Esercitazione 4: Processore RISC-V in Verilog

- OBIETTIVO: DESCRIVERE IL PROCESSORE RISC-V IN VERILOG e IMPARARE A SCRIVERE IL TESTBENCH PER UN MODULO

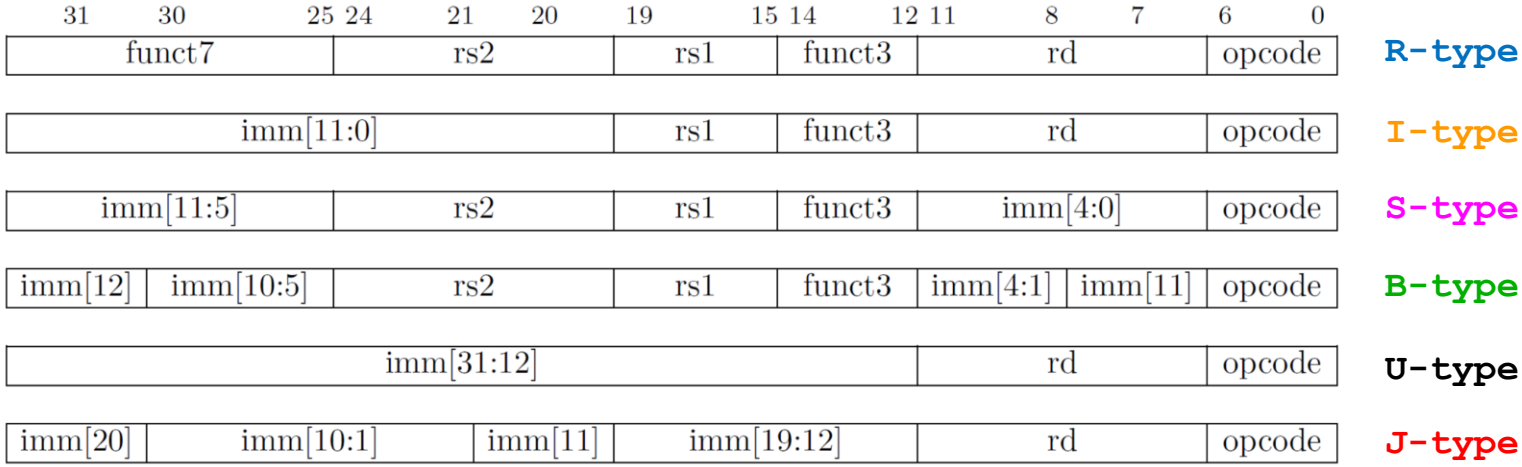
  - Verificare la struttura dei vari moduli che lo compongono in modo da descrivere i corrispondenti moduli del processore RISC-V a 64 bit

  - Ad OGNI GRUPPO è assegnato un modulo:
    il gruppo dovrà scrivere un testbench per fare il testing del modulo assegnato

  - Verificare nel Verilogger
    la correttezza del diagramma temporale di ciascun modulo

  - Possibilmente: mettiamo insieme tutti i moduli per avere una descrizione completa del processore RISC-V

Ricapitolazione dei formati e istruzioni principali



- **Istruzione** **Significato** **Variante** **Significato**

```
add x5,x6,x7    x5 = x6 + x7                addi x5,x6,10     x5 = x6 + 10
sub x5,x6,x7    x5 = x6 - x7
slt x5,x6,x7    x5 = (x6<x7) ? 1 : 0        slti x5,x6,10     x5 = (x6<10) ? 1 : 0

ld  x5,100(x9)  x5 = Memory[x9+100] (64 bit)  lw  x5,100(x9)  (32 bit)
sd  x5,100(x9)  Memory[x9+100] = x5 (64 bit)  sw  x5,100(x9)  (32 bit)

bne x6,x7,L     salta a L se x6!=x7 (L si trova a offset imm13=(&L-PC))
beq x6,x7,L     salta a L se x6==x7 (L si trova a offset imm13=(&L-PC))

lui x5,imm20    carica imm20 nei 20 bit alti

jal FUN         esegue la funzione FUN (che si trova a offset imm21=(&FUN-PC))
ret             ritorna da funzione (jalr x0,0(x1))
```

Programma riscvtest.s → derivare gli opcode = bit 6:0
func7,func3 =bit 31:25,14:12, o immediate,func3=bit 31:20,14:12, o…

