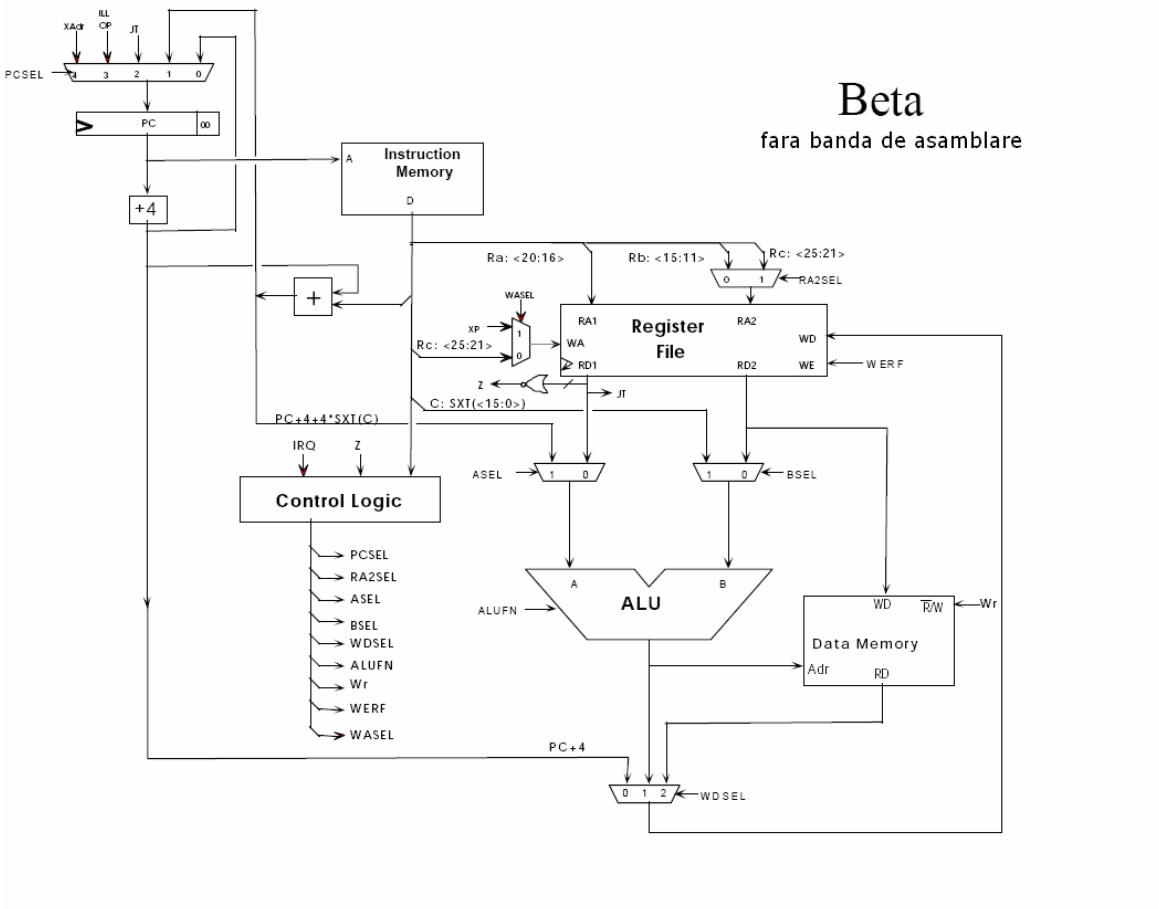
ANEXA la Cursul CN2_2

Procesorul RISC (Beta, fara banda de asamblare) (http://6004.csail.mit.edu/)

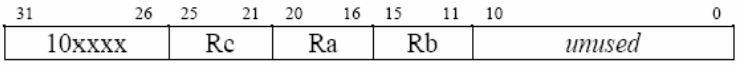


Logica de comand/control:

	OP	*OPC*	*LD*	*ST*	*JMP*	*BEQ*	*BNE*	*LDR*	*Illop*	*trap*
ALUFN	F(op)	F(op)	"+"	"+"	—	—	—	"A"	—	—
WERF	1	1	1	0	1	1	1	1	1	1
BSEL	0	1	1	1	—	—	—	—	—	—
WDSEL	1	1	2	—	0	0	0	2	0	0
WR	0	0	0	1	0	0	0	0	0	0
RA2SEL	0	—	—	1	—	—	—	—	—	—
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	3	4
ASEL	0	0	0	0	—	—	—	1	—	—
WASEL	0	0	0	—	0	0	0	0	1	1

