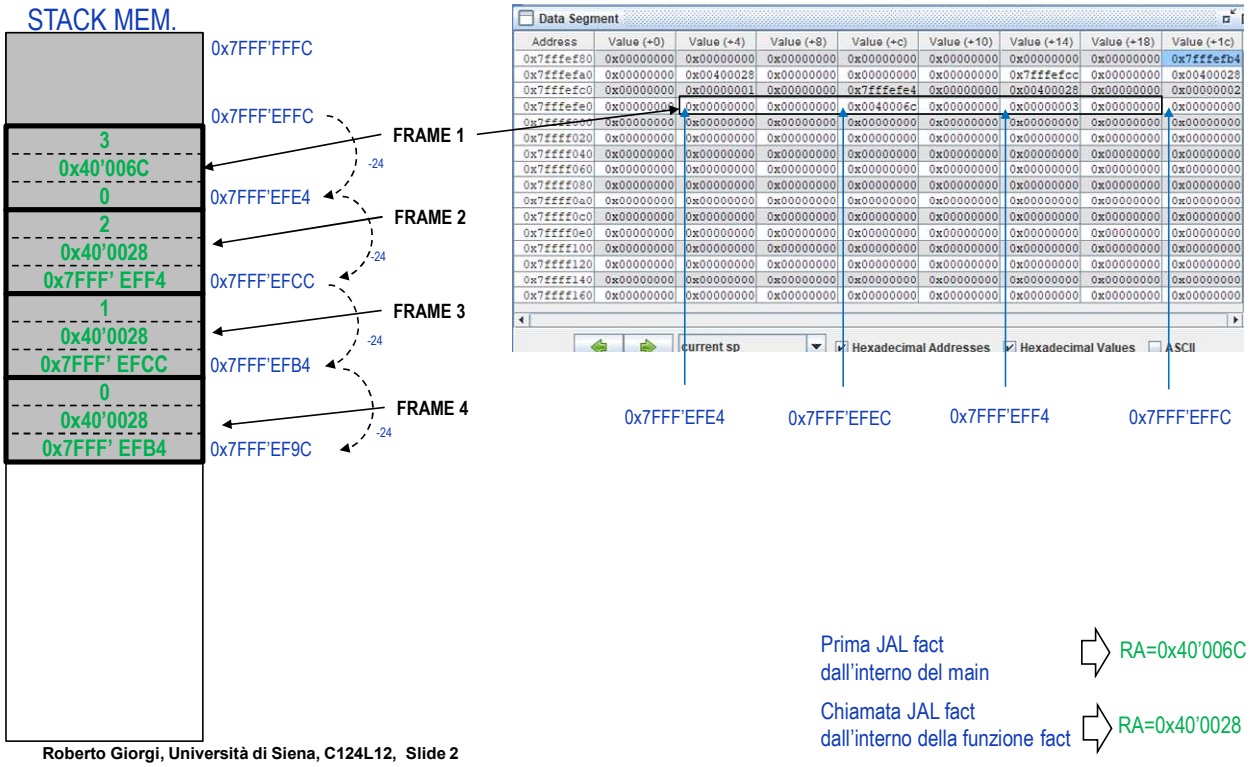
Symbol Table

Label	Address ▲
(global)	
main	0x00400040
fattoriale_riscv.s	
fact	0x00400000
Ric	0x00400020
Fine	0x00400030
messaggio	0x10010000
insdati	0x1001000f
visris	0x1001001e
RTN	0x10010024

Chiamata JAL fact
dall'interno della funzione fact
RA=0x40'0028

Prima JAL fact
dall'interno del main
RA=0x40'006C

Evoluzione dello stack durante la chiamata ' fact(3)'



ESERCIZI in LABORATORIO – NUMERI FLOATING-POINT E ASSEMBLY RISC-V (ESERCITAZIONE/LAB #12) V.2023

OBIETTIVI

1) **Eseguire programmi con "instrumentazione" (ovvero "sonde software")**
2) **Eseguire programmi costituiti da due moduli**
3) **Eseguire programmi con operazioni floating-point**
4) **Sviluppo programmi floating-point che operano su matrici (da provare da soli:)** 😊
5) **Esercizio sul passaggio da numeri floating-point a rappresentazione IEEE-754**
6) **Differenza float versus double**

1) Eseguire programmi con "instrumentazione" (ovvero "sonde software")

DOMANDA: (compito del 03-11-09) https://arcal.diism.unisi.it/esercizi1/c1091103.pdf

Trovare il codice assembly RISC-V corrispondente ai seguenti micro-benchmark (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), rispettando le convenzioni di uso dei registri dell'assembly RISC-V (riportate in calce, per riferimento). Successivamente, calcolare il tempo di esecuzione di ciascuno dei due benchmark ipotizzando che vengano eseguiti su un processore con frequenza di clock pari a 1 GHz, assumendo i seguenti valori per il CPI di ciascuna categoria di istruzioni: aritmetico-logiche-salti 1, branch 3, load-store 10.

<BENCHMARK-1: la funzione e' invocata con fib_it(10)>

```
int fib_it(int n) {
  int tmp, first = 0, second = 1;
  while (n--) {
      tmp = first+second;
      first = second;
      second = tmp;
  }
  return first;
}
```

<BENCHMARK-2: la funzione e' invocata con fib_rc(10)>

```
int fib_rc(int n) {
   if (n == 0) return (0);
   if (n == 1) return (1);
   else return (fib_rc(n-1)+(fib_rc(n-2));
}
```

RISCV Instructions (RV64IMFD) **v210622**

Instruction coding (hexadecimal) opcode+funct3+{funct7,imm}	Instruction	Example		Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
33+0+00/3b+0+00	**add**	**add/addw**	**x5,x6,x7**	x5 ← x6 + x7	Add two operands; exception possible (addw**)
33+0+20/3b+0+20	**subtract**	**sub/subw**	**x5,x6,x7**	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
13+0+imm/1b+0+imm	**add immediate**	**addi/addiw**	**x5,x6,100**	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
33+0+01/3b+0+01	**multiply**	**mul/mulw**	**x5,x6, x7**	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
33+01+01	**multiply high**	**mulh**	**x5,x6,x7**	x5 ← x6 * x7	Higher 64bits of 128-bits product
33+4+01/3b+4+01	**division**	**div/divw**	**x5,x6,x7**	x5 ← x6/x7	(signed/word) division (divw**)
33+6+01/3b+6+01	**reminder**	**rem/remw**	**x5,x6,x7**	x5 ← x6 % x7	Reminder of the division (remw**)
33+2+0/33+3+0	**set on less than**	**slt/sltu**	**x5,x6,x7**	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
13+2+imm/13+3+imm	**set on less than immediate**	**slti/sltiu**	**x5,x6,100**	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (less than)
33+7+0/33+6+0/33+4+0	**and / or / xor**	**and/or/xor**	**x5,x6,x7**	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR
13+7+imm/13+6+imm/13+4+imm	**and /or / xor immediate**	**andi/ori/xori**	**x5,x6,100**	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR register, constant
33+1+0/3b+1+0	**shift left logical**	**sll/sllw**	**x5,x6,x7**	x5 ← x6 << x7	Shift left by register (sllw**)
13+1+imm/1b+1+imm	**shift left logical immediate**	**slli/slliw**	**x5,x6,10**	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
33+5+0/3b+5+0	**shift right logical**	**srl/srlw**	**x5,x6,x7**	x5 ← x6 >> x7	Shift right by register (srlw**)
13+5+imm/1b+5+imm	**shift right logical immediate**	**srli/srliw**	**x5,x6,10**	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
33+5+20/3b+5+20	**shift right arithmetic**	**sra/sraw**	**x5,x6,x7**	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
13+5+imm/1b+5+imm	**shift right arithmetic immediate**	**srai/sraiw**	**x5,x6,10**	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
03+3+imm/03+2+imm/03+0+imm	**load dword / word / byte**	**ld/lw/lb**	**x5,100(x6)**	x5 ← MEM[x6+100]	Data from memory to register
03+6+imm/03+4+imm	**load word / byte unsigned**	**lwu/bu**	**x5,100(x6)**	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
23+3+imm/23+2+imm/23+0+imm	**store dword / word / byte**	**sd/sw/sb**	**x5,100(x6)**	MEM[x6+100] ← x5	Data from register to memory (sw**)
37+imm[31:12] (no funct3)	**load upper immediate**	**lui**	**x5,0x12345**	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION	**load address**	**la**	**x5,var**	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	**REAL: lui x5,H20(&var);ori x5, L12(&var) INST.** (H20=high 20 bit of &var; L12=low 12 bits of &var)
6f+imm[31:12](rd=0) 63+0+imm[11:0](rs1=rs2=0)	**jump/branch**	**j/b**	**label**	PC+=off (off=PC-&label) **(PS.INST.)**	**REAL INST.: jal x0,offset/beq x0,x0,offset**
6f+0+imm[31:12](rd=1,no funct3)	**jump and link (offset)**	**jal**	**label**	x1←(PC+4);PC+=offset **(PS. INST.)**	**REAL INST.: jal x1,offset** (offset=PC-&label)
67+0+imm (rd=0,rs1=1)	**return from procedure**	**Ret**		PC←x1 **(PSEUDO INST.)**	**REAL INST.: jalr x0,0(x1)**
67+0+imm	**jump and link register**	**jalr**	**x1, 100(x5)**	x1 ← (PC + 4);PC=x5+100	Procedure return; indirect call
63+0+(imm÷2)/63+1+(imm÷2)	**branch on equal / not-equal**	**beq/bne**	**x5,x6,100**	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
73+0+0 (rs1=0,rs2=0,rd=0)	**ecall**	**Ecall**		SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
73+0+8 (rs1=0,rs2=2,rd=0)	**sret**	**Sret**		PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION	**move**	**mv**	**x5,x6**	x5 ← x6 **(PSEUDO INST.)**	**REAL INST.: add x5,x0,x6**
PSEUDOINSTRUCTION	**load immediate**	**li**	**x5,100**	x5 ← 100 **(PSEUDO INST.)**	**REAL INST.: addi x5,x0,100**
PSEUDOINSTRUCTION	**no operation (nop)**	**nop**		do nothing **(PSEUDO INST.)**	**REAL INST.: addi x0,x0,0**
53+0+{0,1}/53+0+{4,5}	**floating point add/sub**	**fadd/fsub.{s,d}**	**f0,f1,f2**	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
53+0+{8,9}/53+0+{c,d}	**floating point multiplication/division**	**fmul/fdiv.{s,d}**	**f0,f1,f2**	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION	**floating point move between f-regs**	**fmv.{s,d}**	**f0,f1**	f0←f1 **(PSEUDO INST.)**	**REAL INST.: fsgnj.{s,d} f0,f1,f1**
PSEUDOINSTRUCTION	**floating point negate**	**fneg.{s,d}**	**f0,f1**	f0← – (f1) **(PSEUDO INST.)**	**REAL INST.: fsgnjn.{s,d} f0,f1,f1**
PSEUDOINSTRUCTION	**floating point absolute value**	**fabs.{s,d}**	**f0,f1**	f0← | f1 | **(PSEUDO INST.)**	**REAL INST.: fsgnjx.{s,d} f0,f1,f1**
53+0/1/2+{50,51}	**floating point compare**	**fle/flt/feq.{s,d}**	**x5,f0,f1**	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
53+0+{70,71} (rs2=0)	**move between x (integer) and f regs**	**fmv.x.{s,d}**	**x5,f0**	x5←f0 (no conversion)	Copy (no conversion)
53+0+{78,79} (rs2=0)	**move between f and x regs**	**fmv.{s,d}.x**	**f0,x5**	f0←x5 (no conversion)	Copy (no conversion)
7+2+imm/27+2+imm	**load/store floating point (32bit)**	**flw/fsw**	**f0,0(x5)**	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
7+3+imm/27+3+imm	**load/store floating point (64bit)**	**fld/fsd**	**f0,0(x5)**	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
53+7+21(rs2=0)/53+7+20(rs2=0)	**convert to/from double from/to single**	**fcvt.d.s/fcvt.s.d**	**f0,f1**	f0← (double)f1 / f0← (single)f1	Type conversion
53+7+{60,61} (rs2=0)	**convert to integer from {single,double}**	**fcvt.w.{s,d}**	**x5,f0**	x5← (int)f0	Type conversion
53+7+{68,69} (rs2=0)	**convert to {single,double} from integer**	**fcvt.{s,d}.w**	**f0,x5**	f0← ({single,double})x5	Type conversion
53+0+{2c,2d} (rs2=0)	**square root**	**fsqrt.{s,d}**	**f0,f1**	f0← square root of f1	Single or double square root
53+0/1/2+{10,11}	**sign injection**	**fsgnj/jn/jx.{s,d}**	**f0,f1,f2**	f0←sgn(f2)|f1| /–sgn(f2)|f1| /sgn(f2)f1	Extract the mantissa and exp. from f1 and sign from f2

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---

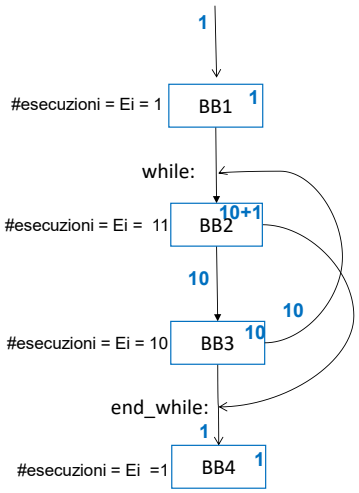
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

SOLUZIONE ("CARTA E PENNA")

Una possibile soluzione per il primo micro-benchmark e' (*la funzione fib_it viene chiamata con input 10: fib_it(10)*):

```
fib_it:
#_____________________________________________________BB1
   li   a4, 0             # first = 0
   li   a5, 1             # second = 1

while:
#_____________________________________________________BB2
   mv   t0, a0            # metto un attimo da parte a0
   addi a0, a0, -1        # n = n - 1
   beq  t0, x0,end_while # se n==?0 termino il ciclo

#_____________________________________________________BB3
   add  t0, a4, a5        # tmp = first+second
   mv   a4, a5            # first = second
   mv   a5, t0            # second = tmp
   j while

end_while:
#_____________________________________________________BB4
   mv   a0,a4             # in a4 ho fib
   ret                    # passo il controllo al chiamante
```



Il partizionamento del programma in basic-block e' fatto prendendo sequenze di una o più istruzioni consecutive che: o terminano con una istruzione di branch condizionale (B) o una jump incondizionata (J o JR) o RET (ma non una JAL), oppure terminano subito prima di un'etichetta di salto. Sono state trascurate tutte quelle parti di codice non richieste espressamente dalla traccia. Sia E_i il numero di esecuzioni del basic-block i-esimo, derivabile dall'analisi del funzionamento programma quando il parametro di ingresso n≡a0 vale 10, come indicato dal testo dell'esercizio. Ricordando che $C_{CPU} = N_{CPU} \cdot \overline{CPI} = N_{CPU} \cdot \sum_{k=ALJ,B,LS} \frac{N_{CPU,k}}{N_{CPU}} \cdot CPI_k = \sum_{k=ALJ,B,LS}^{T} N_{CPU,k} \cdot CPI_k$ e applicando questa relazione a ciascun basic-block i-esimo e sommando tutti i contributi i-esimi ai cicli $C_{CPU,i}$ di ogni basic-block si ottengono i cicli totali $C_{CPU} = \sum_{i=1..4} C_{CPU,i}$

BB_i	$N_{CPU,ALJ}$ (CPI_{ALJ} = 1)	$N_{CPU,B}$ (CPI_B = 3)	$N_{CPU,LS}$ (CPI_{LS} =10)	$(N_{CPU,k} \times CPI_k)_i$	E_i	$C_{CPU,i} = (N_{CPU,k} \times CPI_k)_i * E_i$
BB1	2	0	0	2	1	2 cc
BB2	2	1	0	5	11	55 cc
BB3	4	0	0	4	10	40 cc
BB4	2	0	0	2	1	2 cc
					C_{CPU} (in cicli di clock)	**99 cc**

Si ha quindi che:

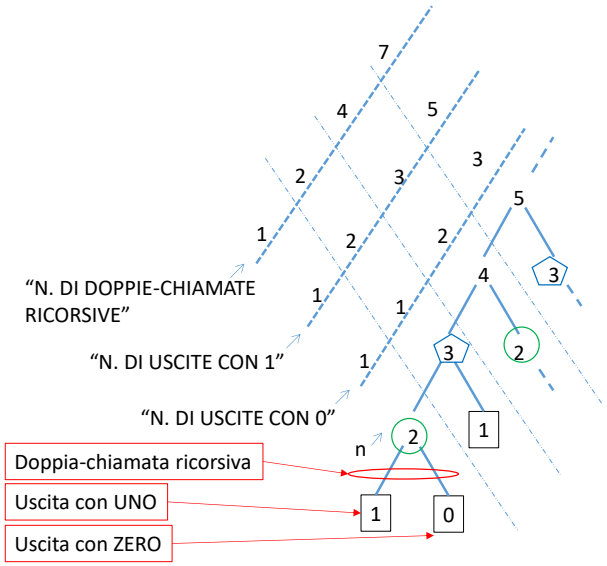
$$T_{CPU}^{1} = \frac{C_{CPU}^{1}}{f_{CPU}} = \frac{99}{10^9} = 99ns$$

Una possibile soluzione per il secondo micro-benchmark e' (*la funzione fib_rc e' richiamata con input 10: fib_rc(10)*):

```
fib_rc:
#_____________________________________________________________________________BB1
   addi sp, sp, -8  # riservo due word nello stack
   sw   ra, 0(sp)   # salvo l'indirizzo di ritorno nella prima word dello stack
   li   t0, 0       # metto il valore fisso di confronto 0 in t0
   beq  a0, t0,exit # se t0==0 allora ho finito

#_____________________________________________________________________________BB2
   li   t0, 1       # metto il valore fisso di confronto 1 in t0
   beq  a0, t0,exit # se t1==1 allora ho finito

#_____________________________________________________________________________BB3

   addi a0, a0, -1  # a0 = n-1
   sw   a0, 4(sp)   # salva (n-1) nello stack
   jal  fib_rc      # chiama fib_rc(n-1): risultato in a0
   lw   t0, 4(sp)   # ripristino ex-a0 in t0 (n-1)
   sw   a0, 4(sp)   # salva fib_rc(n-1) nello stack
   addi a0, t0, -1  # a0=n-2   (era t0=n-1)
   jal  fib_rc      # chiama fib_rc(n-2): risultato in a0c
   lw   t0, 4(sp)   # ripristina ex-a0 in t0 (fib_rc(n-1))
   add  a0, a0, t0  # somma fib_rc(n-2) e fib_rc(n-1)

exit:
#_____________________________________________________________________________BB4
   lw   ra, 0(sp)   # recupero l'indirizzo di ritorno dallo stack
   addi sp, sp, 8   # incremento lo stack
   ret              # passo il controllo al chiamante
```

Il numero di chiamate dei basic-block si evolve così (V. Anche Appendice-1):

i) Per n=2 (cerchietto verde in basso) →1, per n=3 (pentagono blu in basso) →1, per n=4 sono quelle nel ramo di n=3 + quelle nel ramo di n=2, quindi si ottiene una successione di Fibonacci con un n traslato, infatti per n=4 →2 (ovvero F_3), per n=5 → 3 (ovvero F_4), quindi tali chiamate seguono una successione di Fibonacci, dove **il numero di chiamate "zero" al livello n e' pari a F_{n-1}**. (c.f. tabellina della successione di Fibonacci nella stessa figura).