	ISTRUZIONE ASSEMBLY		PC	ISTR.MACCHINA	OPCODE	R-type:func7,func3
main:	addi x5, x0, 5	# initialize x5 = 5	0	00500293	→ 13	→ 005,0
	addi x6, x0, 12	# initialize x6 = 12	4	00c00313	→ 13	→ 00c,0
	addi x7, x6, -9	# initialize x7 = 3	8	ff730393	→ 13	→ ff7,0
	add x3, x0, x0	# initialize x3 = 0	c	000001b3	→ 33	→ 00,0
	or x8, x7, x5	# x8 = (3 OR 5) = 7	10	0053e433	→ 33	→ 00,6
	and x9, x6, x8	# x9 = (12 AND 7) = 4	14	008374b3	→ 33	→ 00,7
	add x9, x9, x8	# x9 = 4 + 7 = 11	18	008484b3	→ 33	→ 00,0
	beq x9, x7, end	# shouldn't be taken	1c	02748263	→ 63	→ 012,0
	slt x8, x6, x8	# x8 = 12 < 7 = 0	20	00832433	→ 33	→ 00,2
	beq x8, x0, around	# should be taken	24	00040463	→ 63	→ 004,0
	addi x9, x0, 0	# shouldn't happen	28	00000493	→ 13	→ 000,0
around:	slt x8, x7, x5	# x8 = 3 < 5 = 1	2c	0053a433	→ 33	→ 00,2
	add x7, x8, x9	# x7 = 1 + 11 = 12	30	009403b3	→ 33	→ 00,0
	sub x7, x7, x5	# x7 = 12 - 5 = 7	34	400538b3	→ 33	→ 20,0
	add x9, x3, x6	# x9 = 10008000 + 12	38	006184b3	→ 33	→ 00,0
	sw x7, 68(x9)	# [10008050] = 7	3c	0474a223	→ 23	→ 047,2
end:	lw x5, 80(x3)	# x5 = [10008050] = 7	40	0501a283	→ 03	→ 050,2

I-type:imm, func3

B-type:bit31:25,11:5→imm[12],imm[10:5],imm[4:1],imm[11], func3

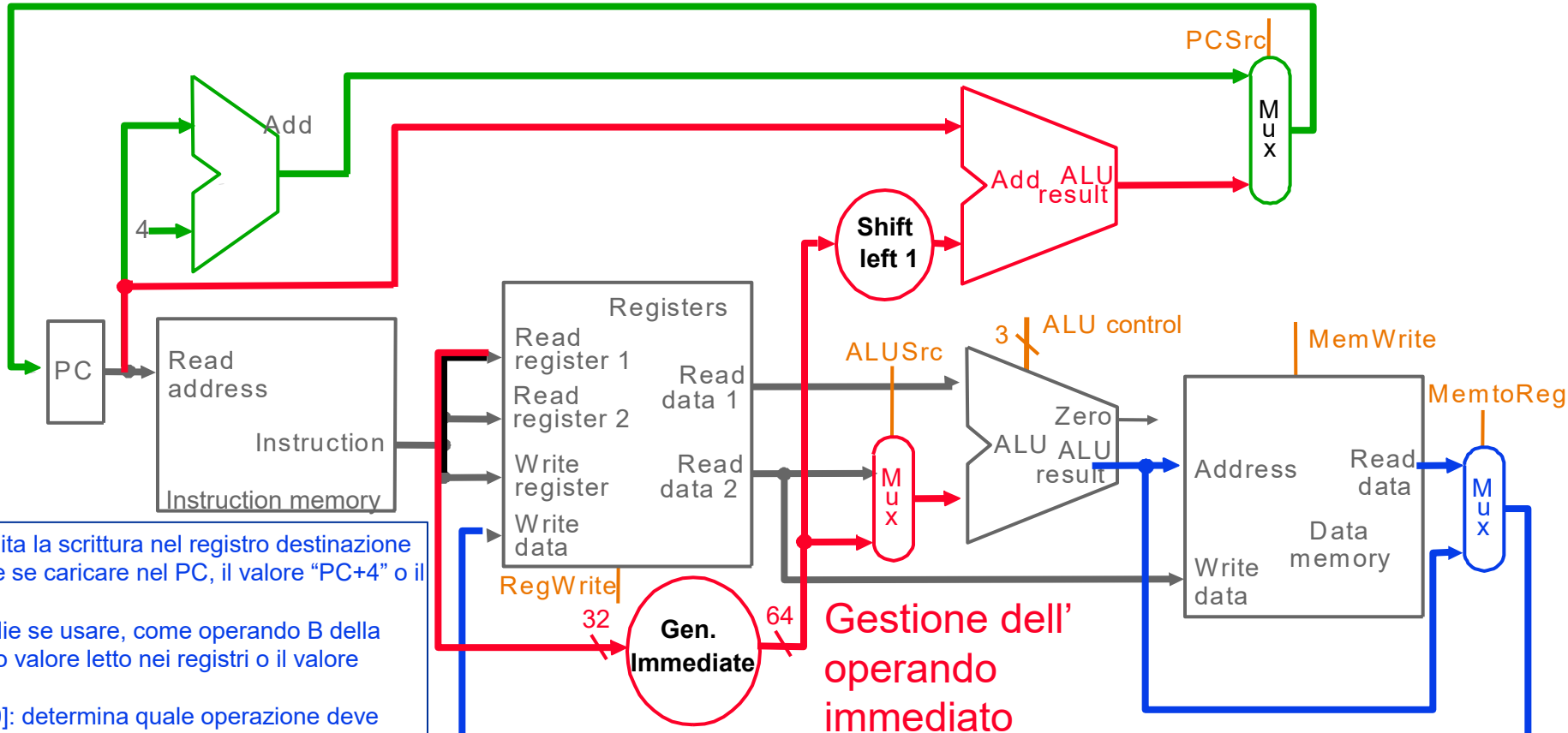
S-type:bit31:25,11:5→imm[11:5],imm[4:0], func3