β Instruction Formats Summary

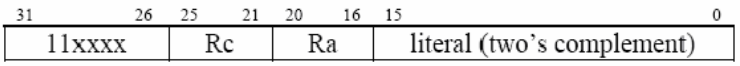
Operate Class:



Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer

OP(Ra,Ra,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

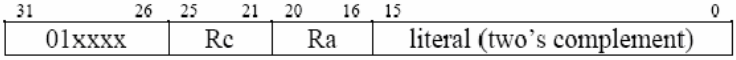
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

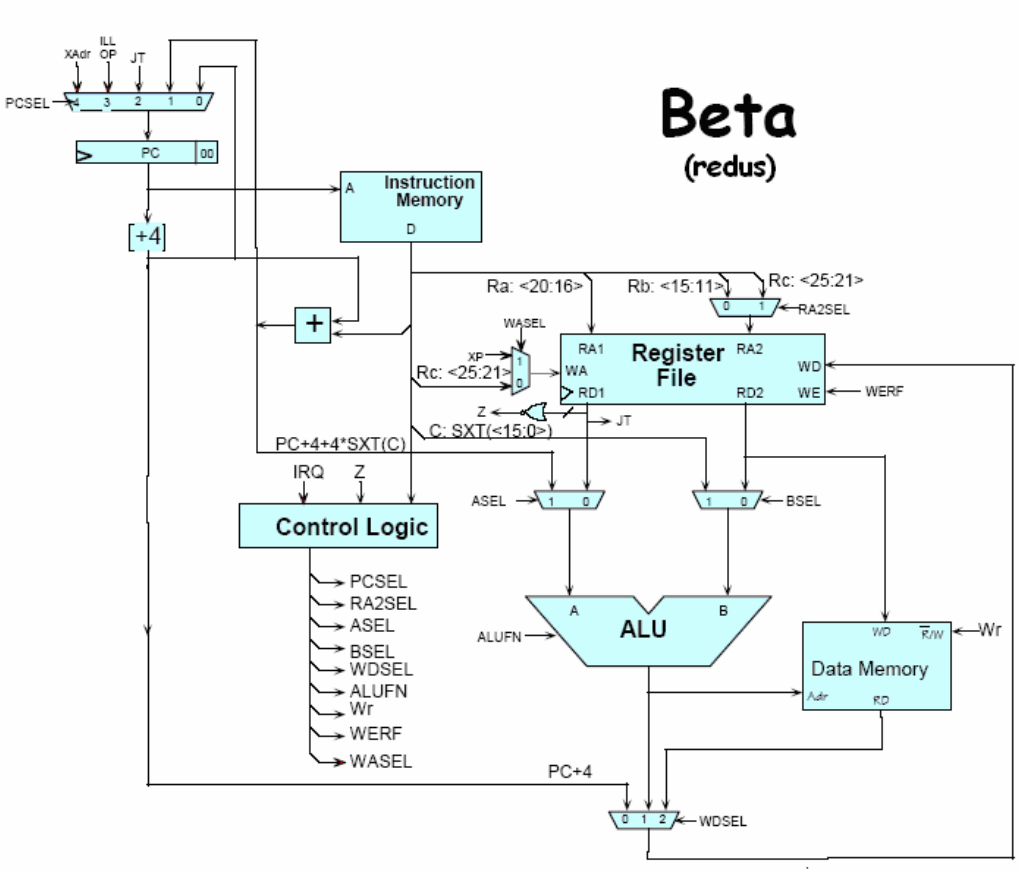
Other:

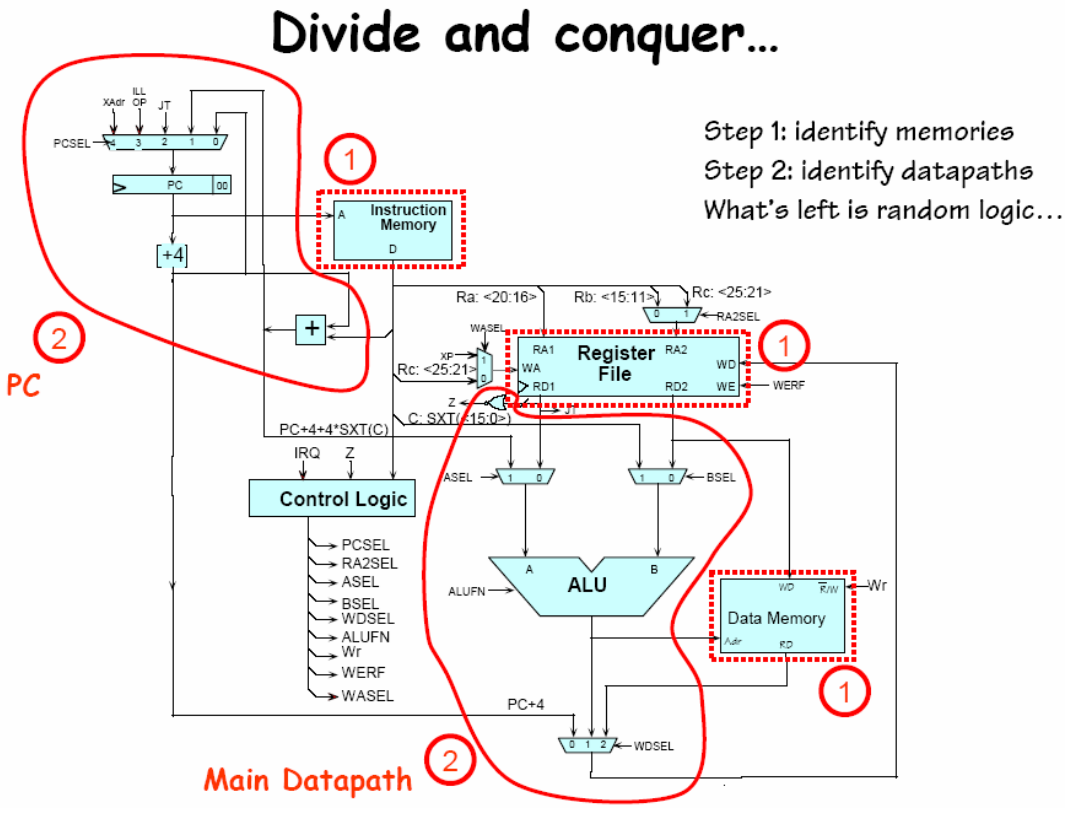


LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/**BF**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/**BT**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(Ra,label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / 5:3	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	


Beta
(redus)
XAdr
ILL OP
JT
PCSEL
PC
+4
Instruction Memory
Ra: <20:16>
Rb: <15:11>
Rc: <25:21>
RA2SEL
WASEL
XP
Register File
RA1
RA2
WA
WD
RD1
RD2
WE
WERF
Z
JT
C: SXT(<15:0>)
PC+4+4*SXT(C)
IRQ
Control Logic
ASEL
BSEL
PCSEL
RA2SEL
ASEL
BSEL
WDSEL
ALUFN
Wr
WERF
WASEL
ALU
Data Memory
PC+4
WDSEL


Divide and conquer…
Step 1: identify memories
Step 2: identify datapaths
What's left is random logic…
PC
Main Datapath
Instruction Memory
Register File
Control Logic
ALU
Data Memory