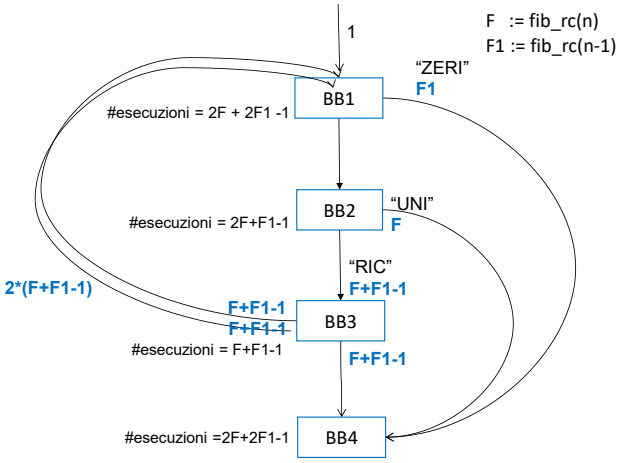


n	Fn
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

ii) Ragionando in modo analogo, sempre osservando la precedente figura, il numero di chiamate che si risolve restituendo direttamente "uno" si evolve così: per n=2 →1, per n=3 →2, per n=4, come nel caso precedente sono quelle nel ramo di n=3 + quelle nel ramo di n=2, quindi si parla sempre di una successione di Fibonacci, ma stavolta per n=2,3,4 ottengo 1,2,3 quindi **il numero di chiamate "uno" e' proprio F_n**.

iii) Infine, se la funzione non restituisce subito "zero" o "uno" significa che effettua la doppia chiamata ricorsiva a fib_rc(n-1) e fib_rc(n-2) e contando dalla stessa figura il numero di tali doppie chiamate si trova: per n=2 →1, per n=3 →2, per n=4 una doppia-chiamata che e' quella del n corrente + le chiamate sul ramo di n=3 + le chiamate sul ramo di n=2, quindi n=4→1+1+2=4, per n=5→1+2+4=7 e così via; quindi, nel caso generale ho 1+a+b dove per n=2,3,4,5,6,... a si evolve secondo 0,0,1,2,4,... e b secondo 0,1,2,4,7,... ovvero a=(F_{n-1}-1) e b=(F_n-1); pertanto **il numero di doppie-chiamate ricorsive (quelle all'interno di BB3) si evolve secondo 1+(F_{n-1}-1) +(F_n-1) = F_n+F_{n-1}-1**.

Sulla base delle precedenti considerazioni possiamo quindi riportare tali valori nel seguente diagramma di flusso che coinvolge i quattro basic-block, sui tre archi etichettati con "ZERI", "UNI" e "RIC" e dedurre quindi il numero di volte in qui transito su ognuno degli archi rimanenti. Infatti:



i) BB3 viene invocato $F_n + F_{n-1}$ -1 volte come già dedotto e dato che al suo interno chiama due volte nuovamente la funzione fib_rc significa che l'arco che richiama da BB3 il BB1 e' percorso 2 (F_n+F_{n-1}-1) volte.

ii) **BB1 viene quindi invocato 2 (F_n+F_{n-1}-1)** volte più la volta iniziale e quindi 2 F_n + 2F_{n-1} -1 volte.

iii) BB2 viene invocato (2 F_n + 2F_{n-1} -1) volte meno il numero di uscite dall'arco "ZERI", ovvero F_{n-1} quindi **2 F_n + F_{n-1} -1 volte**.

iv) BB4 viene invocato dall'arco "RIC", dall'arco "ZERI" e dall'arco "UNI" quindi (F_n+F_{n-1}-1) + (F_n) + (F_{n-1}) = 2 **F_n + 2F_{n-1} -1 volte**.

Ricapitolando:

BB1: 2 F_n + 2F_{n-1} -1→ per n=10 vale 177, BB2: 2 F_n + F_{n-1} -1→ per n=10 vale 143, BB3: F_n + F_{n-1} -1→ per n=10 vale 88, BB4: 2 F_n + 2F_{n-1} -1 → per n=10 vale 177 **QUINDI:**

BB_i	$N_{CPU,ALJ}$ (CPI_{ALJ} = 1)	$N_{CPU,B}$ (CPI_B = 3)	$N_{CPU,LS}$ (CPI_{LS} =10)	$(N_{CPU,k} \times CPI_k)_i$	E_i	$C_{CPU,i} = (N_{CPU,k} \times CPI_k)_i * E_i$
BB1	2	1	1	15	177	2655 cc
BB2	1	1	0	4	143	572 cc
BB3	5	0	4	45	88	3960 cc
BB4	2	0	1	12	177	2124 cc
					C_{CPU} (in cicli di clock)	**9311 cc**

VERIFICA AL CALCOLATORE (QUESTA SOLUZIONE CONSISTE NELL 'INSERIRE SONDE SOFTWARE NEL CODICE)

1) Scaricare il simulatore RARS (SI TRATTA DI UN FILE JAR)
2) Scaricare il programma (NOTA: COMPRENDE ENTRAMBI I MICROBENCHMARK): https://arcal.diism.unisi.it/esercizi1/c1091103-pro1_riscv.s
3) Cliccare sull'icona del simulatore RARS e poi MENU: File/Open → c1101103-pro1_riscv.s
4) Selezionare dal MENU Settings/Initialize_Program_Counter_to_global_'main'_if_defined
5) Selezionare dal MENU Assemble (F3) e poi Go (F5)
6) Inserire il numero (10)
7) Verificare il risultato → conteggio dei cicli e dei basic-blocks nel caso iterativo e ricorsivo:
   - caso iterativo: fib=55,Ni=1,11,10,1,Ccpu=98
   - caso ricorsivo: fib=55,Ni=177,143,88,177,Ccpu=9311

2) Eseguire programmi costituiti da due moduli

Es. n.2 del 02/11/2007: https://arcal.diism.unisi.it/esercizi1/c1071102-sol.pdf: Trovare il codice assembly RISC-V corrispondente del seguente programma (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), **rispettando le convenzioni di utilizzazione dei registri dell'assembly RISC-V** (riportate in calce, per riferimento). La funzione exp() riceve un numero di tipo "float" e ne restituisce l'esponenziale (è una funzione esterna da dichiarare opportunamente, in particolare a0=exp(a3)...).

```
#FILE N.1
#-----------------------------------------
.data
f:
 .float 0.0
A:
 .float 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0
ret:
 .asciz "\n"
esi:
 .asciz "esito="
nco:
 .asciz "   n.cond="


.text
 .globl main

   # non necessario su RARS: caricare exp.s e
ncond.s in sequenza
   # a3=**T
   # a1=n
   # a2=*nc
   # a0=r
   # t0=j
   # t1=k

num_cond:
   # spazio per a3,a0,t0,t1
   addi sp, sp, -24
   sw   ra, 0(sp)  # salva ra perche' usa altra f.
   sw   s0, 4(sp)  # salva fp perche' usa frame
   add  s0, sp, x0
   add  a0, x0, x0 # j = 1
   addi t0, x0, 1  # costante f.p. 1
   addi t2, x0, 1
   fcvt.s.w fa1, t2# costante f.p. 0
   addi t2, x0, 0
   fmv.s.x  fa0, t2# la codifica di 0.0 e' 0...0
while_ini:

   slt t2, a1, t0  # j>?n
   bne t2, x0, while_end   # se si while_end
   add t1, x0, x0  # k = 0
for_ini:
   slt t2, t1, a1  # k<?n
   beq  t2, x0, for_end   # j -1
   addi t3, t0, -1
   add  t2, t3, t3
   add  t2, t2, t3   # (j-1)*3
   add  t2, t2, t1   # (j-1)*3+k
   slli t2, t2, 2    # *4
   add  t2, t2, a3   # &T[j-1][k]
   flw  fa2, 0(t2)   # == 0.0
   feq.s t5,fa2,fa0
   bne  t5, x0, ramo_else
   sw   a3,  8(s0)   # salva a3
   sw   a0, 12(s0)   # salva a0
   sw   t0, 16(s0)   # salva t0
   sw   t1, 20(s0)   # salva t1
   fneg.s fa2, fa2   # cambio segno
   fmv.x.s a3, fa2
   jal  exp   # a0=exp(a3)
   fmv.s.x fa2, a0
   lw   t1, 20(s0)   # salva t1
   lw   t0, 16(s0)   # salva t0
   lw   a0, 12(s0)   # ripristina a0
   lw   a3, 8(s0)    # ripristina a3
   fdiv.s fa2, fa1, fa2   # 1/sqr(-T[][])
   flw  ft0, 0(a2)   # *nc
   fadd.s ft0, ft0, fa2   # +=
   fsw  ft0, 0(a2)   # *nc=
   j if_end
ramo_else:
   # r = 1
   addi a0, x0, 1
if_end:
   # ++k
   addi t1, t1, 1
   j for_ini
for_end:
   # ++j
   addi t0, t0, 1
   j while_ini
while_end:

   lw   ra, 0(s0)
   lw   s0, 4(s0)
   addi sp, sp, 24   # ritorna a0
   ret
main:
   # param.1
   la   a3,  A   # param.2
   addi a1, x0, 3   # param.3
   la   a2,  f
   jal  num_cond

   add  s0, a0, x0   #print "esi...
   la   a0,  esi
   addi a7,x0, 4
   ecall
   add  a0, s0, x0   # ripristina ec
   addi a7,x0, 1
   ecall   # print "nco...
   la   a0,  nco
   addi a7,x0, 4
   ecall   # print f
   la   a0,  f
   flw  fa0, 0(a0)
   addi a7,x0, 2
   ecall  # print ret
   la a0,  ret
   addi a7,x0, 4
   ecall   # exit

   addi a7,x0, 10
   ecall

#FILE N.2
#-----------------------------------------
.text
 .globl exp

exp:

   fmv.s.x ft5, a3
   fmul.s ft5, ft5, ft5
   fmv.x.s a0, ft5
   ret

OUTPUT:
```

```
Run I/O
esito=0   n.cond=1.5397677

-- program is finished running --
```

VERIFICA SU SIMULATORE

1) Scaricare il simulatore RARS (SI TRATTA DI UN FILE JAR)
2) Scaricare i programmi:
   https://arcal.dii.unisi.it/esercizi1/c1071102-pro1_riscv.s
   https://arcal.dii.unisi.it/esercizi1/c1071102-pro2_riscv.s
3) Cliccare sull'icona del simulatore RARS e poi MENU: File/Open → c1071102-pro1_riscv.s
   E POI ANCORA File/Open → c1071102-pro2_riscv.s
4) Selezionare dal MENU Settings/Initialize_Program_Counter_to_global_'main'_if_defined
5) Selezionare dal MENU Settings/**Assemble_all_files_currently_open**
6) Selezionare dal MENU Assemble (F3) e poi Go (F5)
7) Verificare il risultato: messaggio "esito=0 n.cond=1.5397677"

3) Eseguire programmi con operazioni floating point

1) Scaricare il simulatore RARS (SI TRATTA DI UN FILE JAR)
2) Scaricare il programma:
   https://arcal.dii.unisi.it/esercizi1/c1160210-pro1_riscv.s
3) Selezionare dal MENU Settings/Initialize_Program_Counter_to_global_'main'_if_defined
4) Cliccare sull'icona del simulatore RARS e poi MENU: File/Open → c1160210-pro1_riscv.s
5) Selezionare dal MENU Assemble (F3) e poi Go (F5)
6) Verificare il risultato:

```
X: 0.9375        1.9791667        -1.0059525         - iter =1      e2=0.08770908
X: 0.99949515        1.9999616        -1.0000668         - iter =2      e2=4.0286675E-5
X: 0.9999995         2.0000005        -1.0000002         - iter =3      e2=6.8213524E-10
X: 1.0       2.0         -1.0000001         - iter =4      e2=0.0
X: 1.0       2.0         -1.0000001         - iter =4      e2=0.0

-- program is finished running (dropped off bottom) --
```

4) Capire l'uso di dati float di una matrice

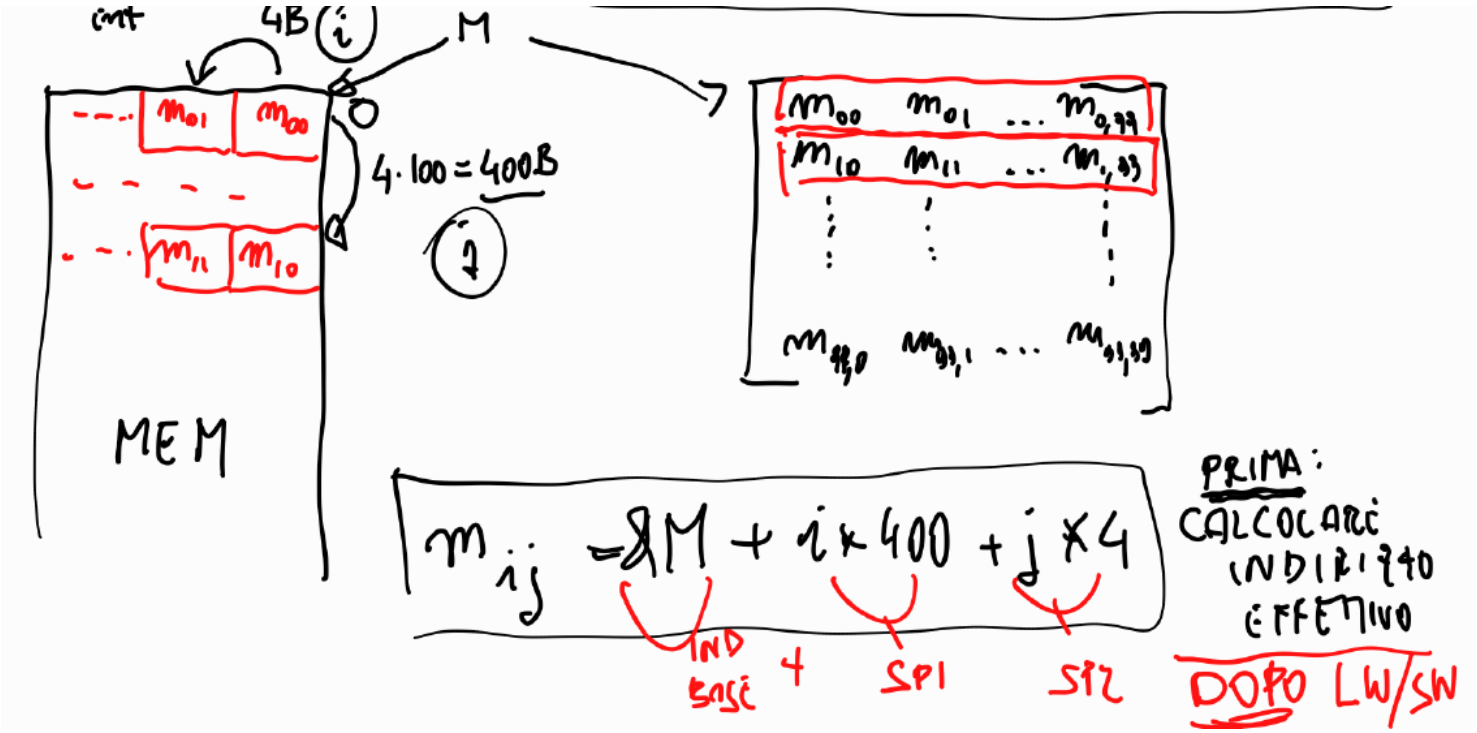
Es. n.1 - compito del 26/10/2005: https://arcal.diism.unisi.it/esercizi1/c1051026-sol.pdf

Trovare il codice assembly MIPS corrispondente dei seguenti micro-benchmark (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), rispettando le convenzioni di utilizzazione dei registri dell'assembly MIPS (riportate in calce, per riferimento).

```
<BENCHMARK-1:>
<M1,M2,MR sono matrici di 'int'>

for (j = 0; j < 100; ++j) {
   for (k = 0; k < 100; ++k) {
      t = 0;
      for (i = 0; i < 100; ++i) {
          t += M1[j][i]*M2[i][k];
      }
      MR[j][k] = t;
    }
}
```

```
<BENCHMARK-2:>
<la funzione e' invocata con fibo(10)>

int fibo(int n)
{
   if (n < = 2) return (1);
   else return (fibo(n-1)+(fibo(n-2));
}
```

Ci concentriamo qui su come tradurre l'accesso a un elemento della matrice nel primo micro-benchmark:



Questo corrisponde nel codice RISC-V a qualcosa del tipo:

```
# calcolo dell'indirizzo elemento M1[j][i]
        mul t6, t0, t3    # j * 400 e  prendo solo LO perche' j e' sempre < 100
        mul a4, t2, t4    # i * 4 e prendo solo LO perche' i e' sempre < 100
        add t6, t6, a4    # offset elemento: (j * 400) + (i * 4)
        add t6, a1, t6    # indirizzo elemento: sommo base (a1) e offset (t6)
         lw t5, 0(t6)      # carico M1[j][i] in t5
```

Per l'esempio completo:

https://arcal.diism.unisi.it/tools1/matrixmul_riscv.s

5) Esercizio sul passaggio da numeri floating-point a rappresentazione IEEE-754

Es. n.1 - compito del 2/11/2007: https://arcal.diism.unisi.it/esercizi1/c1081102-sol.pdf

DOMANDA:

Ricavare la rappresentazione binaria in formato IEEE-754 singola precisione del numero 0.045 supponendo di effettuare un arrotondamento al più vicino valore in virgola mobile rappresentabile nel suddetto formato.

SOLUZIONE:

Normalizzando 0.045 si ottiene: 1.44*2^-5. L' "uno" iniziale non viene rappresentato nel formato IEEE-754. Successivamente si puo' ricavare la rappresentazione binaria di 0.44 con 23 bit moltiplicando per 2 e ricavando la n-esima cifra più significativa della mantissa M:

M = 0111 0000 1010 0011 1101 011

(es. 0.44 *2 =0.88 → 0, 0.88*2 =1.76 → 1, 0.76*2 = 1.52 → 1, 0.52*2= 1.04 → 1, 0.04 *2 = 0.08 → 0, ...)