Costruzione del Datapath del processore RISC-V

- c.f. PHRV1 figura 4.11

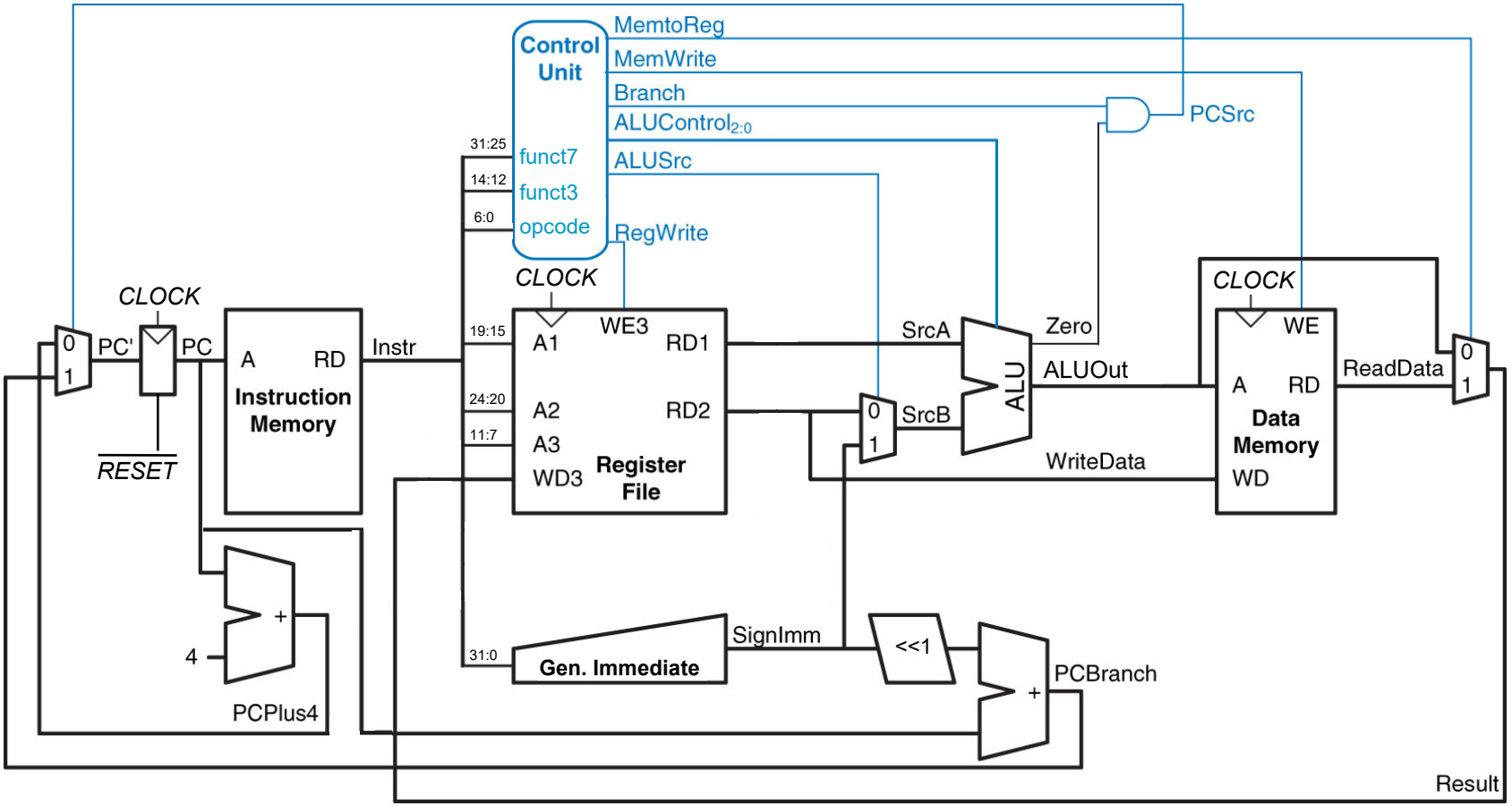
Gestione indirizzo istruzione successiva

I segnali di controllo vengono pilotati in base a “op-code” e “funct” dell’istruzione:



- **RegWrite**: abilita la scrittura nel registro destinazione
- **PCSrc**: sceglie se caricare nel PC, il valore “PC+4” o il target del salto
- **ALUSrc**: sceglie se usare, come operando B della ALU, il secondo valore letto nei registri o il valore immediato
- **AluControl**[2:0]: determina quale operazione deve essere eseguita dalla ALU
- **MemWrite**: abilita la scrittura nella memoria
- **MemtoReg**: sceglie se scrivere nel registro destinazione il valore letto dalla memoria o il risultato prodotto dalla ALU

Datapath del RISC-V (più preciso)



RegWrite
PCSrc
ALUSrc
AluControl[2:0]
MemWrite
MemtoReg

MemRead — Non c'e' rispetto al Patterson

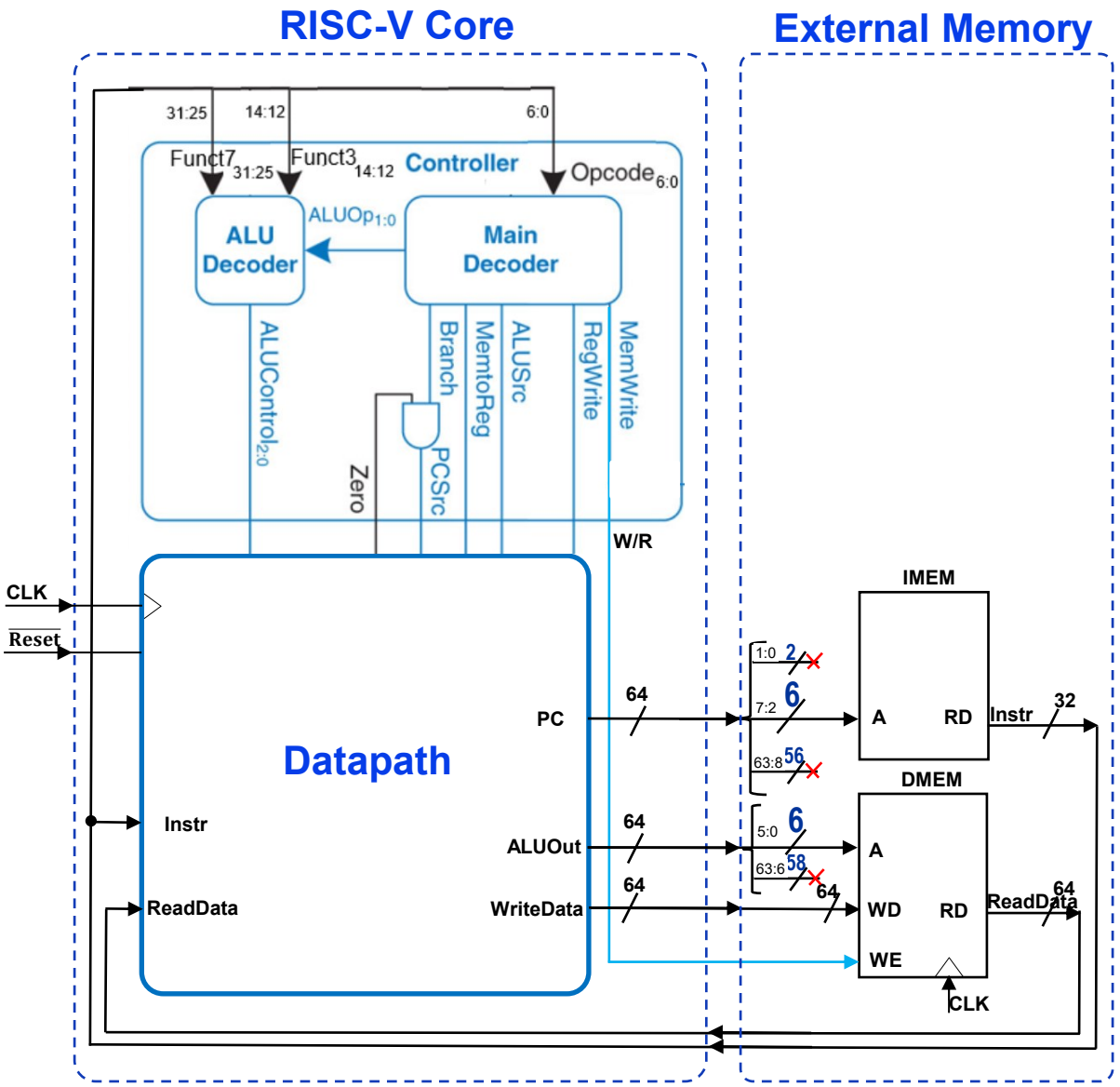
Branch — Nuovo

ALUcontrol — 3 fili anziche' 4

TESTBENCH:

```
`timescale 1ns/1ps
module cpu_testbench;
   reg reset_; initial begin reset_=0; #22 reset_=1; #400; end
   reg clock;  initial clock=0; always #5 clock<=(!clock);
   initial begin wait(reset_); #180 $finish; end
   riscvmem cpu(clock,reset_);
endmodule
```