Beta Module Hierarchy

- beta
  - control [random logic]
  - regfile [memory]
  - pc [datapath]
  - datapath
    - dp_misc [datapath]
    - dp_alu
      - dp_addsub [datapath]
      - dp_boole [datapath]
      - dp_shift [datapath]
      - dp_cmp [datapath]
      - dp_mux [datapath]
    - dp_wdata [datapath]

```
// Beta PC logic
module pc(clk,reset,pcsel,offset,jump_addr,branch_addr,pc,pc_plus_4);
  input clk;
  input reset;                    // forces PC to 0x80000000
  input [2:0] pcsel;              // selects source of next PC
  input [15:0] offset;            // inst[15:0]
  input [31:0] jump_addr;      // from Reg[RA], used in JMP instruction
  output [31:0] branch_addr;   // send to datapath for LDR instruction
  output [31:0] pc;           // used as address for instruction fetch
  output [31:0] pc_plus_4;// saved in regfile during branches,JMP,traps

  reg [31:0] pc;
  wire [30:0] pcinc;
  wire [31:0] npc;

  // the Beta PC increments by 4, but won't change supervisor bit
  assign pcinc = pc + 4;
  assign pc_plus_4 = {pc[31],pcinc};

  // branch address = PC + 4 + 4*sxt(offset)
  assign branch_addr = {1'b0, pcinc +
{{13{offset[15]}},offset[15:0],2'b00}};




  assign npc = reset ? 32'h80000000 :
         (pcsel == 0) ? {pc[31],pcinc} :               // normal
         (pcsel == 1) ? {pc[31],branch_addr[30:0]} :   // branch
         (pcsel == 2) ? {pc[31] & jump_addr[31],jump_addr[30:0]}://jump
```

```
                (pcsel == 3) ? 32'h80000004 :
                (pcsel == 4) ?  32'h80000008 :         // illop, trap
                32'hXXXXXXXX;                         // catch errors...

  // pc register, pc[31] is supervisor bit and gets special treatment
  always @(posedge clk) pc <= npc;
endmodule
```

```
// 2-read, 1-write 32-location register file
module regfile(ra1,rd1,ra2,rd2,clk,werf,wa,wd);
  input [4:0] ra1;          // address for read port 1 (Reg[RA])
  output [31:0] rd1;        // read data for port 1
  input [4:0] ra2; // address for read port 2 (Reg[RB], Reg[RC] for ST)
  output [31:0] rd2;        // read data for port 2
  input clk;
  input werf;               // write enable, active high
  input [4:0] wa;           // address for write port (Reg[RC])
  input [31:0] wd;          // write data

  reg [31:0] registers[31:0];   // the register file itself

  // read paths are combinational
  // logic to ensure R31 reads as zero is in main datapath
  assign rd1 = registers[ra1];
  assign rd2 = registers[ra2];

  // write port is active only when WERF is asserted
  always @(posedge clk)
    if (werf) registers[wa] <= wd;
endmodule
```

```
module dp_addsub(fn,alu_a,alu_b,result,n,v,z);
  input fn;              // 0 for add, 1 for subtract
  input [31:0] alu_a;    // A operand
  input [31:0] alu_b;    // B operand
  output [31:0] result; // result
  output n,v,z;          // condition codes computed from result

  reg n,v,z;
  reg [31:0] result;

  always @(fn or alu_a or alu_b) begin: ripple
    integer i;          // FOR loop index, not in hardware
    reg cin,p,g;        // expanded at compile time into many signals
    reg [31:0] xb;      // hold's complement of ALU_B during subtract

    // simple ripple-carry adder for now
    xb = fn ? ~alu_b : alu_b;   // a - b == a + ~b + 1
    cin = fn;                   // carry in is 0 for add, 1 for sub
    // remember: this FOR is expanded at *compile* time
    for (i = 0; i < 32; i = i + 1) begin
      p = alu_a[i] ^ xb[i];    // carry propagate
      g = alu_a[i] & xb[i];    // carry generate
```

```
      result[i] = p ^ cin;
      cin = g | (p & cin);     // carry into next stage
    end

    n = result[31];            // negative
    z = ~|result;              // zero
    v = (alu_a[31] & xb[31] & !n) | (~alu_a[31] & ~xb[31] & n);//overfl
  end
endmodule
```

```