Per l'esponente, ricordando che nel caso di singola precisione il valore della polarizzazione e' 127:

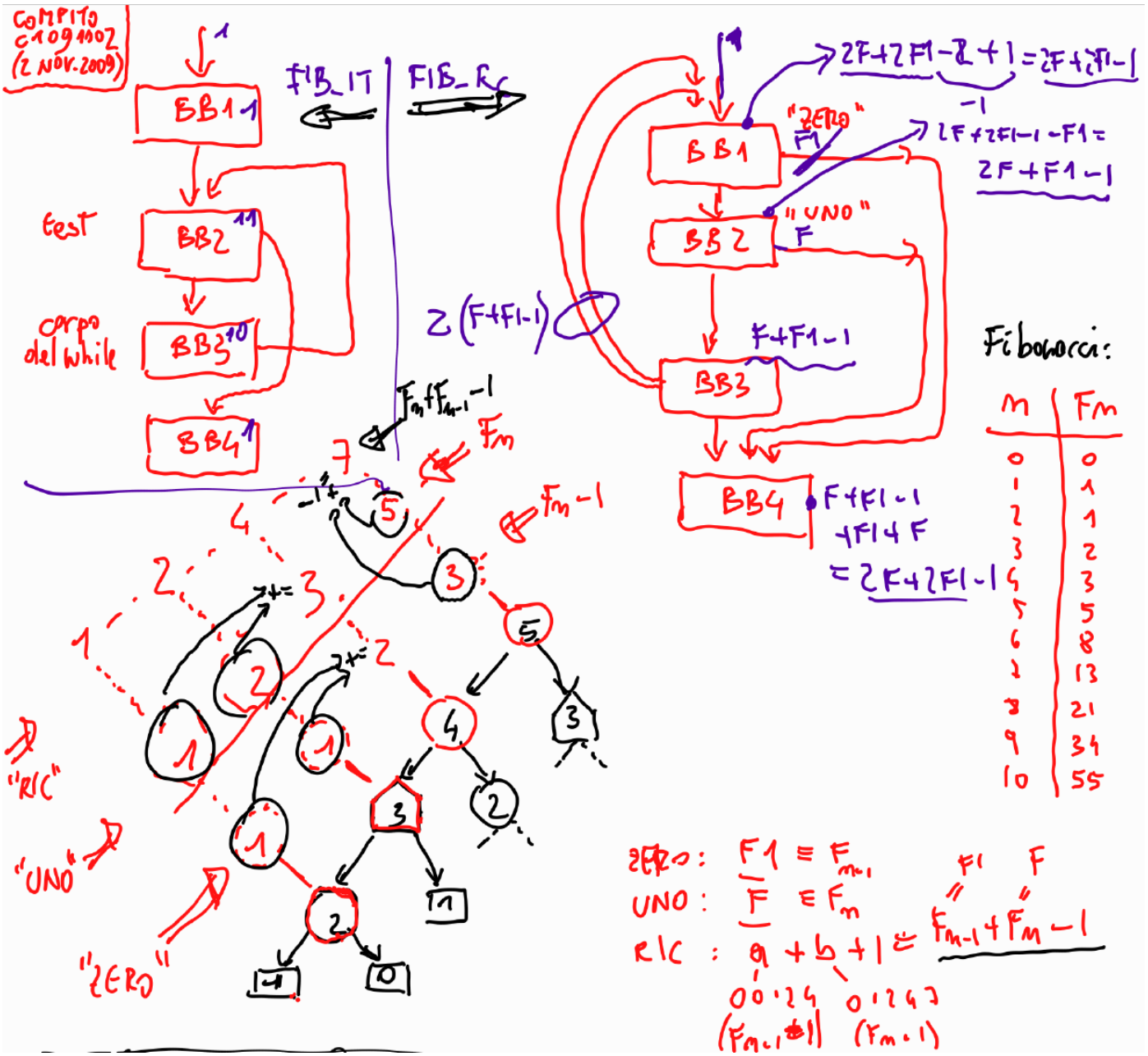
E = -5+127=122 ovvero 01111010

Osservando che le cifre binarie di M si ripetono dalla 21-esima cifra e in particolare la 24-esima cifra è un 1, si desume che la rappresentazione più vicina è quella superiore. Quindi la rappresentazione cercata e': 0 0111 1010 0111 0000 1010 0011 1101 100

6) Differenza float versus double

➔ Trasformare questo programma in modo che anziché usare i float usi i double.

APPENDICE 1



ESERCIZI in LABORATORIO – GESTIONE DRIVER (ESERCITAZIONE/LAB #13) (versione 231123)

OBIETTIVI

1) **Capire la struttura di un semplice timer e della sua routine di gestione dell'interrupt (usando RARS)**
2) **Confronto dei sistemi a polling, interrupt e DMA**
3) **Capire la gestione di driver e passaggio da modalità user e kernel**

1) Capire la struttura di un semplice timer e della sua routine di gestione dell'interrupt

Utilizzando il simulatore RARS è possibile scrivere sia la routine di gestione dell'interrupt del timer (all'interno del simulatore stesso) che il programma di inizializzazione di tale routine. Il programma main è costituito da una parte di setup e da una parte di ciclo di attesa infinita, durante il quale avvengono le interruzioni del timer. In Fig.2 è riportato un possibile programma in assembly per il processore RISC-V.

Nella Fig.1 viene descritta la differenza fra i registri di supporto alle interruzioni del processore RISC-V (registri CSR che iniziano con 'S', ovvero STVEC, SSTATUS, SIE) e gli analoghi registri del simulatore RARS (registri CSR che iniziano con 'U' ovvero UTVEC, USTATUS, UIE).

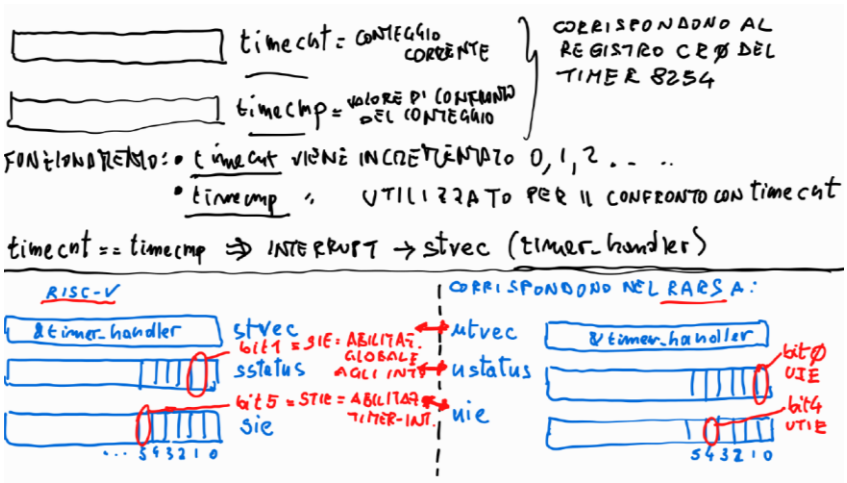


Fig.1. Registri di supporto all'interrupt del timer nel simulatore RARS.

Il codice del seguente programma può essere scaricato dal sito del corso:
https://arcal.diism.unisi.it/tools1/timer_riscv.s
Inoltre, seguire le seguenti istruzioni per provare il programma (usare RARS versione 1.4 o successiva):

1) Caricare il programma in RARS
2) Impostare "main" come label di partenza: RARS ->settings-> Initialize Program Counter to global "main" if defined (se non è già impostato)
3) Compilare il programma cliccando sul pulsante:
4) Lanciare il Timer Tool: RARS->Tools-> Timer Tool
5) Nel Timer Tools cliccare su "Connect to Program"
6) Eseguire il programma cliccando sul pulsante:
7) Nel Timer Tool Cliccare su "Play"

```
.data
message: .asciz "Got a timer interrupt\n"
newLine: .asciz "\n"
timecnt: .word 0xFFFF0018 # &TIMERCOUNT   reg. conteggio (ms)
timecmp: .word 0xFFFF0020 # &TIMERCOMPARE reg. confronto (ms)
savereg: .space 40

.text
.globl main

setup:
   # Initialize timecmp to trigger
   # the first timer interrupt after 5 secs
   la     a0, timecmp    # &timercmp
   ld     a0, 0(a0)      # &TIMERCOMPARE
   li     a1, 5000
   sd     a1, 0(a0)

   # Set the handler address and enable interrupts
   la     t0, timer_handler
   csrrw zero, utvec, t0 #handler address into the utvec

   # devo mettere a 1 il bit 4 di SIE
   # per abilitare gli interrupt dallo user-timer **
   csrrsi zero, uie, 0x10     # enable user timer interrupt

   # devo mettere a 1 il bit 0 di SSTATUS
   # per abilitare globalmente gli interrupt in user mode **
   csrrsi zero, ustatus, 0x1 # enable global user interrupts
   # ** NOTA: RARS usa la modalita USER
   #e non la modalita SYSTEM
   # (in modalita SYSTEM, i bit da mettere a 1 sarebbero
   # rispettivamente il bit 5 e il bit 1)
   ret


loop: # ciclo di attesa infinita
      # per attendere gli interrupt dal timer
   add    x0,x0,x0
   j      loop
```

```
timer_handler:
   #---------------------------------------------------------
   # PROLOGO:
   # Salvo TUTTI I REGISTRI CHE USO
   csrrw x0, uscratch,a4 # salvo a4 nello uscratch register
   la     a4, savereg     # (base della zona di salvataggio)
   sd     t2, 32(a4)      # NOTA: non posso usare lo stack!
   sd     t1, 24(a4)
   sd     t0, 16(a4)
   sd     a0, 8(a4)
   sd     a7, 0(a4)
   #---------------------------------------------------------

   # Print out the message
   li     a7, 4
   la     a0, message
   ecall

   # Re-initialize timer to interrupt every 5 seconds
   la     a0, timecnt    # &timercnt
   ld     a0,0(a0)       # &TIMERCOUNT
   ld     t2, 0(a0)
   li     t1, 5000
   add    t1, t2, t1     # add +5sec to current count
   la     t0, timecmp    # &timercmp
   ld     t0, 0(t0)      # &TIMERCOMPARE
   sd     t1, 0(t0)      # update time-compare register

   #---------------------------------------------------------
   # EPILOGO:
   # Reload the saved registers and return
   ld     t2, 32(a4)
   ld     t1, 24(a4)
   ld     t0, 16(a4)
   ld     a0, 8(a4)
   ld     a7, 0(a4)
   #---------------------------------------------------------
   uret    # back to the user (in RARS uso urete invece di sret)

main:
   jal setup
   jal loop
```

2) Confronto dei sistemi a polling, interrupt e DMA

Es. n.3 del 30/6/2016: https://arcal.diism.unisi.it/esercizi1/c1160630.pdf

Calcolare e confrontare i tempi di esecuzione in spazio Kernel (comprensivi dei tempi di setup dei controller e di gestione della stampa stessa) della stampa di un testo di lunghezza 1024 byte nei tre casi in cui si gestisca l'operazione con la tecnica di: i) polling; ii) interrupt; iii) DMA. Si utilizzino i seguenti tempi: A) per il setup del DMA controller 20 cicli; B) per acknowledgment di interrupt 4 cicli; C) per abilitazione di interrupt, per ritorno a User space e per ritorno da interrupt 2 cicli; D) per sbloccaggio utente 15 cicli; E) per passaggio di controllo allo scheduler e per riconoscimento dell'interrupt e lancio della routine di gestione dell'interrupt 3 cicli; F) nell'accesso ai registri di I/O della periferica (status, control, data): 2 cicli per ogni scrittura e 2 cicli per ogni lettura; G) ogni variabile temporanea e allocata in un registro del processore e ogni operazione del processore impiega sempre un ciclo (ALUJ, BRANCH, LOAD/STORE); H) si supponga che letture successive al registro di stato abbiano successo una volta ogni 10 accessi.

SVOLGIMENTO DELL'ESERCIZIO 2 (REVISIONE 2.4)

STAMPA A POLLING → 1+8'196+50'179+2=**58'378**

- copy_from_user(buffer, p, count); → * } 8'196
  for (k=0; k<count; ++k) { → 1 ciclo per statement iniziale +2 cicli (SLT+BNE)
  while (*printer_status_reg != READY); → 4(2 per lettura I/O+BNE+J)*10 (tentativi [H])
  *printer_data_register = p[k]; → 3(2 per calcolo offset+1 lettura)+ 2 (scrittura I/O) cicli
  } → 2 cicli (ADDI+J) → 1 + (2+40+5+2)*1024+2(ultima SLT+BNE)=50'179 cicli
  return_to_user(); → ([C]) 2 cicli

* copy_from_buffer:

```
(a0=&buffer,a1=&p,a2=count)

      ADD  t3, x0, x0     →1 # i=0
LOOP: SLT  t0, t3, a2     →1 # i<count
      BEQ  t0, x0,FINE    →1 # falso→fine
      ADDI t3, t3, 1      →1 # i++
      ADD  t0, a0, t3     →1 # &buffer[i]
      ADD  t1, a1, t3     →1 # &p[i]
      LW   t2, 0(t0)      →1 # t2=buffer[i]
      SW   t2, 0(t1)      →1 # p[i]=t2
      J LOOP              →1
FINE: RET                 →1

→1+1024*8+2(ultima slt+bne))+1= 8'196 cicli
```

STAMPA A INTERRUPT → 1+8'196+50+20'489=**28'726**

- Implementazione dalla system call:
  copy_from_user(buffer, p, count); → * } 8'196
  while (*printer_status_reg != READY); → 4(2 per lettura I/O+BNE+J)*10(tentativi)
  *printer_data_register = p[0]; → 2(scrittura I/O)+1(lettura) cicli
  count = count - 1; → (ADDI) 1 ciclo
  enable_interrupts(); → ([C]) 2 cicli
  scheduler(); → ([E]) 3 cicli
  } 50
- Implementazione della routine di servizio: la routine viene chiamata 1024 volte per trasferire 1023 byte (l'ultima chiamata serve solo a sbloccare l'utente; non fa un trasferimento effettivo)
  → 1024 * (3(ricon [E])+3(lanc.int [E])) + 1023*14 + 1*23=20'489
  if (count == 0) { → (BNE) 1 ciclo — Per 1 volta
  unblock_user(); → ([D]) 15 cicli
  } else { → (J jump) 1 ciclo — Per 1023 volte
  *printer_data_register = p[k]; → 3(2 per calcolo offset+1 lettura)+ 2 (scrittura I/O) cicli
  count = count - 1; → (ADDI) 1 ciclo
  k = k + 1; → (ADDI) 1 ciclo
  } 14
  acknowledge_interrupt(); → ([B]) 4cicli
  return_from_interrupt(); → ([C]) 2 cicli — 23

STAMPA A DMA → 1+8'196+23+3(ricon [E])+3(lanc.int [E])+21=**8'247**

- Implementazione dalla system call:
  copy_from_user(buffer, p, count); → * } 8'196
  setup_DMA_controller(); → ([A]) 20 cicli
  scheduler(); → ([E]) 3 cicli
- Implementazione della routine di servizio:
  acknowledge_interrupt(); → ([B]) 4 cicli
  unblock_user(); → ([D]) 15 cicli
  return_from_interrupt(); → ([C]) 2 cicli

Stampa di una stringa con Polling (Driver)

```
copy_from_user(buffer, p, count);          /* p è un buffer nel kernel */
for (k=0; k<count; ++k) {                  /* ripeti per ogni carattere */
   while (*printer_status_reg != READY);   /* attendi il bit di READY */
   *printer_data_register = p[k];          /* uscita di un carattere */
}
return_to_user();
```

Notare il punto e virgola !

Stampa di una stringa ad Interrupt

Codice di preparazione alla stampa (es. Implementazione di un system call):

```
copy_from_user(buffer, p, count);

while (*printer_status_reg != READY);
*printer_data_register = p[0];
count = count - 1;
enable_interrupts();
scheduler();
```

Trasferisco il primo carattere

La chiamata allo scheduler comporta il blocco (cioè la sua deschedulazione dalla CPU) del processo che ha invocato questa system call

Routine di servizio dell'interrupt

```
if (count == 0) {
  unblock_user();
} else {
  *printer_data_register = p[k];
  count = count - 1;
  k = k + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

Inizialmente avrò k=1

Stampa di una stringa con DMA

Codice di preparazione alla stampa (es. Implementazione di un system call):

```
copy_from_user(buffer, p, count);
setup_DMA_controller();
scheduler();
```

La chiamata allo scheduler comporta il blocco (cioè la sua deschedulazione dalla CPU) del processo che ha invocato questa system call→ il controllo viene quindi passato al sistema operativo che schedula un altro processo

Routine di servizio dell'interrupt

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

3) Capire la gestione di driver e passaggio da modalità user e kernel

Es. n.1 del 30/6/2016: https://arcal.diism.unisi.it/esercizi1/c1160630.pdf

Trovare il codice assembly RISC-V corrispondente del seguente programma (**utilizzando solo e unicamente istruzioni dalla tabella sottostante e rispettando le convenzioni di utilizzazione dei registri dell'assembly RISC-V** riportate qua sotto, per riferimento). Inoltre si realizzino sempre in assembly RISC-V le funzioni esterne della libreria "arduino", ipotizzando di utilizzare per la comunicazione seriale il chip 16550A mappato ad indirizzo 0x1001'01e0 e per generare ritardi il chip 8254 mappato ad indirizzo 0x1001'0040 (default trasmissione: 8 bit dati, parità dispari di zeri, 1 bit di stop) e per mantenere lo stato del led il bit4 di una porta posta ad indirizzo 0x1001'5678, per mantenere lo stato del pulsante il bit17 di una porta posta ad indirizzo 0x1001'4321. Notare che le funzioni di tale libreria devono risiedere tutto nello spazio Kernel. Si ricorda inoltre che il 16550A e' temporizzato con una frequenza Fc=1.8432MHz, mentre l'8254 con una frequenza Fc=1.19MHz.

```
#include <arduino.h>
#define INPUT 1
#define OUTPUT 0
#define HIGH 1
#define LOW 0

// Pin 13 has an LED connected
int led = 13;

// digital pin 2 has a pushbutton attached to it.
int pushButton = 2;
int buttonState = 0;

void setup() {
  pinMode(led, OUTPUT);

  // initialize serial communication
  // at 9600 bits per second:
  Serial.begin(9600);
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
}

void loop() {
  digitalWrite(led, HIGH); // turn the LED
  delay(1000);             // wait for a second

  // read the input pin:
  buttonState = digitalRead(pushButton);

  // print out the state of the button:
  Serial.println(buttonState);
  delay(10);               // delay 10ms