RISC-V processor



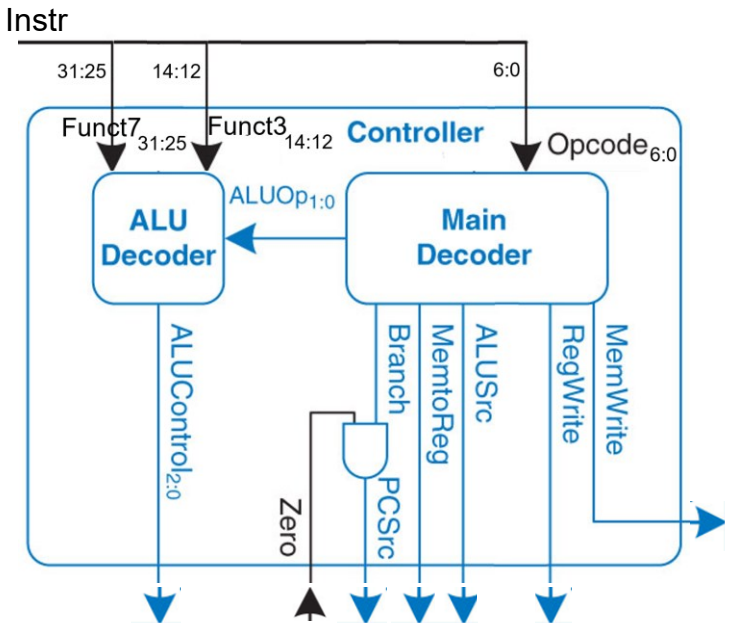

```
module riscvmem(clk,reset_);
   input clk,reset_; wire memwrite;
   wire [63:0] readdata,pc,aluout,writedata;
   wire [31:0] instr;
   // instantiate processor and memories
   rvcore mycore(clk,reset_,readdata,instr,
                 pc,aluout,writedata,memwrite);
   imem imem(pc[7:2],  instr);
   dmem dmem(clk,aluout[5:0],writedata,memwrite,
             readdata);
endmodule
```

```
module rvcore(clk,reset_,readdata,instr,
              pc,aluout,writedata,memwrite);
   input        clk, reset_;
   input[63:0]  readdata;
   input[31:0]  instr;
   output[63:0] pc,aluout,writedata;
   output       memwrite;
   wire         regwrite,alusrc,memtoreg,pcsrc,
                zero;
   wire[2:0]    alucontrol;

   controller c(instr[31:25],instr[14:12],
        instr[6:0],zero,  memwrite,regwrite,
        alusrc,memtoreg,pcsrc,alucontrol);

   datapath dp(clk,reset_,instr,readdata,
        regwrite,alusrc,memtoreg,pcsrc,
        alucontrol,  zero,pc,aluout,writedata);
endmodule
```

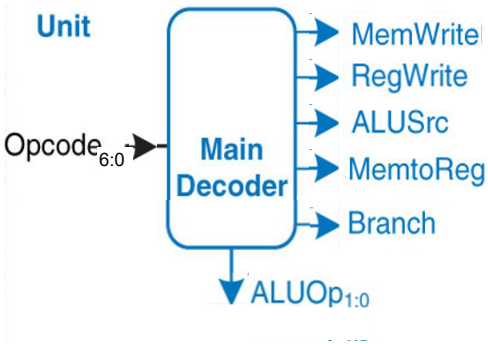
RISC-V Controller



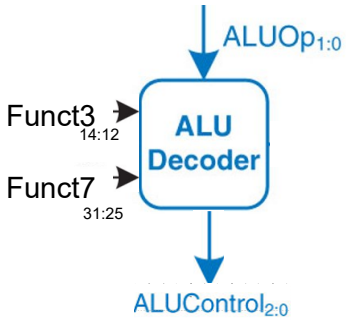

```
module controller(funct7,funct3,opcode,zero,
                  memwrite,regwrite,alusrc,memtoreg,
                  pcsrc,alucontrol);
   input[2:0] funct3;
   input[6:0] opcode,funct7;
   input zero;
   output memwrite,regwrite,alusrc,memtoreg,pcsrc;
   output [2:0] alucontrol;
   wire [1:0] aluop;
   wire branch;

   maindec md(opcode,  memwrite,regwrite,alusrc,
              memtoreg,branch,aluop);
   aludec ad(funct7,funct3,aluop,  alucontrol);
   assign pcsrc = branch & zero;
endmodule
```