// fn == 2'b01: eq (Z)
// fn == 2'b10: lt (N ^ V)
// fn == 2'b11: le (Z | (N ^ V))
// result is 1 if comparison is true, 0 otherwise
module dp_cmp(fn,n,v,z,result);
  input [1:0] fn;
  input n,v,z;
  output [31:0] result;

  assign result = {31'b0,(fn[0] & z) | (fn[1] & (n ^ v))};
endmodule
```

```
// fn == 2'b00: logical shift left
// fn == 2'b01: logical shift right
// fn == 2'b11: arithmetic shift right
module dp_shift(fn,alu_a,alu_b,result);
  input [1:0] fn;
  input [31:0] alu_a;
  input [4:0] alu_b;
  output [31:0] result;

  wire sin;
  reg [31:0] u,result;
  wire [31:0] v,w,x,y,z;

  assign sin = fn[1] & alu_a[31];

  // assign u = fn[0] ? alu_a[0:31] : alu_a[31:0]
  always @(fn[0] or alu_a) begin: loopA
    integer i;
    for (i = 0; i < 32; i = i + 1)
      u[i] = fn[0] ? alu_a[31 - i] : alu_a[i];
  end

  assign v = alu_b[4] ? {u[15:0],{16{sin}}} : u[31:0];
  assign w = alu_b[3] ? {v[23:0],{8{sin}}} : v[31:0];
  assign x = alu_b[2] ? {w[27:0],{4{sin}}} : w[31:0];
  assign y = alu_b[1] ? {x[29:0],{2{sin}}} : x[31:0];
  assign z = alu_b[0] ? {y[30:0],sin} : y[31:0];

  // assign result = fn[0] ? z[0:31] : z[31:0]
  always @(fn[0] or z) begin: loopB
    integer i;
```

```
    for (i = 0; i < 32; i = i + 1)
      result[i] = fn[0] ? z[31 - i] : z[i];
  end
endmodule
```

```
// fn is truth table for boolean operation:
//  AND: fn = 4'b1000
//  OR:  fn = 4'b1110
//  XOR: fn = 4'b0110
//  A:   fn = 4'b1010
module dp_boole(fn,alu_a,alu_b,result);
  input [3:0] fn;
  input [31:0] alu_a,alu_b;
  output [31:0] result;

  reg [31:0] result;

  always @(fn or alu_a or alu_b) begin: bits
    integer i;
    for (i = 0; i < 32; i = i + 1) begin
      result[i] = alu_b[i] ? (alu_a[i] ? fn[3] : fn[2]) :
                             (alu_a[i] ? fn[1] : fn[0]);
    end
  end
endmodule
```

```
module dp_mux(sel,addsub,boole,shift,cmp,alu);
  input [1:0] sel;
  input [31:0] addsub;
  input [31:0] boole;
  input [31:0] shift;
  input [31:0] cmp;
  output [31:0] alu;

  assign alu = (sel == 2'b00) ? addsub :
               (sel == 2'b01) ? boole :
               (sel == 2'b10) ? shift :
               (sel == 2'b11) ? cmp :
               32'hXXXXXXXX;
endmodule
```

```
module dp_misc(rd1zero,rd2zero,asel,bsel,inst,rd1,rd2,branch_addr,
               alu_a,alu_b,jump_addr,mem_wr_data,z);
  input rd1zero;            // RA == R31, so treat RD1 as 0
  input rd2zero;            // RB/RC == R31, so treat RD2 as 0
  input asel;               // select A operand for ALU
  input bsel;               // select B operand for ALU
  input [15:0] inst;        // constant field from instruction
  input [31:0] rd1;         // Reg[RA] from register file
  input [31:0] rd2;      // Reg[RB] from register file (Reg[RC] for ST)
  input [31:0] branch_addr; // PC + 4 + 4*sxt(inst[15:0])
```

```
  output [31:0] alu_a;      // A input to ALU
  output [31:0] alu_b;      // B input to ALU
  output [31:0] jump_addr;  // jump address (from Reg[RA])
  output [31:0] mem_wr_data;  // data memory write data
  output z;           // true if Reg[RA] is zero, used during branches

  // R31 reads as 0, so make that adjustment here
  assign jump_addr = rd1zero ? 32'b0 : rd1;
  assign mem_wr_data = rd2zero ? 32'b0 : rd2;

  assign z = ~|jump_addr;
  assign alu_a = asel ? branch_addr : jump_addr;
  assign alu_b = bsel[0] ? {{16{inst[15]}},inst[15:0]} : mem_wr_data;
endmodule
```

```
module dp_alu(alufn,alu_a,alu_b,alu);
  input [5:0] alufn;
  input [31:0] alu_a;
  input [31:0] alu_b;
  output [31:0] alu;

  wire [31:0] addsub;
  wire [31:0] boole;
  wire [31:0] shift;
  wire [31:0] cmp;
  wire alu_n;
  wire alu_v;
  wire alu_z;

  // the four functional units
  dp_addsub alu1(alufn[0],alu_a,alu_b,addsub,alu_n,alu_v,alu_z);
  dp_boole alu4(alufn[3:0],alu_a,alu_b,boole);
  dp_shift alu3(alufn[1:0],alu_a,alu_b[4:0],shift);
