

DA RESTITUIRE INSIEME AGLI ELABORATI e A TUTTI I FOGLI
 → NON USARE FOGLI NON TIMBRATI
 → ANDARE IN BAGNO PRIMA DELL'INIZIO DELLA PROVA
 → NO FOGLI PERSONALI, NO TELEFONI, SMARTPHONE/WATCH, ETC

COGNOME _____

NOME _____

NOTA: dovrà essere consegnato l'elaborato dell'es.1 come file <COGNOME>.s e quelli dell'es. 4 come files <COGNOME>.v e <COGNOME>.png

1) [10/30] Trovare il codice assembly RISC-V corrispondente al seguente micro-benchmark (utilizzando solo e unicamente istruzioni dalla tabella sottostante), rispettando le convenzioni di uso dei registri dell'assembly (riportate qua sotto, per riferimento).

```
#define SAMPLES 10
#define MATSIZE 16
int mat[MATSIZE][MATSIZE];

void transpose() {
    for (int i = 0 ; i < MATSIZE ; i++) {
        for (int j = i+1 ; j < MATSIZE ; j++) {
            int aux = mat[i][j];
            mat[i][j] = mat[j][i];
            mat[j][i] = aux;
        }
    }
}

int main() {
    for (int i = 0 ; i < MATSIZE ; i++)
        for (int j = 0 ; j < MATSIZE ; j++)
            mat[i][j] = i+j;
    for (int k = 0 ; k < SAMPLES ; k++)
        transpose();
    int r = 0;
    for (int i = 0 ; i < MATSIZE ; i++)
        r = r + mat[0][i];
    print_int(r);
}
```

Nota: 'int' è un intero a 64 bit.

RISCV Instructions (RV64IMFD)

v210622

| Instruction coding (hexadecimal) | Instruction | Example | Register operation | Meaning |
|----------------------------------|--|-------------------------------|---|---|
| 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 | sub | 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+0+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 | remainder | rem/remw x5, x6, x7 | x5 ← x6 % x7 | Remainder 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 | sra/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 ← 0x12345000 | Load most significant 20 bits |
| PSEUDOINSTRUCTION | load address | la x5, var | x5 ← &var (PSEUDO INST.) | REAL INST.: lui x5, H20(&var); ori x5, L12(&var) |
| 66+imm[31:12](rd=0) | jump/branch | j/b label | PC←off (off=PC-&label) (PS.INST.) | REAL INST.: jal x0, offset/beq x0, x0, offset |
| 66+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-reg | 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

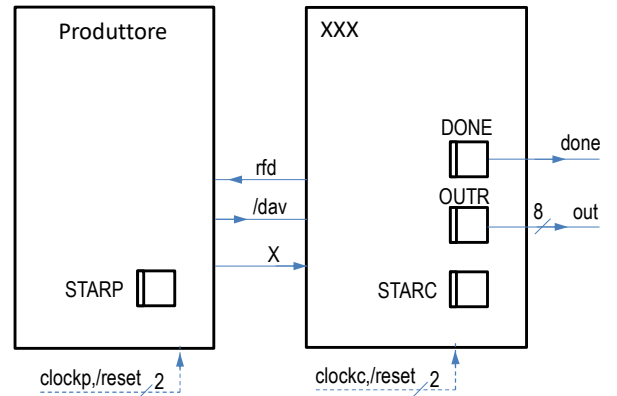
| Service Name | Serv.No.(a7) | INPUT Arguments | OUTPUT Args |
|--------------|--------------|--------------------------------------|-------------|
| print int | 1 | a0=integer to print | --- |
| print float | 2 | fa0=float to print | --- |
| print double | 3 | fa0=double to print | --- |
| print string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

- 2) [5/30] Si consideri una cache di dimensione 32B e a 2 vie di tipo write-back/write-non-allocate. La dimensione del blocco e' 4 byte, il tempo di accesso alla cache e' 4 ns e la penalita' in caso di miss e' pari a 40 ns, la politica di rimpiazzamento e' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 7123, 7339, 7327, 7339, 7328, 7139, 7333, 7354, 7325, 7354, 7322, 7354, 7339, 7126, 7354, 7324, 7354, 7329, 7354, 7328, 7354. 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, i bit di modifica (se presenti) e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento ed inoltre in corrispondenza di quale riferimento il blocco e' eliminato.

- 3) [5/30] Disegnare l'organizzazione fisica di una memoria DRAM da 16Mbit indicando i collegamenti fra i blocchi CTRL, ROW_LATCHES, COL_LATCHES, ROW_DECODER, COL_DECODER, ROW_BUFFERS, PRECHARGE, TRISTATE e spiegare lo svolgimento delle operazioni di accesso ad un singolo bit.

- 4) [10/30] Descrivere e sintetizzare in Verilog il modulo XXX di figura che funziona nel seguente modo: riceve un bit (X) dal modulo produttore col quale colloquia tramite i segnali rfd e /dav; ogni otto bit (Xi) il modulo presenta sull'uscita out un byte (8-bit), indicandone la disponibilita' abilitando il segnale done per 1 ciclo di clock di XXX. Il modulo XXX opera con un clockc di periodo 4ns mentre il modulo Produttore, con clockp, ha un periodo di 2ns: verificare il corretto funzionamento. Il codice del produttore e del testbench e' dato qua sotto. **Tracciare il diagramma di temporizzazione (punti 5/10)** come verifica della correttezza del modulo realizzato.



```

module testbench;
  reg reset_;
  initial begin reset_ = 0; #1 reset_ = 1; #400; $stop; end
  reg clockc;
  initial clockc = 0; always #2 clockc <= (!clockc);
  wire[1:0] STARC = Xxx.STARC;
  wire[7:0] out; wire done, rfd, dav_;
  wire[1:0] X;
  reg clockp;
  initial clockp = 0; always #1 clockp <= (!clockp);
  wire[1:0] STARP = PRO.STARP;
  wire[7:0] QP = PRO.QP, CP = PRO.C, SX = Xxx.S, CX = Xxx.C;
  Xxx(Xxx(dav_, X, clockc, reset_, rfd, out, done));
  Produttore PRO(rfd, clockp, reset_, dav_, X);
endmodule
    
```

```

module Produttore(rfd, clock, reset_, dav_, X);
  input rfd, clock, reset_; output dav_, X;
  reg DAV_; assign dav_ = DAV_;
  reg XP; assign X = XP;
  reg[7:0] QP, C;
  reg[1:0] STAR; parameter S0=0, S1=1, S2=2, Q0=173;
  always @(reset == 0) begin C = 0; STAR <= S0; end
  always @(posedge clock) if (reset == 1) #0.1
    casex (STAR)
      S0: begin if (C == 0) begin C = 8; QP = Q0; XP = QP[7]; end
            DAV_ = 1; STAR <= (rfd == 1) ? S1 : S0; end
      S1: begin XP = QP[7]; QP = QP << 1; C = C - 1;
            DAV_ = 0; STAR <= S2; end
      S2: begin STAR <= (rfd == 1) ? S2 : S0; end
    endcase
endmodule
    
```