RISC-V Main Decoder e RISC-V ALU Decoder



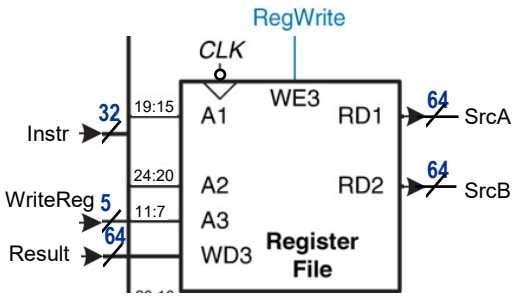

```
module maindec(opcode,  memwrite,regwrite,alusrc,memtoreg,branch,aluop);
   input[6:0] opcode;
   output memwrite,regwrite,alusrc,memtoreg,branch; output[1:0] aluop;reg[7:0]controls;
   assign {regwrite,alusrc,branch,memwrite,memtoreg,aluop} = controls;
   always @(opcode) casex(opcode)          //       (opcode->controls)
      7'b0110011: controls <= 7'b1000010; // R-TYPE  (0x33->0x42)
      7'b0000011: controls <= 7'b1100100; // LW      (0x03->0x64)
      7'b0100011: controls <= 7'b0101000; // SW      (0x23->0x28)
      7'b1100011: controls <= 7'b0010001; // BEQ     (0x63->0x11)
      7'b0010011: controls <= 7'b1100000; // ADDI    (0x13->0x60)
      default:    controls <= 7'bxxxxxxx; // illegal op
   endcase
endmodule
```



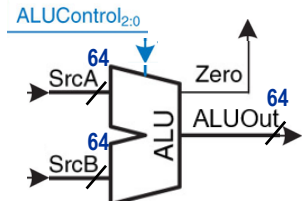
ALUcontrol	Function
0 00 =0	AND
0 01 =1	OR
0 10 =2	ADD
1 10 =6	SUB
1 11 =7	SLT
1 10 =6	EQ

```
module aludec(funct7,funct3,aluop,   alucontrol);
   input[6:0] funct7; input[2:0] funct3; input[1:0] aluop;
   output [2:0] alucontrol; reg[2:0] alucontrol;
   always @(aluop or funct3 or funct7) casex(aluop)        //      (aluop->alucontrol)
      2'b00: alucontrol <= 3'b010;                 //lw/sw/addi          0->2 alu=add
      2'b01: alucontrol <= 3'b110;                 //beq                 1->6 alu=eq
      2'b10: casex(funct3)                         //R-TYPE instructions 2->R-type
            3'b000: casex(funct7)                    //add or sub  (funct3->alucontrol)
               7'b0000000: alucontrol <= 3'b010;   //add  0(funct7=0x00)->2 alu=add
               7'b0100000: alucontrol <= 3'b110;   //sub  0(funct7=0x20)->6 alu=sub
               endcase
            3'b111:  alucontrol <= 3'b000;         //and                 7->0 alu=and
            3'b110:  alucontrol <= 3'b001;         //or                  6->1 alu=or
            3'b010:  alucontrol <= 3'b111;         //slt                 2->7 alu=slt
            default: alucontrol <= 3'bxxx;         //???                 Unknown funct3
         endcase
      default: alucontrol<=3'bxxx;                 //???                 Unknown aluop
   endcase
endmodule
```

RISC-V Registerfile, Adder, ALU

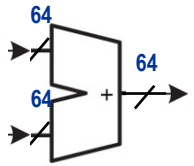



```
module regfile(clk,we3,a1,a2,a3,wd3,   rd1,rd2);
   input clk,we3; input[4:0] a1,a2,a3; input[63:0] wd3;
   output[63:0] rd1,rd2; reg[63:0] rd1,rd2;
   reg[63:0] rf[0:31];
   // three ported register file: read two ports combinationally;
   // write third port on falling edge of clk; register 0 hardwired to 0
   always @(negedge clk) if (we3) rf[a3] <= wd3;
   always @(a1) rd1 <= (a1 != 0) ? rf[a1] : 0;
   always @(a2) rd2 <= (a2 != 0) ? rf[a2] : 0;
endmodule
```



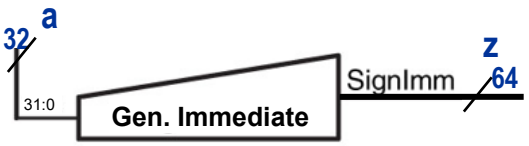
ALUcontrol	Function
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT
1 10	EQ

```
module alu(a,b,aluctrl,    aluout,zero);
   input[63:0] a,b; input[2:0] aluctrl;
   output[63:0] aluout; reg[63:0] aluout; output zero;
   assign zero = (aluout==0);
   always @(aluctrl or a or b) #0.1 //delay to avoid infinite speed loops
      casex (aluctrl)
         0: aluout <= a & b;
         1: aluout <= a | b;
         2: aluout <= a + b;
         6: aluout <= a - b;
         7: aluout <= a<b ? 1:0;
         default: aluout<=0; // should not happen!
      endcase
endmodule
```