  dp_cmp alu2(alufn[2:1],alu_n,alu_v,alu_z,cmp);

  // 4-to-1 mux selects alu output from functional units
  dp_mux alu5(alufn[5:4],addsub,boole,shift,cmp,alu);
endmodule
```

```
module dp_wdata(wdsel,pc_plus_4,alu,mem_data,wdata);
  input [1:0] wdsel;
  input [31:0] pc_plus_4;
  input [31:0] alu;
  input [31:0] mem_data;
  output [31:0] wdata;

  assign wdata = wdsel[1] ? mem_data :
                 wdsel[0] ? alu : pc_plus_4;
endmodule
```

```
module datapath(inst,rd1,rd2,pc_plus_4,branch_addr,mem_rd_data,
                rd1zero,rd2zero,asel,bsel,wdsel,alufn,
                wdata,mem_addr,jump_addr,mem_wr_data,z);
  input [15:0] inst;         // constant field from instruction
  input [31:0] rd1;          // Reg[RA] from register file
  input [31:0] rd2;       // Reg[RB] from register file (Reg[RC] for ST)
  input [31:0] pc_plus_4;    // incremented PC
  input [31:0] branch_addr; // PC + 4 + 4*sxt(inst[15:0])
  input [31:0] mem_rd_data; // memory read data (for LD)
  input rd1zero;             // RA == R31, so treat RD1 as 0
  input rd2zero;             // RB/RC == R31, so treat RD2 as 0
  input asel;                // select A operand for ALU
  input bsel;                // select B operand for ALU
  input [1:0] wdsel;         // select regfile write data
  input [5:0] alufn;         // operation to be performed by alu
  output [31:0] wdata;       // regfile write data (output of WDSEL mux)
  output [31:0] mem_addr; // alu output, doubles as data memory address
  output [31:0] jump_addr;  // jump address (from Reg[RA])
  output [31:0] mem_wr_data;  // data memory write data (from Reg[RC])
  output z;             // true if Reg[RA] is zero, used during branches

  wire [31:0] alu_a;         // A input to ALU
  wire [31:0] alu_b;         // B input to ALU

  // compute A and B inputs into alu, also Z bit for control logic
  dp_misc misc(rd1zero,rd2zero,asel,bsel,inst,rd1,rd2,branch_addr,
               alu_a,alu_b,jump_addr,mem_wr_data,z);

  // where all the heavy-lifting happens
  dp_alu alu(alufn,alu_a,alu_b,mem_addr);

  // select regfile write data from PC+4, alu output, and memory data
  dp_wdata wdata(wdsel,pc_plus_4,mem_addr,mem_rd_data,wdata);
endmodule
```

```
module control(reset,irq,supervisor,z,inst,

alufn,asel,bsel,pcsel,ra2,rd1zero,rd2zero,wa,wdsel,werf,mem_we);
  input reset;                  // active high
  input irq;                    // interrupt request, active high
  input supervisor;             // 1 for kernel-mode, 0 for user-mode
  input z;                      // Reg[RA] == 0, used for branch tests
  input [31:0] inst;            // current instruction
  output [5:0] alufn;           // selects ALU function
  output asel;                  // selects ALU A operand
  output bsel;                  // selects ALU B operand
  output [2:0] pcsel;           // selects next PC
  output [4:0] ra2;             // RB or RC
  output rd1zero;               // RA == 31
  output rd2zero;               // ra2 == 31
  output [4:0] wa;              // RC or XP
  output [1:0] wdsel;           // selects regfile write data
  output werf;                  // regfile write enable
  output mem_we;                // memory write enable, active high
```

```
reg [15:0] ctl;             // local
wire interrupt;

// interrupts enabled in user mode only
assign interrupt = irq & ~supervisor;

// control ROM
always @(inst or interrupt) begin
  if (interrupt)
              ctl = 16'bx_100_1xx_xxxxxx_00_0;  // interrupt
  else case (inst[31:26])
    //                     ppp     aaaaaa ww
    //                   b ccc w   llllll dd
    //                   t sss aab uuuuuu ss
    //                   e eee sss ffffff ee x
    //                   s lll eee nnnnnn ll w
    //                   t 210 lll 543210 10 r