  digitalWrite(led, LOW);  // turn the LED off
  delay(1000);             // wait for a second
```

RISCV Instructions (RV64IMFD) **v221117**

Instruction coding (hexadecimal) funct7/imm	funct3	opcode	Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
00	0	33/3b	**add**	`add/addw x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
20	0	33/3b	**subtract**	`sub/subw x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
imm	0	13/1b	**add immediate**	`addi/addiw x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
01	0	33/3b	**multiply**	`mul/mulw x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
01	1	33	**multiply high**	`mulh x5,x6,x7`	x5 ← x6 * x7	Higher 64bits of 128-bits product
01	4	33/3b	**division**	`div/divw x5,x6,x7`	x5 ← x6/x7	(signed/word) division (divw**)
01	6	33/3b	**reminder**	`rem/remw x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
00	2/3	33	**set on less than**	`slt/sltu x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
imm	2/3	13	**set on less than immediate**	`slti/sltiu x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (less than)
00	7/6/4	33	**and / or / xor**	`and/or/xor x5,x6,x7`	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR
imm	7/6/4	13	**and /or / xor immediate**	`andi/ori/xori x5,x6,100`	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR register, constant
0	1	33/3b	**shift left logical**	`sll/sllw x5,x6,x7`	x5 ← x6 << x7	Shift left by register (sllw**)
imm	1	13/1b	**shift left logical immediate**	`slli/slliw x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
0	5	33/3b	**shift right logical**	`srl/srlw x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
imm	5	13/1b	**shift right logical immediate**	`srli/srliw x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
20	5	33/3b	**shift right arithmetic**	`sra/sraw x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
imm	5	13/1b	**shift right arithmetic immediate**	`srai/sraiw x5,x6,10`	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
imm	3/2/0	03	**load dword / word / byte**	`ld/lw/lb x5,100(x6)`	x5 ← MEM[x6+100]	Data from memory to register
imm	6/4	03	**load word / byte unsigned**	`lwu/lbu x5,100(x6)`	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
imm	3/2	23	**store dword / word / byte**	`sd/sw/sb x5,100(x6)`	MEM[x6+100] ← x5	Data from register to memory (sw**)
imm[31:12]	-	37	**load upper immediate**	`lui x5,0x12345`	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION			**load address**	`la x5,var`	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	`REAL: lui x5,H20(&var);ori x5, L12(&var) INST.` (H20=high 20 bits of &var; L12=low 12 bits of &var)
imm[31:12] (rd=0) imm[11:0] (rs1=rs2=0)	- 0	6f/63	**jump/branch**	`j/b label`	PC+=off (off=PC-&label) **(PS.INST.)**	`REAL INST.: jal x0,offset/beq x0,x0,offset`
imm[31:12] (rd=1)	-	6f	**jump and link (offset)**	`jal label`	x1←(PC+4); PC+=offset **(PS. INST.)**	`REAL INST.: jal x1,offset` (offset=PC-&label)
Imm (rd=0,rs=1)	0	67	**return from procedure**	`ret`	PC←x1 **(PSEUDO INST.)**	`REAL INST.: jalr x0,0(x1)`
imm	0	67	**jump and link register**	`jalr x1, 100(x5)`	x1 ← (PC + 4); PC=x5+100	Procedure return; indirect call
imm÷2	0/1	63	**branch on equal / not-equal**	`beq/bne x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
00 (rs1=0,rs2=0,rd=0)	0	73	**ecall**	`ecall`	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
08 (rs1=0,rs2=2,rd=0)	0	73	**sret**	`sret`	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION			**move**	`mv x5,x6`	x5 ← x6 **(PSEUDO INST.)**	`REAL INST.: add x5,x0,x6`
PSEUDOINSTRUCTION			**load immediate**	`li x5,100`	x5 ← 100 **(PSEUDO INST.)**	`REAL INST.: addi x5,x0,100`
PSEUDOINSTRUCTION			**no operation (nop)**	`nop`	do nothing **(PSEUDO INST.)**	`REAL INST.: addi x0,x0,0`
{0,1} / {4,5}	0	53	**floating point add/sub**	`fadd/fsub.{s,d} f0,f1,f2`	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
{8,9} / {c,d}	0	53	**floating point multiplication/divisio**	`fmul/fdiv.{s,d} f0,f1,f2`	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION			**floating point move between f-regs**	`fmv.{s,d} f0,f1`	f0←f1 **(PSEUDO INST.)**	`REAL INST.: fsgnj.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point negate**	`fneg.{s,d} f0,f1`	f0← – (f1) **(PSEUDO INST.)**	`REAL INST.: fsgnjn.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION			**floating point absolute value**	`fabs.{s,d} f0,f1`	f0← | f1 | **(PSEUDO INST.)**	`REAL INST.: fsgnjx.{s,d} f0,f1,f1`
{50,51}	0/1/2	53	**floating point compare**	`fle/flt/feq.{s,d} x5,f0,f1`	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
{70,71} (rs2=0)	0	53	**move between x (integer) and f regs**	`fmv.x.{s,d} x5,f0`	x5←f0 (no conversion)	Copy (no conversion)
{78,79} (rs2=0)	0	53	**move between f and x regs**	`fmv.{s,d}.x f0,x5`	f0←x5 (no conversion)	Copy (no conversion)
imm	2	7	**load/store floating point (32bit)**	`flw/fsw f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
imm	3	7	**load/store floating point (64bit)**	`fld/fsd f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
21/20 (rs2=0)	7	53	**convert to/from double from/t single**	`fcvt.d.s/fcvt.s.d f0,f1`	f0← (double)f1 / f0← (single)f1	Type conversion
{60,61} (rs2=0)	7	53	**convert to integer fror {single,double}**	`fcvt.w.{s,d} x5,f0`	x5← (int)f0	Type conversion
{68,69} (rs2=0)	7	53	**convert to {single,double} fror integer**	`fcvt.{s,d}.w f0,x5`	f0← ({single,double})x5	Type conversion
{2c,2d} (rs2=0)	0	53	**square root**	`fsqrt.{s,d} f0,f1`	f0← square root of f1	Single or double square root
{10,11}	0/1/2	53	**sign injection**	`fsgnj/jn/jx.{s,d} f0,f1,f2`	f0←sgn(f2)|f1| / –sgn(f2)|f1| / sgn(f2)*f1	Extract the mantissa and exp. from f1 and sign from f2

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

SVOLGIMENTO DELL'ESERCIZIO 3 in RARS (versione >=1.4)

Il codice dovrà essere organizzato in due moduli: A) codice utente; B) codice kernel (mini-drivers delle periferiche coinvolte).

```
# PARTE A (CODICE UTENTE)
.data
led: .word 13
pushButton:  .word 2
buttonState: .word 0
baudrate: .word 9600
#HANDLER DATA
saveframe:
#a0,a1,a2,t0...t4
.space 36
_syscallmsg: .asciz "Syscall "
_nl: .asciz "\n"
fc1:  .word 1190000
fc2:  .word 1843200

.text
.globl main
#-------------------------------------
# MAIN eseguito implicitam. da Arduino
main:
#loading the handler
la a3, handler
csrrw zero, utvec, a3 # set utvec
csrrsi zero, ustatus, 1 # set interrupt enable
jal setup

aloop:
jal loop
#EXIT
addi a7,x0,10
ecall

#-------------------------------------
# ARDUINO SETUP FUNCTION
setup:
#alloc.stackspace
addi sp, sp, -4
sw ra, 0(sp)    #punta a led
la t0, led
lw a2 , 0(t0)  #legge led
add a1, x0, x0 #setta II param.
addi a7,x0,25  #pinMode
ecall
lui a2,9600
addi a7,x0,21 #Serial.begin
ecall
la t0, pushButton #punta pushButton
lw a2 , 0(t0)      #setta II param.
addi a1, x0, 1
addi a7,x0,25     #pinMode
ecall
lw ra, 0(sp)     #deall.stackspace
addi sp, sp, 4
jr ra

#-------------------------------------
# ARDUINO LOOP FUNCTION
loop:
#alloc.stackspace
addi sp, sp, -4
sw ra, 0(sp)       #punta a led
la t0, led
lw a2 , 0(t0)      #setta II param.
addi a1, x0, 1
addi a7,x0,24     #digitalWrite
ecall
addi a2 , x0, 1000
addi a7,x0,26     #delay
ecall
la t0, pushButton
lw a2 , 0(t0)     #legge pushButton
addi a7,x0,23     #digitalRead
ecall
la t0, buttonState
sw a7, 0(t0)      #val. ritorno
add a2 , x0, a7   #prep. I param.
addi a7,x0,22     #Serial.println
ecall
addi a2 , x0, 10
addi a7,x0,26 #delay
ecall
la t0, led
lw a2 , 0(t0)      #setta II param.
addi a1, x0, 0
addi a7,x0,24      #digitalWrite
ecall
addi a2 , x0, 1000
addi a7,x0,26       #delay
ecall
lw ra, 0(sp)       #deall.stackspace
addi sp, sp, 4
jr ra


####EXCEPTION HANDLER -- SUPERVISOR MODE####
handler:
#saving registers
csrrw x0,uscratch,a4
la a4 , saveframe
sw a0, 0(a4)
sw a2 , 4(a4)
sw a1, 8(a4)
sw t0, 12(a4)
sw t1, 16(a4)
sw t2, 20(a4)
sw t3, 24(a4)
sw t4, 28(a4)# print_str
sw a7, 32(a4)

la a0, _syscallmsg
addi a7,x0,4
ecall #printf syscall str
lw a0 , 32(a4 )# print_int
addi a7,x0,1
ecall# print_str
la a0, _nl
addi a7,x0,4
ecall
# Switch to mysyscall code
lw a0, 0(a4)    # pop a0
lw a2 , 4(a4)   # pop a2
lw a1, 8(a4)    # pop a1
lw a7, 32(a4)
addi t0, x0, 21
beq t0, a7, syscall21
addi t0, x0, 22
beq t0, a7, syscall22
addi t0, x0, 23
beq t0, a7, syscall23
addi t0, x0, 24
beq t0, a7, syscall24
addi t0, x0, 25
beq t0, a7, syscall25
addi t0, x0, 26
beq t0, a7, syscall26
isr_ret:
# a0 (can be mysyscall output)
lw a5 , 0(a4)    # pop a0
addi t0, x0, 13
beq t0, a5 , dno_v0
lw a0, 0(a4)    # pop a0
lw a7, 32(a4)   #pop a7 parameter
dno_v0:
lw a2 , 4(a4)   # pop a2
lw a1, 8(a4)    # pop a1
lw t0, 12(a4)   # pop t0
lw t1, 16(a4)   # pop t1
lw t2, 20(a4)   # pop t2
lw t3, 24(a4)   # pop t3
lw t4, 28(a4)   # standard exception exit
lw a7, 32(a4)
csrrw a4, uepc,x0  #update EPC
addi a4, a4, 4
csrrw x0, uepc,a4
uret

### SUPERVISOR CODE (DRIVER)###
syscall21:
# a2=BR
lui t0, 0x10010
ori t0,t0, 0x01e0
addi t1, x0, 0xAB
# bit2=1stopbit, bit1-0=8bitframe
sb t1, 3(t0)
sb t1, 3(t0)
#calculate the time constant
slli t2,a2 , 4     #&fc2
la t1, fc2
lw t1, 0(t1)       #C=fc1/(BR*16)
div t1, t1, t2     #valore DL
sb t1, 0(t0)       #DLL (byte -signif)
srli t1,t1, 8      #val DLM in byte +sig.
sb t1, 0(t0)       # LCRval.(DLAB=0)
addi t1, x0, 0x2B
sb t1, 3(t0)
j isr_ret
syscall22:
lui t0, 0x10010
ori t0,t0, 0x01e0
sb a2 , 0(t0)     #write data byte
j isr_ret
syscall23:
addi t0, x0, 13      #LED
beq a2 , t1, isled1 #PUSHBUTTON
addi t0, x0, 2
beq a2 , t1, ispsh1 #ispushbutton
j iserror1
isled1:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x5678
or t0,t0, t4
lb t2, 0(t0)         #mask bit 4
andi t2, t2, 0x10
srli a0,t2, 4
j fine1

ispsh1:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x4321
or t0,t0, t4
lb t2, 0(t0)         #mask bit 7
andi t2, t2, 0x80
srli a0,t2, 7
j fine1

iserror1:
#do nothing for now
fine1:
j isr_ret


syscall24:
addi t0, x0, 13       #LED
beq a2 , t1, isled2  #PUSHBUTTON
addi t0, x0, 2
beq a2 , t1, ispsh2  #ispushbutton
j iserror2
isled2:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x5678
or t0,t0,t4
andi t1, a1, 1
srli t1,t1, 4        #select bit4
lb t2, 0(t0)         #clear bit4
andi t2, t2, 0xEF
or t3, t2, t1        #set bit4
sb t3, 0(t0)         #store
j fine2
ispsh2:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x4321
or t0,t0, t4
andi t1, a1, 1
srli t1,t1, 7       #select bit7
lb t2, 0(t0)        #clear bit7
andi t2, t2, 0x7F
or t3, t2, t1       #set bit7
sb t3, 0(t0)        #store
j fine2
iserror2:
#do nothing for now
fine2:
j isr_ret
syscall25:
addi t0, x0, 13       #LED
beq a2 , t1, isled3  #PUSHBUTTON
addi t0, x0, 2
beq a2 , t1, ispsh3  #ispushbutton
j iserror3
isled3:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x5678
or t0,t0, t4
lb t2, 0(t0)         #mask bit0 of a1
andi t1, a1, 1       #clear bit0
andi t2, t2, 0xFE
or t3, t2, t1        #set bit0
sb t3, 0(t0)
j fine3
ispsh3:
lui t0, 0x10010
ori t0,t0, 0x01e0
lui t4, 0x4321
or t0,t0, t4
lb t2, 0(t0)         #mask bit0 of a1
andi t1, a1, 1       #clear bit0
andi t2, t2, 0xFE
or t3, t2, t1        #set bit0
j fine3
iserror3:
#do nothing for now
fine3:
j isr_ret
syscall26:
# a2=delay
lui t0, 0x10010
ori t0, t0, 0x0040
la t0, fc1          #1.19 MHz
lw t0, 0(t0)
mul t2, a2 , t0    #count=delta*fc
addi t2, t2, -1    #count=delta*fc-1
andi t3, t2, 0xFF #LSB
srli t4,t2, 8      #MSB
andi t4, t4, 0xFF
addi t1, x0, 0x30 #bit7-6=counter0,
                   #bit5-4=LSB+MSB,
                   #bit3-1=mode-0,
                   #bit0=binary counter
la t0, fc1         #1.19 MHz
sb t1, 3(t0)       #write value in CWR
sb t3, 0(t0)       #write LSB in CR0
sb t4, 0(t0)       #write MSB in CR0
#start counting
addi t1, x0, 0xC2 #red CR0
 #counter latch cmd
sw a2 , 0(t0)      #SIMULA CONT.(init)
add t2, x0, a2
check:
addi t2, t2, -1  #SIMULA CONT.(dec)
sw t2, 0(t0)      #SIMULA CONT.(upd)
sb t1, 3(t0)      #CR0 counter latch cmd
lh t2, 0(t0)      #CR0-LSB (GIUSTO: lb)
lh t3, 0(t0)      #CR0-MSB (GIUSTO: lb)
bne t3, x0, check
bne t2, x0, check
#count finished CR0==0
j isr_ret
```

ESERCIZI in LABORATORIO – GESTIONE STRUTTURE, VETTORI, PUNTATORI, FUNZIONI (ESERCITAZIONE/LAB #14)

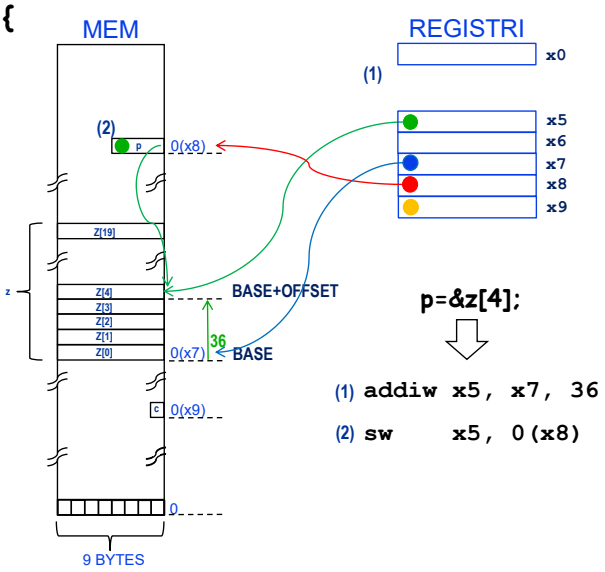
OBIETTIVI

A. **Capire la codifica di strutture e vettori nel RISC-V**
B. **Capire uso e codifica dei puntatori nel RISC-V**
C. **Analisi di vari casi di chiamata a funzione**

Sia data la seguente struttura:	e siano dichiarati quindi i seguenti dati:	come viene tradotta in assembly RISC-V
typedef struct tabletag { **int32 i;** **char *cp, c;** **} table;**	**table z[20];** **table *p;** **char c;**	la seguente operazione: **p=&z[4];** e la seguente operazione: **c=p->c;**

Operazione: p=&z[4]

```
typedef struct tabletag {
    int i;
    char *cp, c;
} table;

char c;
table z[20];
table *p;
```

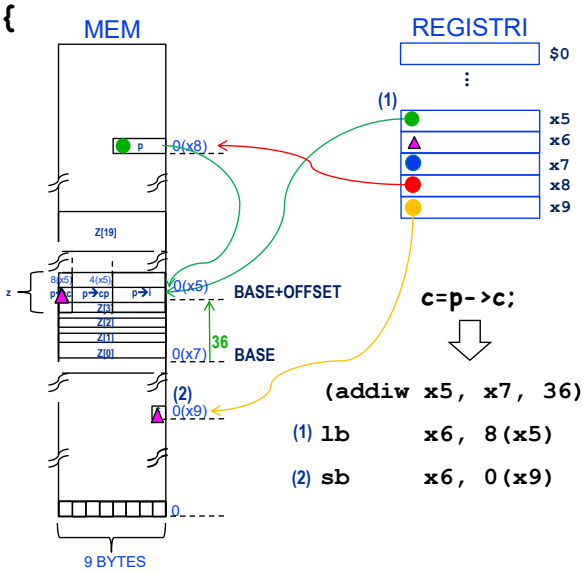


Ipotesi:

&z ≡ x7
&p ≡ x8
&c ≡ x9

Operazione: c=p->c

```
typedef struct tabletag {
    int i;
    char *cp, c;
} table;

char c;
table z[20];
table *p;
```

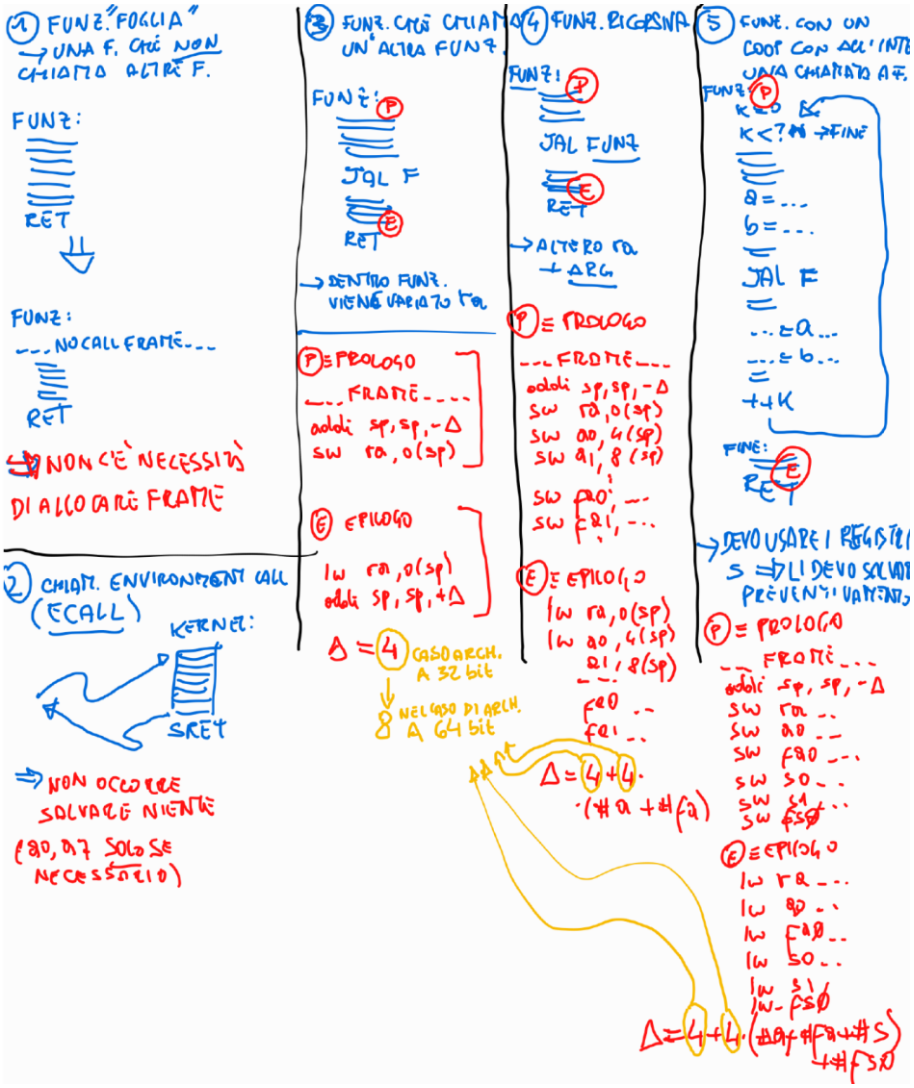


Ipotesi:

&z ≡ x7
&p ≡ x8
&c ≡ x9

Le chiamate a funzione possono essere fatte in diversi contesti: di seguito sono analizzati i casi più frequenti.

1) **Chiamata di "funzione foglia": è una funzione che NON chiama altre funzioni**
2) **Chiamata a System-Call (Environment Call): deve preservare l'ambiente**
3) **Funzione che chiama un'altra funzione**
4) **Funzione ricorsiva: la funzione richiama sé stessa**
5) **Funzione che contiene un loop con all'interno una chiamata a funzione**



Inoltre, ricordare

Funzione "main"

- E' una funzione come le altre
  - Occorre gestire in modo appropriato i registri ra, s, fs
  - Es. compitino del 17-12-2020

```
int main() {
    float dy = 0, x = bisect(-2, 1);
    if (x != 0) dy=df(x);
    print_float(dy); print_string("\n");
    return 0;
}
```

⇨

```
main:
addi sp,sp,-8     #alloco frame
sw   ra,0(sp)     #salvo ra
fsw  fs0,4(sp)    #salvo prec fs0

...
#return 0;
li      a0,0      #0
#}
lw      ra,0(sp)  #ripristino ra
flw     fs0,4(sp) #ripristino prec fs0
addi    sp,sp,8   #dealloco frame
ret
```

Nota2: se non espressamente indicato considerare registri a 32 bit → ogni elemento dello stack è 4 Byte. Se è indicato di operare con int64 (interi a 64 bit): sia i registri che gli elementi dello stack sono 64 bit (8B)

Nota3: RARS segnala un errore finale se il main finisce con RET (si aspetta una ECALL #10 i.e., «exit()») perché inizializza ra prima del main a 0 (questo non accade in un programma/sistema reale, in quanto il main è a sua volta chiamato da un pezzo di codice di startup)

Uso delle pseudistruzioni

- Limitare l'uso delle pseudoistruzioni (anche se in generale un assemblatore RISC-V e anche il RARS ne potrebbero fornire molte!)
- Le istruzioni e pseudoistruzioni consentite nei compiti sono quelle della tabella allegata al compito stesso (che è sempre la stessa)
- Le pseudoistruzioni che usiamo in questo corso sono solo queste:
  - LA t0, variabile
  - LI t0, 12345
  - MV t0, t1
  - CALL myfun (o JAL myfun)
  - RET (o JR ra)
  - B etichetta
  - NOP

  - FMV.S f0, f1 (anche .D)
  - FNEG.S f0, f1 (anche .D)
  - FABS.S f0, f1 (anche .D)
- ~~LW t0, variabile~~ → NON E' AMMESSA (per ridurre le istr. possibili)
  - usare la coppia: LA t0, variabile; LW t0, 0(t0)

ESERCIZI in LABORATORIO – CACHE ASSOCIATIVA (ESERCITAZIONE/LAB #15)

OBIETTIVI

1) **Capire il funzionamento della cache (cache a due vie)**
2) **Capire il funzionamento della cache (cache a tre vie)**

1) Capire il funzionamento della cache (cache a due vie)

Es. n.2 del 3/12/2003: https://arcal.diism.unisi.it/esercizi1/c1031203-sol.pdf

Si consideri una cache di dimensione 64B e a 2 vie. La dimensione del blocco e' 32 byte, il tempo di accesso alla cache e' 5 ns e la penalita' in caso di miss e' pari a 60 ns, la politica di rimpiazzamento e' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 21, 77, 66, 146, 82, 15, 130, 64, 192, 209, 225, 241, 113, 97, 273, 66, 257, 129, 64, 161, 240, 299. Per la sequenza data, ricavare il tempo medio di accesso alla cache, riportare i tag contenuti in cache al termine e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento.