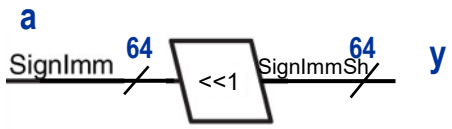



```
module adder(a,b,  aluout);
   input[63:0] a,b;
   output[63:0] aluout;
   assign aluout=a+b;
endmodule
```

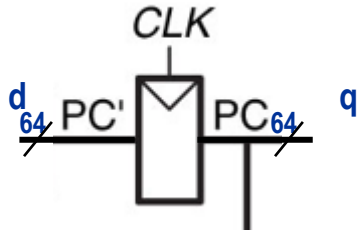
Shift-Left-1, Gen-Immediate, Resettable Flip-flop, MUX2



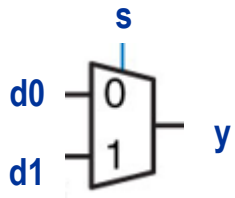

```
module genimm(a,   z);
   input[31:0] a; output[63:0] z; reg[11:0] y; wire[6:0] opc;
   assign opc=a[6:0];
   always @(a or opc) casex(opc)
      7'b00x0011: y <= a[31:20];
      7'b0100011: y <={a[31:25],a[11:7]};
      7'b1100011: y <={a[31],a[7],a[30:25],a[11:8]};
   endcase
   assign z = {{52{y[11]}}, y};
endmodule
```




```
module sl1(a, y); // shift left by 1
  input[63:0] a; output[63:0] y;
  assign y = {a[62:0], 1'b0};
endmodule
```

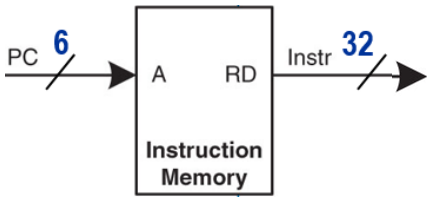



```
module flopr (clk,reset_,d,   q);
   parameter WIDTH = 8;
   input  [WIDTH-1:0] d; input clk, reset_;
   output [WIDTH-1:0] q; reg [WIDTH-1:0] q;
   always @(posedge clk) // reset on posedge
      #1.1 if (!reset_) q <= 0; //delay to avoid infinite speed loops
      else q <= d;
endmodule
```

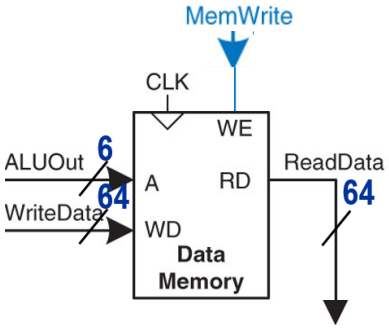



```
module mux2 (d0,d1,s,   y);
   parameter WIDTH = 64;
   input[WIDTH-1:0] d0, d1; input s; output[WIDTH-1:0] y;
   assign y = s ? d1 : d0;
endmodule
```

Instruction Memory and Data Memory

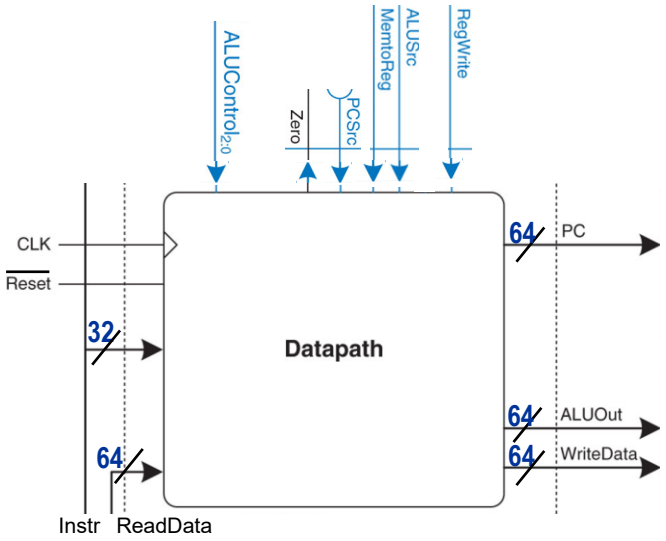


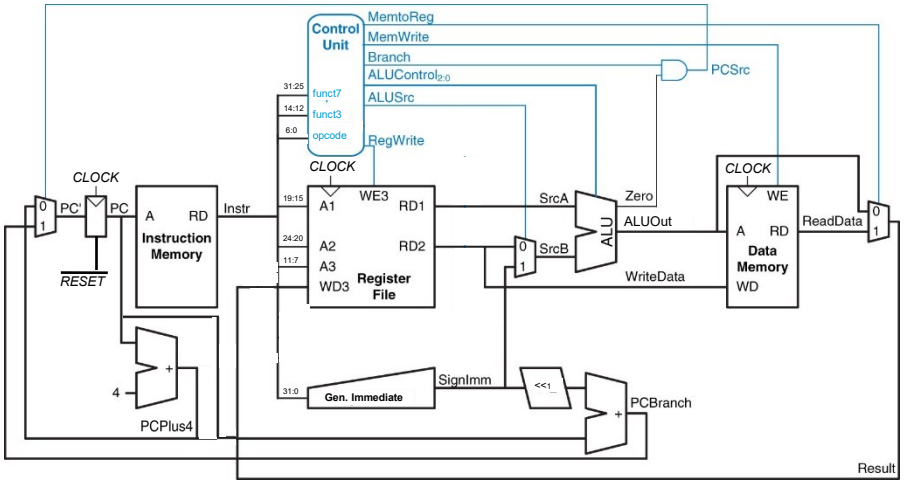

```
module imem(a,  rd);
   input[5:0] a; output[31:0] rd;
   reg[31:0] ROM[0:63];
   initial $readmemh("memfilerv.dat", ROM);
   assign rd = ROM[a]; // word aligned
endmodule
```




```
module dmem(clk,a,wd,we,  rd);
   input clk, we;  input[5:0] a; input[63:0] wd; output[63:0] rd;
   reg[63:0] RAM[0:7];  // 64 bytes (i.e., 8 dwords)
   assign rd = RAM[a[5:3]]; // dword aligned
   always @(posedge clk)
   if (we) RAM[a[5:3]] <= wd;
endmodule
```

RISC-V Datapath






```
module datapath(clk,reset_,instr,readdata,regwrite,
                alusrc,memtoreg,pcsrc,alucontrol,
                zero,pc,aluout,writedata);
   input         clk,reset_;
   input[31:0]   instr;
   input[63:0]   readdata;
   input         regwrite,alusrc,pcsrc,memtoreg;
   input[2:0]    alucontrol;
   output        zero;
   output[63:0] pc,aluout,writedata;
   wire[63:0]    pcnext,pcplus4,pcbranch,signimmsh;
   wire[63:0]    signimm,srca,srcb,result;

   // next PC logic
   flopr #(64)  pcreg(clk,reset_,pcnext,       pc);
   adder        pcadd1(pc,64'b100,             pcplus4);
   sl1          immsh(signimm[63:0],           signimmsh);
   adder        pcadd2(pc,signimmsh,           pcbranch);
   mux2 #(64)   brmux(pcplus4,pcbranch,pcsrc,  pcnext);

   // register file logic
   regfile      rf(clk,regwrite,instr[19:15],instr[24:20],
                   instr[11:7],result,   srca,writedata);
   mux2 #(64)   resmux(aluout,readdata,memtoreg,  result);
   genimm       gimm(instr,   signimm);

   // ALU logic
   mux2 #(64)   srcbmux(writedata,signimm,alusrc,  srcb);
   alu          alu(srca,srcb,alucontrol,   aluout,zero);
endmodule
```

ESERCIZIO (a gruppi – ogni fila un gruppo)

- Costruire il TESTBENCH **SEPARATO** per i seguenti moduli:
- FILA1: CONTROLLER
- FILA2: MAIN-DEC
- FILA3: ALU-DEC
- FILA4: REGFILE
- FILA5: ALU
- FILA6: GEN-IMM + SL1
- FILA7: FLOPR + MUX2
- FILA8: IMEM + DMEM

- TESTBENCH (versione estesa per verificare più segnali):

```
module cpu_testbench;
   reg reset_; initial begin reset_=0; #22 reset_=1; #400; end
   reg clock;  initial clock=0; always #5 clock<=(!clock);
   initial begin wait(reset_); #180 $finish; end
   wire[63:0] PC; wire[11:0] IMM; wire[31:0] IR; wire[6:0]OPCODE, FUNCT7;
   wire[2:0] FUNCT3,ALUCTRL; wire[4:0] RD, RS1, RS2; wire[1:0] ALUOP;
   assign PC=cpu.mycore.dp.pc,IR=cpu.mycore.dp.instr, ALUCTRL=cpu.mycore.dp.alu.aluctrl;
   assign ALUOP=cpu.mycore.c.ad.aluop;
   assign OPCODE=cpu.mycore.c.opcode,FUNCT7=cpu.mycore.c.funct7;
   assign FUNCT3=cpu.mycore.c.funct3,RD=cpu.mycore.dp.rf.a3;
   assign RS1=cpu.mycore.dp.rf.a1,RS2=cpu.mycore.dp.rf.a2;
   wire[63:0] A,B,RESULT; assign A=cpu.mycore.dp.srca,B=cpu.mycore.dp.srcb;
   assign RESULT=cpu.mycore.dp.result, IMM=cpu.mycore.dp.gimm.y;
   initial begin wait(reset_); #180
      $display("X5=%h  (should contain '00000007')",cpu.mycore.dp.rf.rf[5]);
      $finish;
   end
   riscvmem cpu(clock,reset_);
endmodule
```