```
A=2, B=32, C=64, RP=LRU, Thit=5, Tpen=60, 22 references:
=== T    X   XM  XT  XS  XB  H   USAGE   TAG
=== R   21    0   0   0  21  0   1,0     0,-
=== W   77    2   2   0  13  0   0,1     0,2
=== R   66    2   2   0   2  1   0,1     0,2
=== W  146    4   4   0  18  0   1,0     4,2
=== R   82    2   2   0  18  1   0,1     4,2
=== W   15    0   0   0  15  0   1,0     0,2
=== R  130    4   4   0   2  0   0,1     0,4
=== W   64    2   2   0   0  0   1,0     2,4
=== R  192    6   6   0   0  0   0,1     2,6
=== W  209    6   6   0  17  1   0,1     2,6
=== R  225    7   7   0   1  0   1,0     7,6
=== W  241    7   7   0  17  1   1,0     7,6
=== R  113    3   3   0  17  0   0,1     7,3
=== W   97    3   3   0   1  1   0,1     7,3
=== R  273    8   8   0  17  0   1,0     8,3
=== W   66    2   2   0   2  0   0,1     8,2
=== R  257    8   8   0   1  1   1,0     8,2
=== W  129    4   4   0   1  0   0,1     8,4
=== R   64    2   2   0   0  0   1,0     2,4
=== W  161    5   5   0   1  0   0,1     2,5
=== R  240    7   7   0  16  0   1,0     7,5
=== W  299    9   9   0  11  0   0,1     7,9
-------------------------------------
P1 Nmiss=16   Nhit=6  Nref=22  mrate=0.727273
```

```
(out: XM=0  XT=0  XS=0 )

(out: XM=4  XT=4  XS=0 )
(out: XM=2  XT=2  XS=0 )
(out: XM=0  XT=0  XS=0 )
(out: XM=4  XT=4  XS=0 )

(out: XM=2  XT=2  XS=0 )

(out: XM=6  XT=6  XS=0 )

(out: XM=7  XT=7  XS=0 )
(out: XM=3  XT=3  XS=0 )

(out: XM=2  XT=2  XS=0 )
(out: XM=8  XT=8  XS=0 )
(out: XM=4  XT=4  XS=0 )
(out: XM=2  XT=2  XS=0 )
(out: XM=5  XT=5  XS=0 )
```

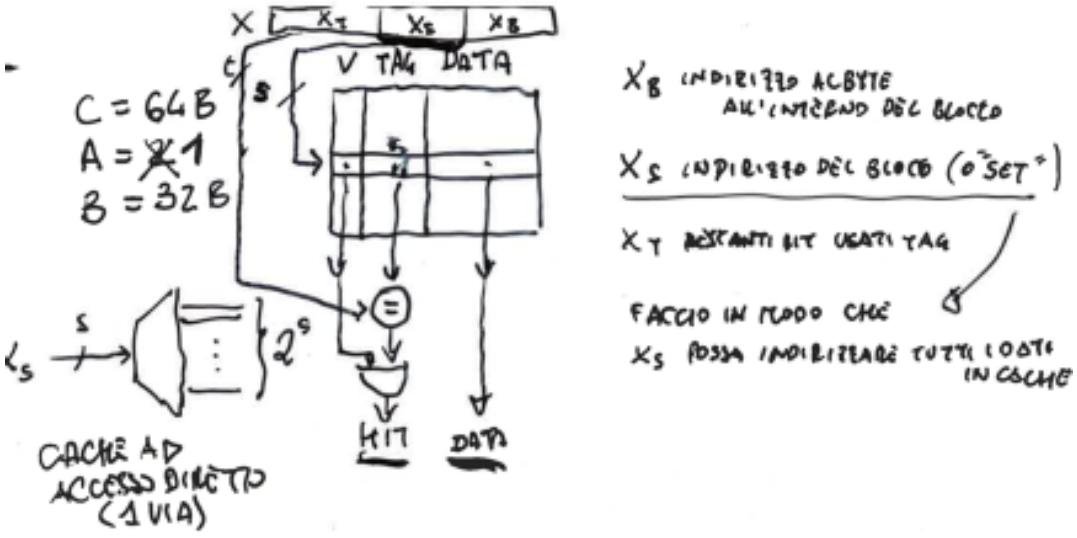
Situazione finale dei tag in cache

Blocchi uscenti

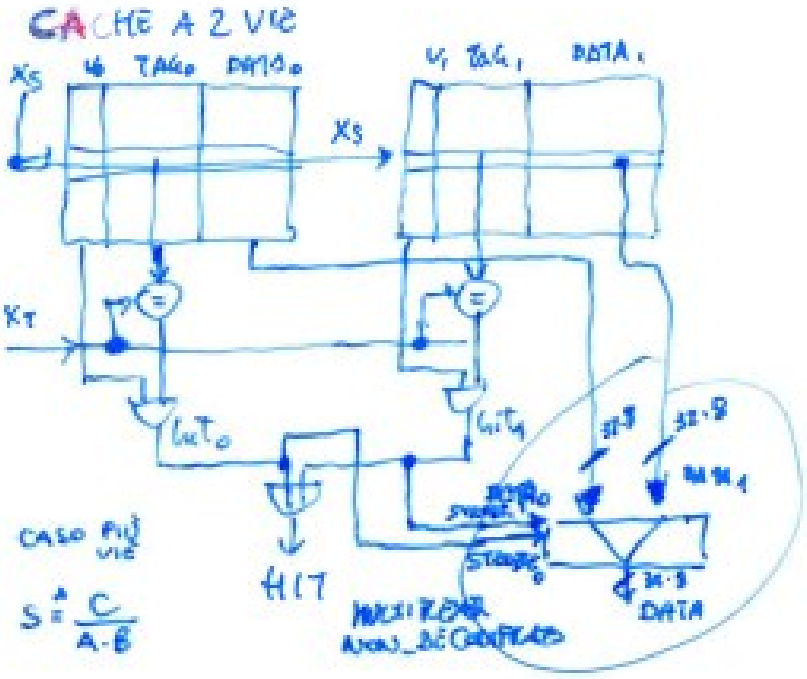
Si ricava quindi che il tempo medio di accesso alla cache e' pari a:

AMAT = $t_{hit}+t_{penalty}*m = t_{hit}+t_{penalty}* (N_{miss}/N_{ref}) = 5+60*16/22 \cong$ **48.63 ns**

Compito del 03-12-2003 - esercizio 2, note (1/3)



Compito del 03-12-2003 – esercizio 2, note (2/3)



X_M = IND. BLOCCO IN MEM. $= X/B$

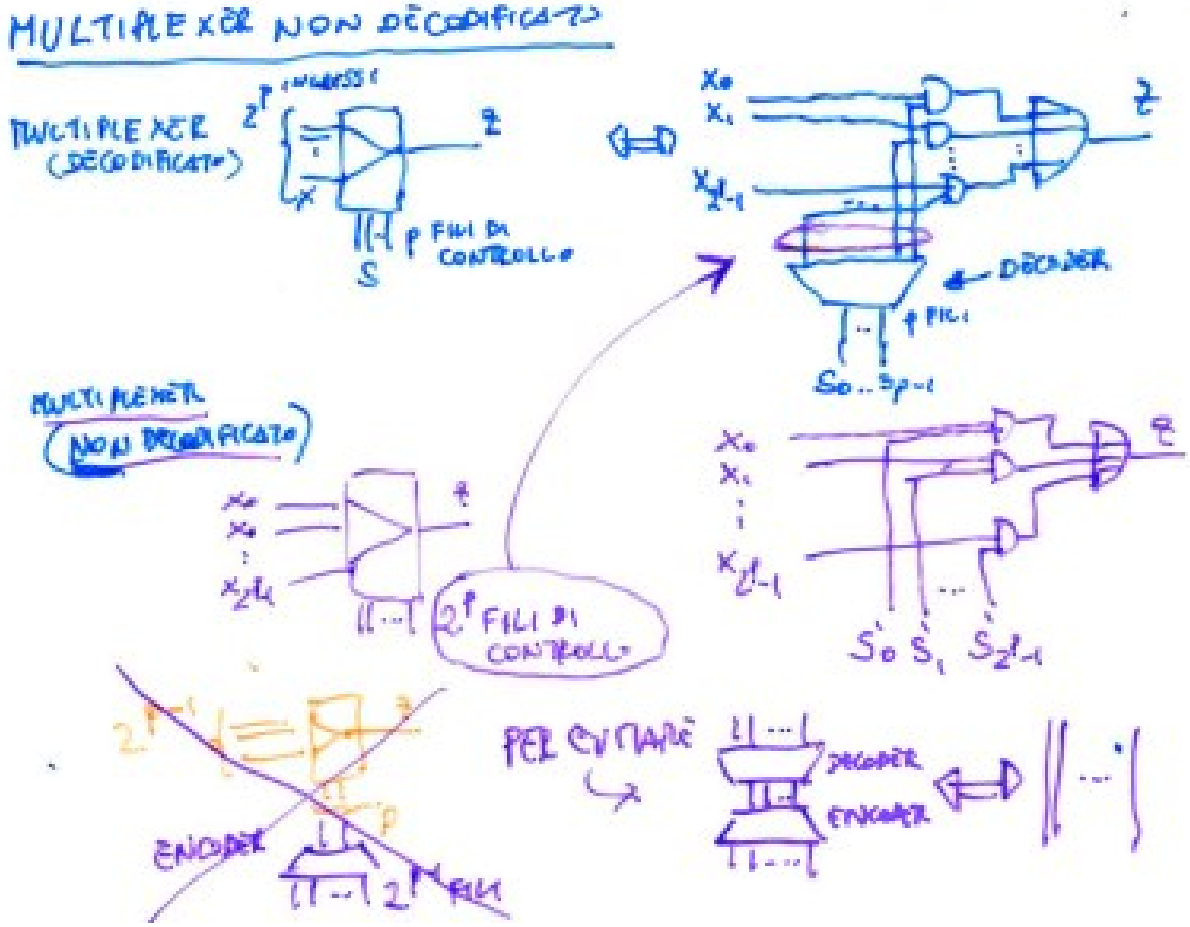
X_T = PARTE TAG DELL'IND. $= X_M/S$ $\quad S = \frac{C}{B}$

X_S = IND. SET (BLOCCO) $\Rightarrow X_M \bmod S$

(X_B = IND. BYTE NEL BLOCCO)

CASO PIÙ VIE: $S = \frac{C}{A \cdot B}$

Compito del 03-12-2003 – esercizio 2, note (3/3)



2) Capire il funzionamento della cache (cache a tre vie)

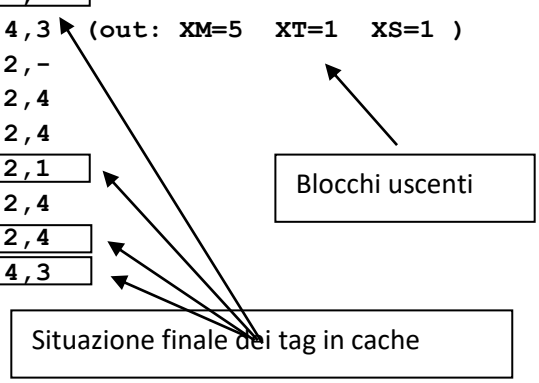
Es. n.2 del 30/11/2005: https://arcal.diism.unisi.it/esercizi1/c1051130-sol.pdf

Si consideri una cache di dimensione 192B e a 3 vie di tipo write-back. La dimensione del blocco e' 16 byte, il tempo di accesso alla cache e' 5 ns e la penalita' in caso di miss e' pari a 60 ns, la politica di rimpiazzamento e' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 22, 71, 65, 143, 81, 17, 133, 61, 190, 211, 212, 210, 115, 98, 275, 64, 259, 130, 61, 67, 70, 25. Tali accessi sono alternativamente letture e scritture. Per la sequenza data, ricavare il tempo medio di accesso alla cache, riportare i tag contenuti in cache al termine e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento.

NOTA: Si ipotizza che la politica di scrittura su miss sia di tipo **WNA** (Write Non-Allocate) ovvero in scrittura, se ho miss prima scrivo in memoria e quindi carico il blocco in stato coerente (non modificato → bit M=0). Se invece il blocco e'gia' in cache (hit) allora una scrittura comporta che il bit M ← 1.

Sia X il generico riferimento, A=associativita'=3, B=dimensione del blocco=16, C=capacita' della cache=192.
Si ricava S=C/B/A=# di set della cache=192/16/3=4, XM=X/B, XS=XM%S, XT=XM/S:

=== T	X	XM	XT	XS	XB	H	[SET]:USAGE	[SET]:MODIF	[SET]:TAG
=== R	22	1	0	1	6	0	[1]:2,0,0	[1]:0,0,0	[1]:0,-,-
=== W	71	4	1	0	7	0	[0]:2,0,0	[0]:0,0,0	[0]:1,-,-
=== R	65	4	1	0	1	1	[0]:2,0,0	[0]:0,0,0	[0]:1,-,-
=== W	143	8	2	0	15	0	[0]:1,2,0	[0]:0,0,0	[0]:1,2,-
=== R	81	5	1	1	1	0	[1]:1,2,0	[1]:0,0,0	[1]:0,1,-
=== W	17	1	0	1	1	1	[1]:2,1,0	[1]:1,0,0	[1]:0,1,-
=== R	133	8	2	0	5	1	[0]:1,2,0	[0]:0,0,0	[0]:1,2,-
=== W	61	3	0	3	13	0	[3]:2,0,0	[3]:0,0,0	[3]:0,-,-
=== R	190	11	2	3	14	0	[3]:1,2,0	[3]:0,0,0	[3]:0,2,-
=== W	211	13	3	1	3	0	[1]:1,0,2	[1]:1,0,0	[1]:0,1,3
=== R	212	13	3	1	4	1	[1]:1,0,2	[1]:1,0,0	[1]:0,1,3
=== W	210	13	3	1	2	1	[1]:1,0,2	[1]:1,0,1	[1]:0,1,3
=== R	115	7	1	3	3	0	[3]:0,1,2	[3]:0,0,0	[3]:0,2,1
=== W	98	6	1	2	2	0	[2]:2,0,0	[2]:0,0,0	[2]:1,-,-
=== R	275	17	4	1	3	0	[1]:0,2,1	[1]:1,0,1	[1]:0,4,3 (out: XM=5 XT=1 XS=1)
=== W	64	4	1	0	0	1	[0]:2,1,0	[0]:1,0,0	[0]:1,2,-
=== R	259	16	4	0	3	0	[0]:1,0,2	[0]:1,0,0	[0]:1,2,4
=== W	130	8	2	0	2	1	[0]:0,2,1	[0]:1,1,0	[0]:1,2,4
=== R	61	3	0	3	13	1	[3]:2,0,1	[3]:0,0,0	[3]:0,2,1
=== W	67	4	1	0	3	1	[0]:2,1,0	[0]:1,1,0	[0]:1,2,4
=== R	70	4	1	0	6	1	[0]:2,1,0	[0]:1,1,0	[0]:1,2,4
=== W	25	1	0	1	9	1	[1]:2,1,0	[1]:1,0,1	[1]:0,4,3



Si ricava quindi che il tempo medio di accesso alla cache e' pari a:
AMAT = thit+tpenalty*m = thit+tpenalty* (Nmiss/Nref) =5+60*11/22 = **35 ns.**

OBIETTIVI

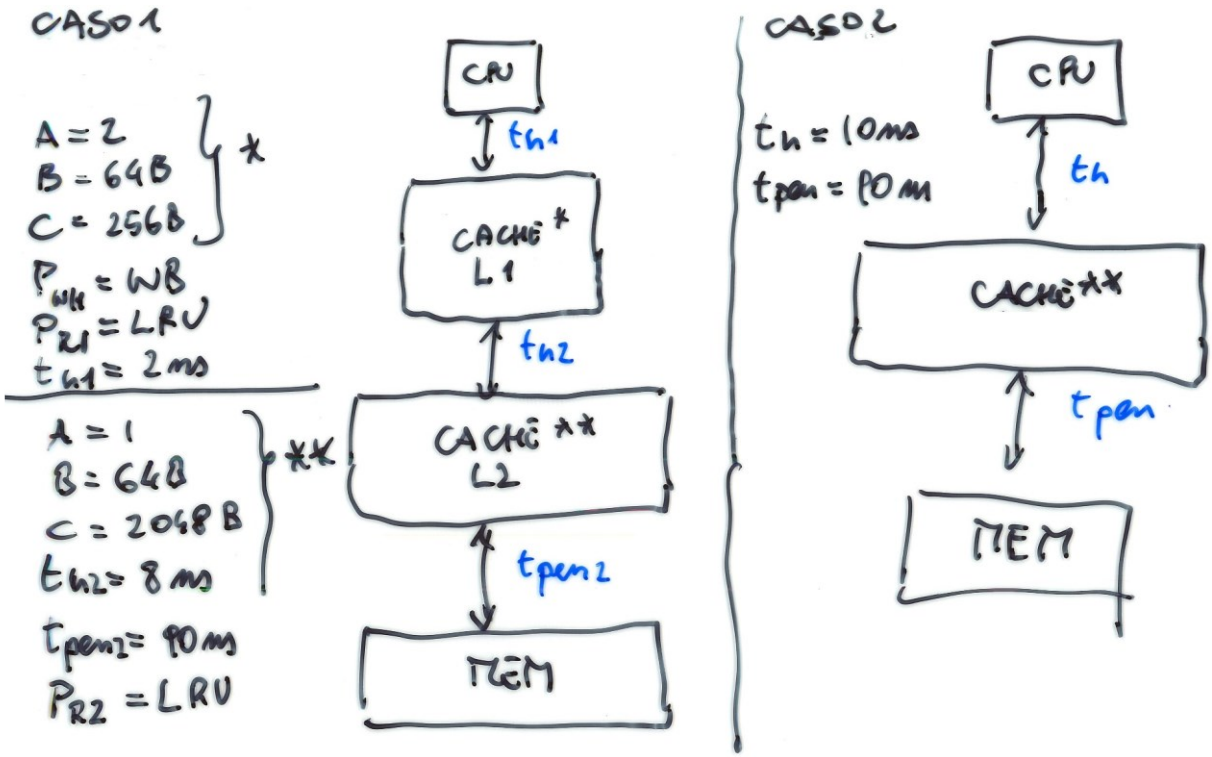
1) **Capire il funzionamento di una cache su 2 livelli (L1+L2)**

1) Capire il funzionamento di una cache su 2 livelli (L1+L2)

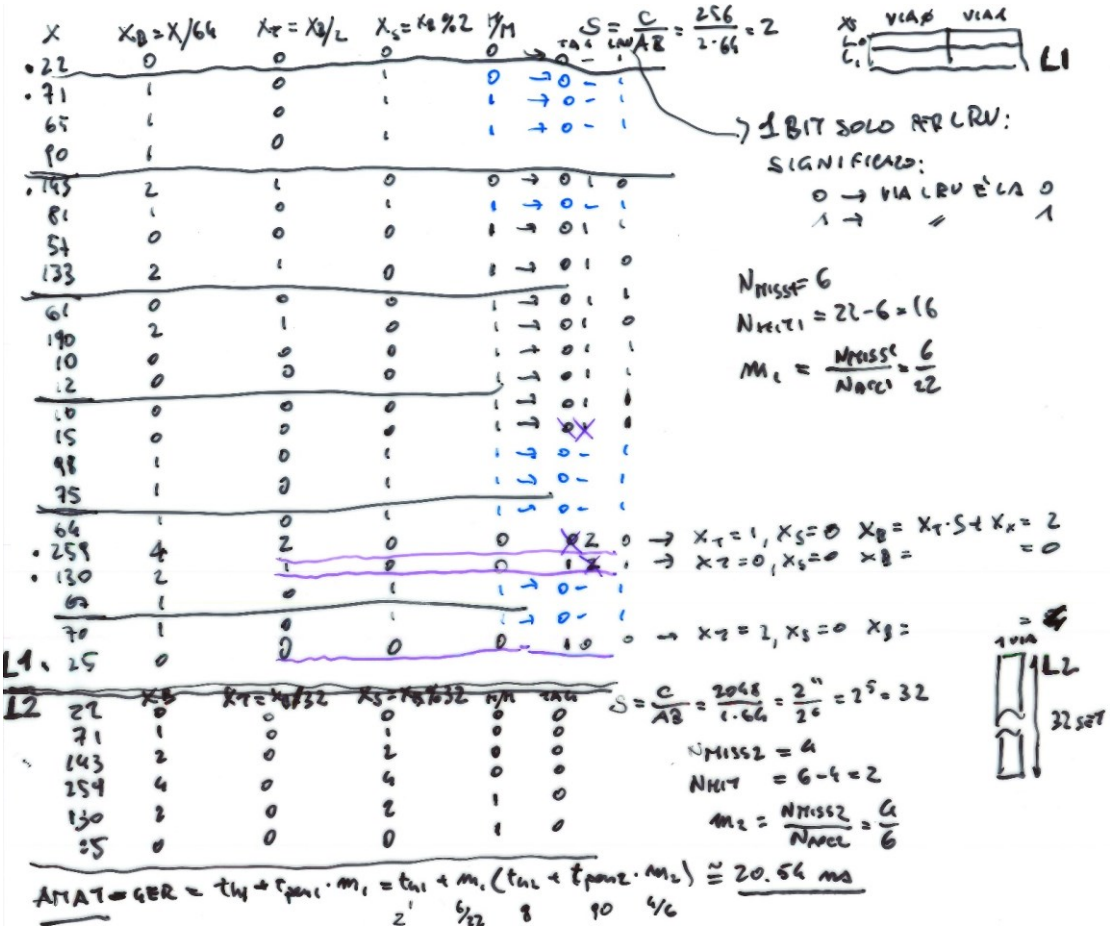
Es. n.2 del 04/12/2007: https://arcal.diism.unisi.it/esercizi1/c1071204-sol.pdf

Si consideri una gerarchia di memoria composta da: a) una cache di primo livello di dimensione 256B, a 2 vie di tipo write-back. La dimensione del blocco e' 64 byte, il tempo di accesso alla cache e' 2 ns, la politica di rimpiazzamento e' LRU; b) una cache di secondo livello di dimensione 2048B, ad accesso diretto di tipo write-back. La dimensione del blocco e' 64 byte, il tempo di accesso alla cache e' 8 ns e la penalita' in caso di miss e' pari a 90 ns, la politica di rimpiazzamento e' LRU; Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 22,71,65,90,143,81,57,133,61,190,10,12,10,15,98,75,64,259,130,67,70,25 Tali accessi sono alternativamente letture e scritture. Per la sequenza data, ricavare il tempo medio di accesso alla gerarchia cosi' composta e confrontarlo con quello che si otterrebbe con la stessa sequenza di accessi usando solo una cache come nel tipo (b), ma con tempo di accesso 10ns.

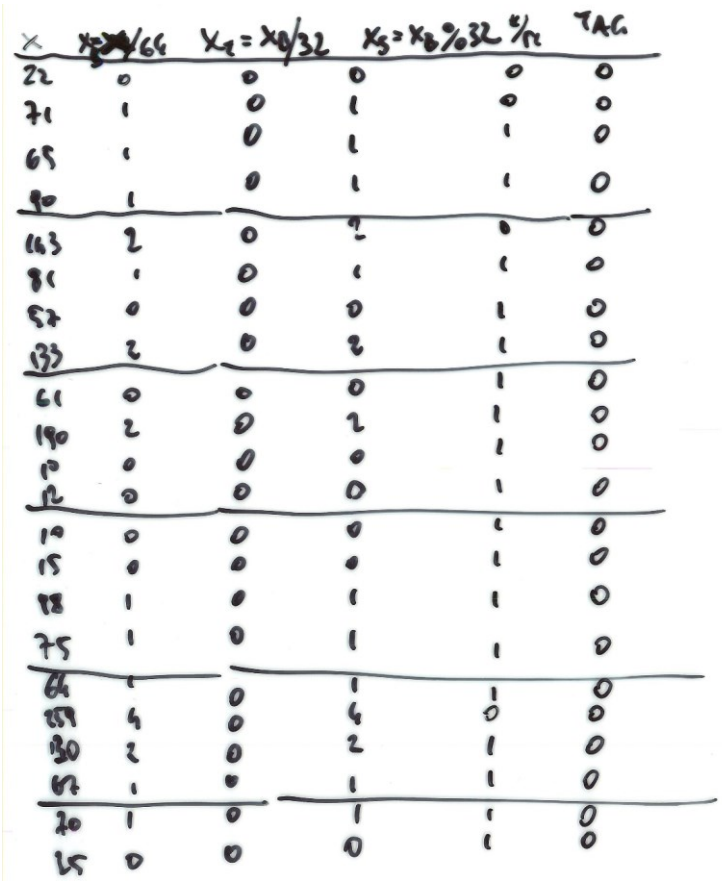
Compito del 04-12-2007 – esercizio 2 svolgimento (1/3)



Compito del 04-12-2007 – esercizio 2 svolgimento (2/3)



Compito del 04-12-2007 – esercizio 2 svolgimento (3/3)



$N_{MISS} = 4$

$N_{HIT} = 22 - 4 = 18$

$m = \frac{N_{MISS}}{N_{ACC}} = \frac{4}{22}$

$AMAT = t_h + m \cdot T_{pen} \cong 26.36$

1 6/22 90

Con $t_h = 8$

$AMAT' = AMAT - 2 \cong 24.36$

ESERCIZI in LABORATORIO – FUNZIONE "FREE" in RISC-V (ESERCITAZIONE/LAB #17) v221214

OBIETTIVI: 1)Funzione Run-Time Library FREE e aggancio con la syscall SBRK; gestione strutture dati 'struct'

(Es. n.1 del 20-11-2018: https://arcal.diism.unisi.it/esercizi1/c1181120-sol.pdf

Trovare il codice assembly RISC-V corrispondente del seguente programma (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), **rispettando le convenzioni di utilizzazione dei registri dell'assembly RISC-V** (riportate in calce, per riferimento).

```
typedef struct header {
   struct header *ptr; unsigned size;
} Header;
static Header base = {NULL,0 };
static Header *freep = NULL;

void myfree(void *ap) {
   Header *bp, *p;  bp = (Header *)ap - 1;
   for (p = freep; !(bp > p && bp < p->ptr); p = p->ptr) {
       if (p >= p->ptr && (bp > p || bp < p->ptr)) break;
   }
   if (bp + bp->size == p->ptr) {
      bp->size += p->ptr->size;
      bp->ptr = p->ptr->ptr;
   } else bp->ptr = p->ptr;
   if (p + p->size == bp) {
      p->size += bp->size;
      p->ptr = bp->ptr;
   } else p->ptr = bp;
   freep = p;
}
```

```
void *alloc_and_print_pun(int sz) {
   void *p = sbrk(sz);
   print_string("p=");
   print_int(p);
   print_string("\n");
   return (p);
}


int main() {
   void *p0, *p1, *p2, *p3;
   base.ptr = &base; freep=&base;
   p0 = alloc_and_print_pun(0);
   p1 = alloc_and_print_pun(256);
   p2 = alloc_and_print_pun(256);
   p3 = alloc_and_print_pun(256);
   myfree(p1); myfree(p2); myfree(p3);
   p0 = alloc_and_print_pun(0);
}
```

RISCV Instructions (RV64IMFD) **v210622**

Instruction coding (hexadecimal) opcode+funct3+{funct7,imm}	Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
33+0+00/3b+0+00	**add**	`add/addw x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
33+0+20/3b+0+20	**subtract**	`sub/subw x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
13+0+imm/1b+0+imm	**add immediate**	`addi/addiw x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
33+0+01/3b+0+01	**multiply**	`mul/mulw x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
33+01+01	**multiply high**	`mulh x5,x6,x7`	x5 ← x6 * x7	Higher 64bits of 128-bits product
33+4+01/3b+4+01	**division**	`div/divw x5,x6,x7`	x5 ← x6/x7	(signed/word) division (divw**)
33+6+01/3b+6+01	**reminder**	`rem/remw x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
33+2+0/33+3+0	**set on less than**	`slt/sltu x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
13+2+imm/13+3+imm	**set on less than immediate**	`slti/sltiu x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (less than)
33+7+0/33+6+0/33+4+0	**and / or / xor**	`and/or/xor x5,x6,x7`	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR
13+7+imm/13+6+imm/13+4+imm	**and /or / xor immediate**	`andi/ori/xori x5,x6,100`	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR register, constant
33+1+0/3b+1+0	**shift left logical**	`sll/sllw x5,x6,x7`	x5 ← x6 << x7	Shift left by register (sllw**)
13+1+imm/1b+1+imm	**shift left logical immediate**	`slli/slliw x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
33+5+0/3b+5+0	**shift right logical**	`srl/srlw x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
13+5+imm/1b+5+imm	**shift right logical immediate**	`srli/srliw x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
33+5+20/3b+5+20	**shift right arithmetic**	`sra/sraw x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
13+5+imm/1b+5+imm	**shift right arithmetic immediate**	`srai/sraiw x5,x6,10`	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
03+3+imm/03+2+imm/03+0+imm	**load dword / word / byte**	`ld/lw/lb x5,100(x6)`	x5 ← MEM[x6+100]	Data from memory to register
03+6+imm/03+4+imm	**load word / byte unsigned**	`lwu/bu x5,100(x6)`	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
23+3+imm/23+2+imm/23+0+imm	**store dword / word / byte**	`sd/sw/sb x5,100(x6)`	MEM[x6+100] ← x5	Data from register to memory (sw**)
37+imm[31:12] (no funct3)	**load upper immediate**	`lui x5,0x12345`	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION	**load address**	`la x5,var`	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	`REAL: lui x5,H20(&var);ori x5, L12(&var)` **INST.** (H20=high 20 bit of &var; L12=low 12 bits of &var)
6f+imm[31:12](rd=0) 63+0+imm[11:0](rs1=rs2=0)	**jump/branch**	`j/b label`	PC+=off (off=PC-&label) **(PS.INST.)**	`REAL INST.: jal x0,offset/beq x0,x0,offset`
6f+0+imm[31:12](rd=1,no funct3)	**jump and link (offset)**	`jal label`	x1←(PC+4);PC+=offset **(PS. INST.)**	`REAL INST.: jal x1,offset` (offset=PC-&label)
67+0+imm (rd=0,rs1=1)	**return from procedure**	`Ret`	PC←x1 **(PSEUDO INST.)**	`REAL INST.: jalr x0,0(x1)`
67+0+imm	**jump and link register**	`jalr x1, 100(x5)`	x1 ← (PC + 4);PC=x5+100	Procedure return; indirect call
63+0+(imm÷2)/63+1+(imm÷2)	**branch on equal / not-equal**	`beq/bne x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
73+0+0 (rs1=0,rs2=0,rd=0)	**ecall**	`Ecall`	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
73+0+8 (rs1=0,rs2=2,rd=0)	**sret**	`Sret`	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION	**move**	`mv x5,x6`	x5 ← x6 **(PSEUDO INST.)**	`REAL INST.: add x5,x0,x6`
PSEUDOINSTRUCTION	**load immediate**	`li x5,100`	x5 ← 100 **(PSEUDO INST.)**	`REAL INST.: addi x5,x0,100`
PSEUDOINSTRUCTION	**no operation (nop)**	`nop`	do nothing **(PSEUDO INST.)**	`REAL INST.: addi x0,x0,0`
53+0+{0,1}/53+0+{4,5}	**floating point add/sub**	`fadd/fsub.{s,d} f0,f1,f2`	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
53+0+{8,9}/53+0+{c,d}	**floating point multiplication/division**	`fmul/fdiv.{s,d} f0,f1,f2`	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION	**floating point move between f-regs**	`fmv.{s,d} f0,f1`	f0←f1 **(PSEUDO INST.)**	`REAL INST.: fsgnj.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION	**floating point negate**	`fneg.{s,d} f0,f1`	f0← – (f1) **(PSEUDO INST.)**	`REAL INST.: fsgnjn.{s,d} f0,f1,f1`
PSEUDOINSTRUCTION	**floating point absolute value**	`fabs.{s,d} f0,f1`	f0← | f1 | **(PSEUDO INST.)**	`REAL INST.: fsgnjx.{s,d} f0,f1,f1`
53+0/1/2+{50,51}	**floating point compare**	`fle/flt/feq.{s,d} x5,f0,f1`	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
53+0+{70,71} (rs2=0)	**move between x (integer) and f regs**	`fmv.x.{s,d} x5,f0`	x5←f0 (no conversion)	Copy (no conversion)
53+0+{78,79} (rs2=0)	**move between f and x regs**	`fmv.{s,d}.x f0,x5`	f0←x5 (no conversion)	Copy (no conversion)
7+2+imm/27+2+imm	**load/store floating point (32bit)**	`flw/fsw f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
7+3+imm/27+3+imm	**load/store floating point (64bit)**	`fld/fsd f0,0(x5)`	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
53+7+21(rs2=0)/53+7+20(rs2=0)	**convert to/from double from/to single**	`fcvt.d.s/fcvt.s.d f0,f1`	f0← (double)f1 / f0← (single)f1	Type conversion
53+7+{60,61} (rs2=0)	**convert to integer from {single,double}**	`fcvt.w.{s,d} x5,f0`	x5← (int)f0	Type conversion
53+7+{68,69} (rs2=0)	**convert to {single,double} from integer**	`fcvt.{s,d}.w f0,x5`	f0← ({single,double})x5	Type conversion
53+0+{2c,2d} (rs2=0)	**square root**	`fsqrt.{s,d} f0,f1`	f0← square root of f1	Single or double square root
53+0/1/2+{10,11}	**sign injection**	`fsgnj/jn/jx.{s,d} f0,f1,f2`	f0←sgn(f2)|f1| /–sgn(f2)|f1| /sgn(f2)|f1|	Extract the mantissa and exp. from f1 and sign from f2

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

```
.data
base:  .word 0
       .word 0
freep: .word 0
RTN:   .asciz "\n"
puneq: .asciz "p="

.globl main
.text
myfree:
#-------------------------------------
# a0=ap,bp   t0=p, sizeof(Header)=8
        addi a0, a0, -8  # bp=ap-8
        la   t1,  freep  # &freep
        lw   t0,  0(t1)  # t1=p=freep
MF_INIFOR1: #scan list of free blocks
        slt  a6, t0, a0  # bp>?p
        lw   t2, 0(t0)   # t2=p->ptr
        slt  a5, a0, t2  # bp<?p->ptr
        and  a4, a5, a6
        bne  a4, x0, MF_FINEFOR1
        slt  a4, t0, t2  # p>=?p->ptr
                         # => !(p<?p->ptr)
        xori a4, a4, -1  #not(.)=> p<?p->ptr
        or   a5, a5, a6  # (... || ...)
        and  a4, a4, a5  # (... && (.))
        bne  a4, x0, MF_FINEFOR1

        add  t0, t2, x0  # p=p->ptr
        j MF_INIFOR1

MF_FINEFOR1:
        lw   t3, 4(a0)   # bp->size
        add  t4, t3, a0  # bp+bp->size
        bne  t4, t2, MF_E1 # (.)!=p->ptr
        lw   t5, 4(t2)   # p->ptr->size
        add  t4, t3, t5  # bp->size+(.)
        sw   t4, 4(a0)   # bp->size=(.)
        lw   t5, 0(t2)   # p->ptr->ptr
        sw   t5, 0(a0)   # bp->ptr=(.)
        j MF_F1

MF_E1:
        sw   t2, 0(a0)   #bp->ptr=p->ptr

MF_F1:
        lw   t6, 4(t0)   # p->size
        add  a4, t6, t0  # p+p->size
        bne  a4, a0, MF_E2 # (.)!=bp
        lw   a5, 4(a0)   # bp->size
        add  t6, t6, a5  # p->size+(.)
        sw   t6, 4(t0)   # p->size=(.)
        lw   t6, 0(a0)   # bp->ptr
        sw   t6, 0(t0)   # p->ptr=(.)
        j MF_F2

MF_E2:
        sw   a0, 0(t0)   # p->ptr=bp

MF_F2:
        sw   t0, 0(t1)   # freep=p
        jr ra
```

```
#-------------------------------------
# a0=sz,  v1=p
alloc_and_print_pun:
        addi a7, x0, 9   # serv.9
        ecall            # sbrk
        add  a1, x0, a0  # salvo in a1
        la   a0, puneq   # stampa msg
        addi a7, x0, 4   # serv.4
        ecall            # print_str
        add  a0, x0, a1  # p
        addi a7, x0, 1   # serv.1
        ecall            # print_int
        la   a0, RTN     # stampa RTN
        addi a7, x0, 4   # serv.4
        ecall            # print_str
        add  a0, a1, x0  # return(p)
        jr ra
#-------------------------------------
main:  #s2=p0, s3=p1, s4=p2, s5=p3
#-------------------------------------
# PROLOGO
        addi sp, sp, -24  # alloco frame
        sw   ra, 20(sp)   # salvo OLD-ra
        sw   s0, 16(sp)   # salvo OLD-fp
        add  s0, x0, sp   # NUOVO fp
        sw   s5, 12(s0)   # salvo s5
        sw   s4, 8(s0)    # salvo s4
        sw   s3, 4(s0)    # salvo s3
        sw   s2, 0(s0)    # salvo s2
        la   t0, base     # &base
        la   t1, freep    # &freep
        sw   t0, 0(t0)    # base.ptr=&base
        sw   t0, 0(t1)    # freep=&base
        add  a0, x0, x0   # sz=0 (HEAPtop)
        jal  alloc_and_print_pun
        addi s2, a0, 0    # salva p0

        addi a0, x0, 256  # sz=256
        jal  alloc_and_print_pun
        addi s3, a0, 0    # salva p1

        addi a0, x0, 256  # sz=256
        jal  alloc_and_print_pun
        addi s4, a0, 0    # salva p2

        addi a0, x0, 256  # sz=256
        jal  alloc_and_print_pun
        addi s5, a0, 0    # salva p3

        add  a0, s3, x0   # p1
        jal  myfree
        add  a0, s4, x0   # p2
        jal  myfree
        add  a0, s5, x0   # p3
        jal  myfree
        add  a0, x0, x0   # sz=0 (HEAPtop)
        jal  alloc_and_print_pun
        add  s2, a0, x0   # salva p0
 #-------------------------------------
# EPILOGO
        lw   s2, 0(s0)    # ripristina s2
        lw   s3, 4(s0)    # ripristina s3
        lw   s4, 8(s0)    # ripristina s4
        lw   s5, 12(s0)   # ripristina s5
        lw   ra, 20(s0)   # ripristina ra
        lw   s0, 16(s0)   # ripristina fp
        addi sp, sp, 24   # DEALLOCO FRAME
        addi a7, x0, 10   # serv.10
        ecall             # exit
```

```
-- program is finished running --

p=268697600
p=268697600
p=268697856
p=268698112
p=268698368

-- program is finished running --
```

OBIETTIVO: **1) Capire l'uso di dati float di una matrice attraverso l'algoritmo iterativo di Gauss-Seidel**

Es. n.2 del 10/02/2016: https://arcal.dii.unisi.it/esercizi1/c1160210-sol.pdf :

Codificare in assembly RISC-V il seguente codice (utilizzando solo e unicamente istruzioni dalla tabella sottostante), rispettando le convenzioni di utilizzazione dei registri dell'assembly RISC-V (riportate in calce). Consegnare 2 files: codice RISC-V e l'output relativo.

```
#define N 3
#define MAXITER 20
#define EPSILON 0.0001

float A[N*N] = {  4.0, -1.0, -1.0,
                 -2.0,  6.0,  1.0,
                 -1.0,  1.0,  7.0 };
float B[N]   = {  3.0,  9.0, -6.0 };
float e2, x[N], y[N], c[N], R[N*N], T[N*N];
int iter = 0;

void setup() {
   int i, j, k;
   float t;
   for (i = 0; i < N; ++i) {
      x[i] = 0.0;
      for (j = 0; j < N; ++j) {
         t = 0.0;
         if (i>j) for (k = j; k < i; ++k)
            t -= A[i*N+k] * T[k*N+j];
         if (i==j) t = 1.0;
         T[i*N+j] = (i<j) ? 0.0 : t / A[i*N+i];
      }
   }
   for (i = 0; i < N; ++i) {
      for (j = 0; j < N; ++j) {
         t = 0.0;
         for (k = 0; k < N; ++k)
            if (k<j) t += T[i*N+k] * A[k*N+j];
         R[i*N+j] = t;
      }
      t = 0.0;
      for (k = 0; k < N; ++k)
         t += B[k] * T[i*N+k];
      c[i] = t;
   }
}

void seidel2(float *x, float *y, float *a, float c) {
   int j;
   *x = c;
   for (j = 0; j < N; ++j) *x -= a[j] * y[j];
}

void report() {
    int i;
    print_string("X: ");
    for (i = 0; i < N; ++i) {
       print_float(x[i]); print_string(" ");
    }
    print_string(" - iter="); print_int(iter);
    print_string("  e2="); print_float(e2);
    print_string("\n");
}

int test_convergence() {
    int j, r;
    ++iter; e2 = 0;
    for (j = 0; j < N; ++j)
       e2 += (y[j] - x[j]) * (y[j] - x[j]);
    r = (e2 < (EPSILON * EPSILON) || iter == MAXITER);
    report();
    return r;

}

void compute() {
    int i;
    do {
      for(i=0;i<N;++i) seidel2(&y[i], x, &R[N*i], c[i]);
      for(i=0;i<N;++i) seidel2(&x[i], y, &R[N*i], c[i]);
    } while (!test_convergence());
}

int main() {
   setup(); compute(); report(); exit(0);
}
```

RISCV Instructions (RV64IMFD) **v210622**

Instruction coding (hexadecimal) opcode+funct3+{funct7,imm}	Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
33+0+00/3b+0+00	**add**	**add/addw x5,x6,x7**	x5 ← x6 + x7	Add two operands; exception possible (addw**)
33+0+20/3b+0+20	**subtract**	**sub/subw x5,x6,x7**	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
13+0+imm/1b+0+imm	**add immediate**	**addi/addiw x5,x6,100**	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
33+0+01/3b+0+01	**multiply**	**mul/mulw x5,x6, x7**	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
33+01+01	**multiply high**	**mulh x5,x6,x7**	x5 ← x6 * x7	Higher 64bits of 128-bits product
33+4+01/3b+4+01	**division**	**div/divw x5,x6,x7**	x5 ← x6/x7	(signed/word) division (divw**)
33+6+01/3b+6+01	**reminder**	**rem/remw x5,x6,x7**	x5 ← x6 % x7	Reminder of the division (remw**)
33+2+0/33+3+0	**set on less than**	**slt/sltu x5,x6,x7**	if (x6 < x7) x5 ← 1; else x5 ← 0	(signed/unsigned) compare x6 and x7 (less than)
13+2+imm/13+3+imm	**set on less than immediate**	**slti/sltiu x5,x6,100**	if (x6 < 100) x5← 1; else x5 ← 0	(signed/unsigned) compare x6 and 100 (less than)
33+7+0/33+6+0/33+4+0	**and / or / xor**	**and/or/xor x5,x6,x7**	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR
13+7+imm/13+6+imm/13+4+imm	**and /or / xor immediate**	**andi/ori/xori x5,x6,100**	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR register, constant
33+1+0/3b+1+0	**shift left logical**	**sll/sllw x5,x6,x7**	x5 ← x6 << x7	Shift left by register (sllw**)
13+1+imm/1b+1+imm	**shift left logical immediate**	**slli/slliw x5,x6,10**	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
33+5+0/3b+5+0	**shift right logical**	**srl/srlw x5,x6,x7**	x5 ← x6 >> x7	Shift right by register (srlw**)
13+5+imm/1b+5+imm	**shift right logical immediate**	**srli/srliw x5,x6,10**	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
33+5+20/3b+5+20	**shift right arithmetic**	**sra/sraw x5,x6,x7**	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
13+5+imm/1b+5+imm	**shift right arithmetic immediate**	**srai/sraiw x5,x6,10**	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
03+3+imm/03+2+imm/03+0+imm	**load dword / word / byte**	**ld/lw/lb x5,100(x6)**	x5 ← MEM[x6+100]	Data from memory to register
03+6+imm/03+4+imm	**load word / byte unsigned**	**lwu/bu x5,100(x6)**	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
23+3+imm/23+2+imm/23+0+imm	**store dword / word / byte**	**sd/sw/sb x5,100(x6)**	MEM[x6+100] ← x5	Data from register to memory (sw**)
37+imm[31:12] (no funct3)	**load upper immediate**	**lui x5,0x12345**	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION	**load address**	**la x5,var**	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	**REAL: lui x5,H20(&var);ori x5, L12(&var) INST.** (H20=high 20 bit of &var; L12=low 12 bits of &var)
6f+imm[31:12](rd=0) 63+0+imm[11:0](rs1=rs2=0)	**jump/branch**	**j/b label**	PC+=off (off=PC-&label) **(PS.INST.)**	**REAL INST.: jal x0,offset/beq x0,x0,offset**
6f+0+imm[31:12](rd=1,no funct3)	**jump and link (offset)**	**jal label**	x1←(PC+4);PC+=offset **(PS. INST.)**	**REAL INST.: jal x1,offset** (offset=PC-&label)
67+0+imm (rd=0,rs1=1)	**return from procedure**	**Ret**	PC←x1 **(PSEUDO INST.)**	**REAL INST.: jalr x0,0(x1)**
67+0+imm	**jump and link register**	**jalr x1, 100(x5)**	x1 ← (PC + 4);PC=x5+100	Procedure return; indirect call
63+0+(imm÷2)/63+1+(imm÷2)	**branch on equal / not-equal**	**beq/bne x5,x6,100**	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
73+0+0 (rs1=0,rs2=0,rd=0)	**ecall**	**Ecall**	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
73+0+8 (rs1=0,rs2=2,rd=0)	**sret**	**Sret**	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION	**move**	**mv x5,x6**	x5 ← x6 **(PSEUDO INST.)**	**REAL INST.: add x5,x0,x6**
PSEUDOINSTRUCTION	**load immediate**	**li x5,100**	x5 ← 100 **(PSEUDO INST.)**	**REAL INST.: addi x5,x0,100**
PSEUDOINSTRUCTION	**no operation (nop)**	**nop**	do nothing **(PSEUDO INST.)**	**REAL INST.: addi x0,x0,0**
53+0+{0,1}/53+0+{4,5}	**floating point add/sub**	**fadd/fsub.{s,d} f0,f1,f2**	f0←f1+f2 / f0←f1-f2	Single or double precision add / subtract
53+0+{8,9}/53+0+{c,d}	**floating point multiplication/division**	**fmul/fdiv.{s,d} f0,f1,f2**	f0←f1*f2 / f0←f1/f2	Single or double precision multiplication / division
PSEUDOINSTRUCTION	**floating point move between f-regs**	**fmv.{s,d} f0,f1**	f0←f1 **(PSEUDO INST.)**	**REAL INST.: fsgnj.{s,d} f0,f1,f1**
PSEUDOINSTRUCTION	**floating point negate**	**fneg.{s,d} f0,f1**	f0← – (f1) **(PSEUDO INST.)**	**REAL INST.: fsgnjn.{s,d} f0,f1,f1**
PSEUDOINSTRUCTION	**floating point absolute value**	**fabs.{s,d} f0,f1**	f0← | f1 | **(PSEUDO INST.)**	**REAL INST.: fsgnjx.{s,d} f0,f1,f1**
53+0/1/2+{50,51}	**floating point compare**	**fle/flt/feq.{s,d} x5,f0,f1**	x5← (f0<f1)	Single and double: compare f0 and f1 <=,<,==
53+0+{70,71} (rs2=0)	**move between x (integer) and f regs**	**fmv.x.{s,d} x5,f0**	x5←f0 (no conversion)	Copy (no conversion)
53+0+{78,79} (rs2=0)	**move between f and x regs**	**fmv.{s,d}.x f0,x5**	f0←x5 (no conversion)	Copy (no conversion)
7+2+imm/27+2+imm	**load/store floating point (32bit)**	**flw/fsw f0,0(x5)**	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
7+3+imm/27+3+imm	**load/store floating point (64bit)**	**fld/fsd f0,0(x5)**	f0←MEM[x5] / MEM[x5]←f0	Data from FP register to memory
53+7+21(rs2=0)/53+7+20(rs2=0)	**convert to/from double from/to single**	**fcvt.d.s/fcvt.s.d f0,f1**	f0← (double)f1 / f0← (single)f1	Type conversion
53+7+{60,61} (rs2=0)	**convert to integer from {single,double}**	**fcvt.w.{s,d} x5,f0**	x5← (int)f0	Type conversion
53+7+{68,69} (rs2=0)	**convert to {single,double} from integer**	**fcvt.{s,d}.w f0,x5**	f0← ({single,double})x5	Type conversion
53+0+{2c,2d} (rs2=0)	**square root**	**fsqrt.{s,d} f0,f1**	f0← square root of f1	Single or double square root
53+0/1/2+{10,11}	**sign injection**	**fsgnj/jn/jx.{s,d} f0,f1,f2**	f0←sgn(f2)|f1|/–sgn(f2)|f1|/sgn(f2)|f1|	Extract the mantissa and exp. from f1 and sign from f2

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

Una possibile soluzione è la seguente:

```
.data
EPSILON:   .float 0.00001
A:         .float  4.0, -1.0, -1.0
           .float -2.0,  6.0,  1.0
           .float -1.0,  1.0,  7.0
B:         .float  3.0,  9.0, -6.0
e2:        .space 4
x:         .space 12
y:         .space 12
c:         .space 12
R:         .space 36
T:         .space 36
iter:      .word 0
x_str:     .asciz "X: "
spazio:    .asciz " "
iter_str:  .asciz " - iter ="
e_str:     .asciz "      e2="
ret:       .asciz "\n"

.text
setup:
#________NO CALL FRAME__________
# t2=i, t3=j, t4=k, ft0=t, t5=i*N

li      t6, 3          # N
la      a4, x          # &x
la      a5, A          # &A
la      a6, T          # &T
li      t2, 0          # t2=i=0

Setup_start_for1:
   slti    t0, t2, 3    # i <? N  (=3)
   beq     t0, x0, Setup_end_for1
   #_____________________________
   mul     t5, t2, t6   # t5=i*N
   slli    t0,t2, 2     # i*sizeof(float)
   add     t1, a4, t0   # &x[i]
   sw      x0, 0(t1)    # x[i]=0.0
   add     t3, x0, x0   # t3=j=0
   Setup_start_for2:
      slti    t0, t3, 3# j <? N  (=3)
      beq     t0, x0, Setup_end_for2
#_____________________________
      fmv.s.x fa0, x0  # t=0

      #if1
      slt     t0, t3, t2     # j <? i
      beq     t0, x0, Setup_fine_if1
      add     t4, t3, x0     # t4=k=j
      Setup_start_for3:
         slt     t0, t4, t2  # k <? i
         beq     t0, x0, Setup_end_for3
#_____________________________
         add     t0, t5, t4  # i*N+k
         slli    t0,t0,2 #*sizeof(float)
         add     t1, a5, t0  # &A[i,k]
         flw     fa2, 0(t1)  # k*N
         mul     t0, t4, t6  # t0=k*N
         add     t0, t0, t3  # k*N+j
         slli    t0,t0,2 #*sizeof(float)
         add     t1, a6, t0  # &T[k,j]
         flw     ft1,0(t1)   #ft1=T[k,j]
         fmul.s  ft3,fa2,ft1#ft3=A[]*T[]
         fsub.s  fa0, fa0,ft3# t=t-(.)
#_____________________________
         addi    t4, t4, 1    # ++k
         j Setup_start_for3
      Setup_end_for3:
      Setup_fine_if1:

      #if2
      bne      t2, t3, Setup_fine_if2
#_____________________________
      li       t0, 1
      fcvt.s.w fa0, t0
#_____________________________
      Setup_fine_if2:
      #if3 (espressione ternaria)
      slt      t0, t2, t3   # i <? j
      beq      t0, x0 Setup_else_if3
#_____________________________
      fmv.s.x fa4, x0       # fa4=0.0
      j Setup_fine_if3
#_____________________________
      Setup_else_if3:
#_____________________________
      add      t0, t5, t2    # t0=i*N+i
      slli     t0,t0, 2 # i*sizeof(float)
      add      t1, a5, t0    # &A[i,k]
      flw      fa2, 0(t1)    # A[i,i]
      fdiv.s   fa4, fa0, fa2
#_____________________________
      Setup_fine_if3:

      add      t0, t5, t3 # t0=i*N+j
      slli     t0,t0, 2  #i*sizeof(float)
      add      t1, a6, t0 # &T[i,j]
      fsw      fa4, 0(t1) # T[i,j] = fa4
#_____________________________
      addi     t3, t3, 1    # ++j
      j Setup_start_for2
   Setup_end_for2:
   #_____________________________
   addi     t2, t2, 1 # ++i
   j Setup_start_for1
Setup_end_for1:

la      a4,  B         # &B
add     t2, x0, x0     # t2=i=0
Setup_start_for4:
   slti    t0, t2, 3    # i <? N(3)
   beq     t0, x0 Setup_end_for4
   #_____________________________
   mul     t5, t2, t6   # t5=i*N
   add     t3, x0, x0   # t3=j=0
   Setup_start_for5:
      slti    t0, t3, 3    # j <? N(3)
      beq     t0, x0 Setup_end_for5
#_____________________________
      fmv.s.x fa0, x0   # t=0
      add     t4, x0, x0    # t4=k=0
      Setup_start_for6:
         slti    t0, t4, 3 # k <? N(3)
         beq     t0, x0 Setup_end_for6
#_____________________________
         #if4
         slt     t0, t4, t3  # k <? j
         beq     t0, x0 Setup_end_if4
#_____________________________
         add     t0, t5, t4  # i*N+k
         slli    t0,t0, 2
#*sizeof(float)
         add     t1, a6, t0  # &T[i,k]
         flw     fa2, 0(t1)  # k*N
         mul     t0, t4, t6
         add     t0, t0, t3  # k*N+j
         slli    t0,t0, 2#*sizeof(float)
         add     t1, a5, t0  # &A[i,k]
         flw     ft1, 0(t1)  # A[i,k]
         fmul.s  ft3,fa2,ft1#ft3=T[]*A[]
         fadd.s  fa0, fa0, ft3# t=t+(.)
#_____________________________
         Setup_end_if4:
#_____________________________
         addi    t4, t4, 1    # ++k
         j Setup_start_for6
      Setup_end_for6:
      add     t0, t5, t3  #t0=i*N+j
      slli    t0,t0,2   #i*sizeof(float)
      la      t1,  R     # &R
      add     t1, t1, t0 # &R[i,j]
      fsw     fa0, 0(t1) # R[i,j]=t
#_____________________________
      addi    t3, t3, 1  # ++j
      j Setup_start_for5
   Setup_end_for5:

   fmv.s.x  fa0, x0      # t=0
   add      t4, x0, x0   # t4=k=0
   Setup_start_for7:
      slti     t0, t4, 3 # k <? N(3)
      beq      t0, x0 Setup_end_for7
#_____________________________
      slli     t0,t4,2  #k*sizeof(float)
      add      t1, a4, t0  # &B[k]
      flw      fa2, 0(t1)  # f2=B[k]
      add      t0, t5, t4  # i*N+k
      slli     t0,t0, 2 # *sizeof(float)
      add      t1, a6, t0  # &T[i,k]
      flw      ft1, 0(t1)  # ft1=T[i,k]
      fmul.s   ft3,fa2,ft1 # ft3=B[]*T[]
      fadd.s   fa0, fa0, ft3# ++k
#_____________________________
      addi     t4, t4, 1
      j Setup_start_for7
   Setup_end_for7:
   slli     t0,t2, 2    #
i*sizeof(float)
   la       t1,  c      # &c
   add      t1, t1, t0  # &c[i]
   fsw      fa0, 0(t1)
   #_____________________________
   addi     t2, t2, 1
   j Setup_start_for4
Setup_end_for4:
ret

seidel2:
#________NO CALL FRAME__________
# INPUT: a0=*x,a1=*y,a2=*a,fa0=c,t0=j
fsw     fa0, 0(a0)    # store c into *x

li      t0, 0         # t0=j=0
Seidel2_start_for1:
   slti    a6, t0, 3    # j<?N  (=3)
   beq     a6, x0 Seidel2_end_for1
   #_____________________________
   slli    t3, t0, 2    #j*sizeof(float)
   add     t4, t3, a2   # &a[j]
   flw     ft0, 0(t4)   # load a[j]
   add     t4, t3, a1   # &y[j]
   flw     ft1, 0(t4)   # load y[j]
   fmul.s  ft2, ft0, ft1# (a[j]*y[j])
   flw     ft3, 0(a0)   # load *x
   fsub.s  ft3, ft3, ft2# [(*x)-(.)]
   fsw     ft3, 0(a0)   # store [.]
   #_____________________________
   addi    t0, t0, 1    # ++j
   j Seidel2_start_for1
Seidel2_end_for1:
ret

report:
la      a0,  x_str    # print "X: "
li      a7,4          # print_string
ecall

li      t0, 0         # s0=i=0
Report_start_for1:
   slti    t1, t0, 3    # i <? N
   beq     t1, x0 Report_end_for1
   #_____________________________
   slli    t3, t0, 2    # i*sizof(float)
   la      t4, x        # t4=&x
   add     t4, t4, t3   # t4= x[i]
   flw     fa0, 0(t4)   # load x[i]
   addi    a7, x0, 2    # print_float
   ecall
   la      a0, spazio   # & " "
   li      a7, 4        #print_string
   ecall
   #_____________________________
   addi    t0, t0, 1    # ++i
   j Report_start_for1
Report_end_for1:

la      a0,  iter_str   # & " - iter="
li      a7,4            # print_string
ecall
la      t0,  iter       # &iter
lw      a0, 0(t0)       # load iter
li      a7,1            # print_int
ecall
la      a0,  e_str      # & " e2="
li      a7,4            # print_string
ecall
la      t0,  e2         # &e2
flw     fa0, 0(t0)      # load e2
li      a7,2            # print_float
ecall
la      a0,  ret        # & "\n"
li      a7,4            # print_string
ecall
ret

test_convergence:
#________CALL FRAME__________
# ra                        4B
# saved variables: s0       4B
#___________________Totale 8B
# OUTPUT: s0=r
addi    sp, sp, -8
sw      ra, 0(sp)
sw      s0, 4(sp)      # r

la      t0, iter       # &iter
lw      t1, 0(t0)      # load iter
addi    t1, t1, 1      # ++iter
sw      t1, 0(t0)      # store iter
la      t0, e2         # &e2
sw      x0, 0(t0)      # store e2=0

li    t0, 0            # t0=j=0
Test_Converg_start_for:
   slti    a6, t0, 3  # j <? N  (=3)
   beq     a6, x0,   Test_Converg_end_for
   #_____________________________
   la      t2,  x       # &x
   la      t3,  y       # &y
   slli    t5, t0, 2    # j*sizeof(float)
   add     t2, t2, t5   # &x[j]
   flw     ft0, 0(t2)   # load x[j]
   add     t3, t3, t5   # &y[j]
   flw     ft1, 0(t3)   # load y[j]
   fsub.s ft1, ft1, ft0# (y[]-x[])
   fmul.s ft1, ft1, ft1# [(.)*(.)]
   la      t6,  e2      # &e2
   flw     ft2, 0(t6)   # load e2
   fadd.s ft2, ft2, ft1# e2+[.]
   fsw     ft2, 0(t6)   # store e2
   #_____________________________
   addi    t0, t0, 1    # +=j
   j Test_Converg_start_for
Test_Converg_end_for:

la      t6,  e2       # &e2
flw     ft2, 0(t6)    # load  e2
la      t0,  EPSILON  # & EPSILON
flw     ft1, 0(t0)    # load EPSILON
fmul.s ft1, ft1, ft1 # EPSILON*EPSILON
li      s0, 0         # r=0
flt.s  t0,ft2,ft1     # e2 < eps
beq     t0, x0, Test_Converg_c1_f
   #_____________________________
   li      s0, 1       # r=1
   j Test_Converg_c2_end
   #_____________________________
Test_Converg_c1_f:
la      t0,  iter      # & iter
lw      t0, 0(t0)      # load iter
slti  t2, t0, 20       # iter==?MAXITER
(=20)
bne     t2, x0 Test_Converg_c2_end
   #_____________________________
   li     s0, 1        # r=1
   #_____________________________
Test_Converg_c2_end:

jal report

mv      a0,s0          # a0=r=s0
lw      ra, 0(sp)
lw      s0, 4(sp)
addi    sp, sp, 8
ret

compute:
#________CALL FRAME__________
# ra                        4B
# saved variables: s0       4B
#___________________Totale 8B
addi    sp, sp, -8
sw      ra, 0(sp)
sw      s0, 4(sp)

Compute_start_do:
   li     s0, 0        # s0=i=0
   Compute_start_for_1:
   slti    t1, s0, 3  # i <? N
   beq     t1, x0 Compute_end_for_1
      #_____________________________
      slli    t3, s0, 2 #i*sizeof(float)
      la      a0, y     # &y
      add     a0, a0, t3# a0=&y[i]
      la      a1, x     # &x
      li      t1, 3     # t1=N  (=3)
      mul    t0,t1,t3#(N*i*sizeof(float))
      la      a2,  R    # &R
      add     a2, a2, t0# &R[(.)]
      la      t4,  c    # t4=&c
      add     t4, t4, t3# t4=&c[i]
      flw     fa0, 0(t4)# load c[i]
      jal seidel2
      #_____________________________
      addi    s0, s0, 1 # ++i
      j Compute_start_for_1
   Compute_end_for_1:

   li     s0, 0        # s0=i=0
   Compute_start_for_2:
   slti    t1, s0, 3  # i <? N
   beq     t1, x0 Compute_end_for_2
      #_____________________________
      slli    t3, s0, 2 #i*sizeof(float)
      la      a0, x     # &x
      add     a0, a0, t3# a0=&y[i]
      la      a1, y     # &y
      addi    t1, x0, 3 # t1=N  (=3)
      mul    t0,t3,t1#(N*i*sizeof(float))
      la      a2,  R    # &R
      add     a2, a2, t0# &R[(.)]
      la      t4,  c    # t4=&c
      add     t4, t4, t3# t4=&c[i]
      flw     fa0, 0(t4)# load c[i]
      jal seidel2
      #_____________________________
      addi    s0, s0, 1 # ++i
      j Compute_start_for_2
   Compute_end_for_2:
   jal test_convergence # Output:a0
   beq a0, x0 Compute_start_do
Compute_end_do:

lw      ra, 0(sp)
lw      s0, 4(sp)
addi    sp, sp, 8
ret

.globl main
main:
#________NO CALL FRAME__________
jal     setup
jal     compute
jal     report
addi    a7,x0,10
ecall
```

Test del programma

Come al solito: scaricare il simulatore RARS (E' UN FILE JAR);2) Scaricare il programma: https://arcal.diism.unisi.it/esercizi1/c1160210-pro1_riscv.s 3) Selezionare dal MENU Settings/Initialize_Program_Counter_to_global_'main'_if_defined; 4) Cliccare sull'icona del simulatore RARS e poi MENU: File/Open → c1160210-pro1_riscv.s; 5) Selezionare dal MENU Assemble (F3) e poi Go (F5); 6) Verificare il risultato:

Run I/O

```
X: 0.9375 1.9791667 -1.0059525  - iter =1      e2=0.08770908
X: 0.99949515 1.9999616 -1.0000668  - iter =2      e2=4.0286675E-5
X: 0.9999995 2.0000005 -1.0000002  - iter =3      e2=6.8213524E-10
X: 1.0 2.0 -1.0000001  - iter =4      e2=0.0
X: 1.0 2.0 -1.0000001  - iter =4      e2=0.0
```

ESERCIZI in LABORATORIO – ESERCIZI DI PREPARAZIONE ALL'ESAME: RISC-V (ESERCITAZIONE/LAB #19)

OBIETTIVI

1) **Esercizio RISC-V per realizzare la moltiplicazione di operandi a 64-bit utilizzando istruzioni che operano su 32-bit**

1) realizzare la moltiplicazione di operandi a 64-bit utilizzando istruzioni che operano su 32-bit

Variante dell'es. n.1 del 22/04/2016: https://arcal.dii.unisi.it/esercizi1/c1160422-sol.pdf

i) Scrivere in assembly RISC-V una funzione che svolga la moltiplicazione di due interi con segno a 64 bit per ottenere un risultato nei 64 bit meno significativi dei previsti 128 bit del prodotto. Si assuma che i registri macchina contengano operandi a 32-bit (si DEVE impostare tale opzione nel simulatore). Gli operandi a 64 bit saranno quindi composti dalla coppia di due registri a 32 bit: a1 (parte alta) e a0 (parte bassa) in modo che (a1:a0) rappresenti uno degli operandi a 64 bit; similmente per l'operando (a3:a2). Il risultato (64 bit meno significativi del prodotto) deve trovarsi in (a1:a0); ii) scrivere inoltre un chiamante che stampi su schermo in esadecimale il prodotto di 0x12345678 per se stesso e il prodotto di tale numero per il suo negato (in complemento a due). NB: usare solo istruzioni della tabella sottostante.

RISCV Instructions (RV64IMFD) **v221117**

Instruction coding (hexadecimal)			Instruction	Example	Register operation	Meaning (** instructions available only in RV64, i.e. 64-bit case)
funct7/imm	funct3	opcode				
00	0	33/3b	**add**	`add/addw x5,x6,x7`	x5 ← x6 + x7	Add two operands; exception possible (addw**)
20	0	33/3b	**subtract**	`sub/subw x5,x6,x7`	x5 ← x6 – x7	Subtracts two operands; exception possible (subw**)
imm	0	13/1b	**add immediate**	`addi/addiw x5,x6,100`	x5 ← x6 + 100	Add a constant ; exception possible (addiw**)
01	0	33/3b	**multiply**	`mul/mulw x5,x6, x7`	x5 ← x6 * x7	(signed/word) Lower 64 bits of 128-bits product (mulw**)
01	1	33	**multiply high**	`mulh x5,x6,x7`	x5 ← x6 * x7	Higher 64bits of 128-bits product
01	4	33/3b	**division**	`div/divw x5,x6,x7`	x5 ← x6/x7	(signed/word) division (divw**)
01	6	33/3b	**reminder**	`rem/remw x5,x6,x7`	x5 ← x6 % x7	Reminder of the division (remw**)
00	2/3	33	**set on less than**	`slt/sltu x5,x6,x7`	if (x6 < x7) x5 ← 1; else x5 ← 0	signed compare x6 and x7 (less than)
imm	2/3	13	**set on less than immediate**	`slti/sltiu x5,x6,100`	if (x6 < 100) x5← 1; else x5 ← 0	unsigned compare x6 and 100 (less than)
00	7/6/4	33	**and / or / xor**	`and/or/xor x5,x6,x7`	x5← x6&x7 / x6|x7 / x6^ x7	Logical AND/OR/XOR register operand
imm	7/6/4	13	**and /or / xor immediate**	`andi/ori/xori x5,x6,100`	x5 ← x6&100 / x6|100 / x6^100	Logical AND/OR/XOR constant operand
0	1	33/3b	**shift left logical**	`sll/sllw x5,x6,x7`	x5 ← x6 << x7	Shift left by register (sllw**)
imm	1	13/1b	**shift left logical immediate**	`slli/slliw x5,x6,10`	x5 ← x6 << 10	Shift left by the immediate value (slliw**)
0	5	33/3b	**shift right logical**	`srl/srlw x5,x6,x7`	x5 ← x6 >> x7	Shift right by register (srlw**)
imm	5	13/1b	**shift right logical immediate**	`srli/srliw x5,x6,10`	x5 ← x6 >> 10	Shift left by immediate value (srliw**)
20	5	33/3b	**shift right arithmetic**	`sra/sraw x5,x6,x7`	x5 ← x6 >> x7 (arith.)	Shift right by register (sign is preserved) (sraw**)
imm	5	13/1b	**shift right arithmetic immediate**	`srai/sraiw x5,x6,10`	x5 ← x6 >> 10 (arith.)	Shift right by immediate value (sraiw**)
imm	3/2/0	03	**load dword / word / byte**	`ld/lw/lb x5,100(x6)`	x5 ← MEM[x6+100]	Data from memory to register
imm	6/4	03	**load word / byte unsigned**	`lwu/lbu x5,100(x6)`	x5 ← MEM[x6+100]	Data from mem. To reg.; no sign extension (lwu**)
imm	3/2	23	**store dword / word / byte**	`sd/sw/sb x5,100(x6)`	MEM[x6+100] ← x5	Data from register to memory (sw**)
imm[31:12]	-	37	**load upper immediate**	`lui x5,0x12345`	x5 ← 0x1234'5000	Load most significant 20 bits
PSEUDOINSTRUCTION			**load address**	`la x5,var`	x5 ← &var **(PSEUDO INST.)** load address of 'var' in x5	`REAL: lui x5,H20(&var);ori x5, L12(&var)` `INST.` (H20=high 20 bits of &var; L12=low 12 bits of &var)
imm[31:12] (rd=0) imm[11:0] (rs1=rs2=0)	- 0	6f/63	**jump/branch**	`j/b label`	PC+=off (off=PC-&label) **(PS.INST.)**	`REAL INST.: jal x0,offset/beq x0,x0,offset`
imm[31:12] (rd=1)	-	6f	**jump and link (offset)**	`jal label`	x1←(PC+4); PC+=offset **(PS. INST.)**	`REAL INST.: jal x1,offset` (offset=PC-&label)
Imm (rd=0,rs=1)	0	67	**return from procedure**	`ret`	PC←x1 **(PSEUDO INST.)**	`REAL INST.: jalr x0,0(x1)`
imm	0	67	**jump and link register**	`jalr x1, 100(x5)`	x1 ← (PC + 4); PC=x5+100	Procedure return; indirect call
imm÷2	0/1	63	**branch on equal / not-equal**	`beq/bne x5,x6,100`	if (x5 = =/!= x6) PC=PC+100	Equal / Not-equal test; PC relative branch
00 (rs1=0,rs2=0,rd=0)	0	73	**ecall**	`ecall`	SEPC←PC;PC←STVEC;save PL/IE;PL=1;IE=0	Call OS (service number in a7); PL= privilege lev; IE=int.en.
08 (rs1=0,rs2=2,rd=0)	0	73	**sret**	`sret`	PC←SEPC; restore PL/IE	Exit supervisor mode; PL= privilege lev; IE=int.en.
PSEUDOINSTRUCTION			**move**	`mv x5,x6`	x5 ← x6 **(PSEUDO INST.)**	`REAL INST.: add x5,x0,x6`
PSEUDOINSTRUCTION			**load immediate**	`li x5,100`	x5 ← 100 **(PSEUDO INST.)**	`REAL INST.: addi x5,x0,100`
PSEUDOINSTRUCTION			**no operation (nop)**	`nop`	do nothing **(PSEUDO INST.)**	`REAL INST.: addi x0,x0,0`

Register Usage

Register	ABI Name	Usage
x10-x11	**a0-a1**	arguments and results
x9, x18-x27	**s1, s2-s11**	Saved
x5-7, x28-x31	**t0-t2, t3-t6**	Temporaries
x12-x17	**a2-a7**	Arguments

Register	ABI Name	Usage
x0	**zero**	The constant value 0
x8, x2	**s0/fp, sp**	frame pointer, stack pointer
x1, x3	**ra, gp**	return address, global pointer
x4	**tp**	thread pointer

Register	ABI Name	Usage
f10-f11	**fa0-fa1**	Argument and Return values
f8-f9, f18-f27	**fs0-fs1, fs2-fs11**	Saved registers
f0 – f7, f28-f31	**ft0-ft7, ft8-ft11**	Temporaries registers
f12-17	**fa2-fa7**	Function arguments

System calls

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Args
print_int	1	a0=integer to print	---
print_float	2	fa0=float to print	---
print_double	3	fa0=double to print	---
print_string	4	a0=address of ASCIIZ string to print	---
read_int	5	---	a0=integer

Service Name	Serv.No.(a7)	INPUT Arguments	OUTPUT Arguments
read_float	6	---	fa0=float
read_double	7	---	fa0=double
read_string	8	a0=address of input buffer, a1=max chars to read	---
sbrk	9	a0=Number of bytes to be allocated	a0=pointer to allocated memory
exit	10	---	---

```
.data
n0: .word 0x12345678
n1: .word 0x00000000
newline: .asciz "\n"
m64:    .asciz "mul64:\n"
buf:    .space 34

.text
.globl  main

changesign:
   xori   a1, a1, -1 # not(a1)
   xori   a0, a0, -1 # not(a0)
   addi   a0, a0, 1  # a0+1
   sltu   t0, x0, a0 # still!=0 --> no overflow
   bne    t0, x0,cs_end
   addi   a1, a1, 1  # add carry
cs_end:
   ret

# multiply a3:a2 by a1:a0
mul64:
   addi  sp, sp, -4  # space for ra
   sw    ra, 0(sp)
   mul   t0, a0, a2  # lo(a0*a2)
   mulhu t1, a0, a2  # hi(a0*a2)
   mul   t2, a1, a2  # lo(a1*a2)
   add   t3, t1, t2  # hi(a0*a2)+lo(a1*a2)
   mul   t4, a0, a3  # lo(a0*a3)
   add   a1, t3, t4  # hi(a0*a2)+lo(a1*a2)+lo(a0*a3)
   mv       a0, t0  # lo(a0*a2)
   # discarding stuff exceeding 64 bits
   li    a4,16       # print 16 hex digits
   call  hexprint    # print result
   lw    ra, 0(sp)
   addi  sp, sp, 4
   ret
```

```
Run I/O
mul64:
0000000012345678
FFFFFFFFEDCBA988
014B66DC1DF4D840
FEB49923E20B27C0
```

```
# parametric hexprint (up to n=32 digits, n in a4)
# a4=lenght of the hex string
# a3:a2:a1:a0=hex number to print
hexprint:
   la    t0, buf     # buffer address
   add   t1, t0, a4  # ...
   addi  t1, t0, 1   # point ot the end of the buffer
   sb    x0, 0(t1)   # end of string
   la    t1, newline # point to newline character
   lb    t1, 0(t1)   # read newline character
   add   t0, t0, a4  # point to the last char. of buf
   li    t6, 8       # constant 8

loop1:
   sb    t1, 0(t0)   # store character
   addi  a4, a4, -1  # update buffer counter
   addi  t0, t0, -1  # update buffer pointer
   slti  t3, a4, 0
   andi  t1, a0, 0xF # get last hex digit
   srli  a0, a0, 4   # shift right the other digits
   addi  t1, t1, 0x30# transform into a ASCII char
   slti  t2, t1, 0x3A# ...
   bne   t2, x0,step1# if greater than 9
   addi  t1, t1, 7   # add 7 --> A, B, C, D, E, F
step1:
   rem   t4, a4, t6  # check if done with 8 digits
   bne   t4, x0,step2# if so:
   mv    a0, a1      # shift right the other registers
   mv    a1, a2      #
   mv    a2, a3      #
step2:
   beq   t3, x0,loop1# loop on all digits
   addi  a0, t0, 1   # point to the beginning of buf
   li    a7,    4    # print the hex string
   ecall
   ret
```

```
main:
   la    a0, m64
   li    a7, 4
   ecall             # print "mul64:"

   la    s0, n0      # (s1:s0)=(n1:n0)
   lw    s0, 0(s0)
   la    s1, n1
   lw    s1, 0(s1)
   mv    a0, s0      # print 1st operand
   mv    a1, s1
   li    a4, 16      # 64 bit printing
   call  hexprint    # print operand

   mv    a0, s0
   mv    a1, s1
   call  changesign

   mv    s2, a0
   mv    s3, a1      # print 2nd operand
   li    a4, 16      # 64 bit printing
   call  hexprint    # print operand

   mv    a0, s0
   mv    a1, s1
   mv    a2, s0
   mv    a3, s1
   call  mul64       # do 64-bit multiplication
                     # + print result
   mv    a0, s0
   mv    a1, s1
   mv    a2, s2
   mv    a3, s3
   call   mul64      # do 64-bit multiplication
                     # + print result
   li    a7, 10  #exit
   ecall
```