    default:  ctl = 16'bx_011_1xx_xxxxxx_00_0;  // illegal opcode
    6'b011000: ctl = 16'bx_000_001_00xxx0_10_0;  // LD
    6'b011001: ctl = 16'bx_000_x01_00xxx0_10_1;  // ST
    6'b011011: ctl = 16'bx_010_0xx_xxxxxx_00_0;  // JMP
    6'b011101: ctl = 16'b1_001_0xx_xxxxxx_00_0;  // BEQ
    6'b011110: ctl = 16'b0_001_0xx_xxxxxx_00_0;  // BNE
    6'b011111: ctl = 16'bx_000_010_011010_10_0;  // LDR
    6'b100000: ctl = 16'bx_000_000_00xxx0_01_0;  // ADD
    6'b100001: ctl = 16'bx_000_000_00xxx1_01_0;  // SUB
    6'b100100: ctl = 16'bx_000_000_11x011_01_0;  // CMPEQ
    6'b100101: ctl = 16'bx_000_000_11x101_01_0;  // CMPLT
    6'b100110: ctl = 16'bx_000_000_11x111_01_0;  // CMPLE
    6'b101000: ctl = 16'bx_000_000_011000_01_0;  // AND
    6'b101001: ctl = 16'bx_000_000_011110_01_0;  // OR
    6'b101010: ctl = 16'bx_000_000_010110_01_0;  // XOR
    6'b101100: ctl = 16'bx_000_000_10xx00_01_0;  // SHL
    6'b101101: ctl = 16'bx_000_000_10xx01_01_0;  // SHR
    6'b101110: ctl = 16'bx_000_000_10xx11_01_0;  // SRA
    6'b110000: ctl = 16'bx_000_001_00xxx0_01_0;  // ADDC
    6'b110001: ctl = 16'bx_000_001_00xxx1_01_0;  // SUBC
    6'b110100: ctl = 16'bx_000_001_11x011_01_0;  // CMPEQC
    6'b110101: ctl = 16'bx_000_001_11x101_01_0;  // CMPLTC
    6'b110110: ctl = 16'bx_000_001_11x111_01_0;  // CMPLEC
    6'b111000: ctl = 16'bx_000_001_011000_01_0;  // ANDC
    6'b111001: ctl = 16'bx_000_001_011110_01_0;  // ORC
    6'b111010: ctl = 16'bx_000_001_010110_01_0;  // XORC
    6'b111100: ctl = 16'bx_000_001_10xx00_01_0;  // SHLC
    6'b111101: ctl = 16'bx_000_001_10xx01_01_0;  // SHRC
    6'b111110: ctl = 16'bx_000_001_10xx11_01_0;  // SRAC
  endcase
end

assign werf = ~ctl[0];
assign mem_we = !reset & ctl[0];
assign wdsel = ctl[2:1];
assign alufn = ctl[8:3];
assign bsel = ctl[9];
assign asel = ctl[10];
assign wa = ctl[11] ? 5'b11110 : inst[25:21];
```

```
  assign pcsel = ((ctl[14:12] == 3'b001) & (ctl[15] ^ z)) ? 3'b000 :
ctl[14:12];

  assign ra2 = ctl[0] ? inst[25:21] : inst[15:11];
  assign rd1zero = (inst[20:16] == 5'b11111);
  assign rd2zero = (ra2 == 5'b11111);
endmodule
```

```
module beta(clk,reset,irq,inst_addr,inst_data,
            mem_addr,mem_rd_data,mem_we,mem_wr_data);
  input clk;
  input reset;                  // active high
  input irq;                    // interrupt request, active high
  output [31:0] inst_addr;      // address of instruction to be fetched
  input [31:0] inst_data;       // instruction returning from memory
  output [31:0] mem_addr;       // address of data word to be accessed
  input [31:0] mem_rd_data;    // read data returning from memory
  output mem_we;                // memory write enable, active high
  output [31:0] mem_wr_data;    // memory write data

  // internal signals              source     description
  wire [5:0] alufn;             // [control]  selects ALU function
  wire asel;                    // [control]  selects ALU A operand
  wire [31:0] branch_addr;      // [pc]       PC + 4 +
4*sxt(inst[15:0])
  wire bsel;                    // [control]  selects ALU B operand
  wire [31:0] jump_addr;        // [datapath] Reg[RA]
  wire [31:0] pc_plus_4;        // [pc]       PC + 4
  wire [2:0] pcsel;             // [control]  selects next PC
  wire [4:0] ra2;               // [control]  RB or RC
  wire [31:0] rd1;              // [regfile]  Reg[RA]
  wire rd1zero;                 // [control]  RA == 31
  wire [31:0] rd2;              // [regfile]  Reg[ra2]
  wire rd2zero;                 // [control]  ra2 == 31
  wire [4:0] wa;                // [control]  RC or XP
  wire [1:0] wdsel;            // [control]  selects regfile write data
  wire werf;                    // [control]  regfile write enable
  wire [31:0] wdata;            // [datapath] regfile write data
  wire z;                       // [datapath] Reg[RA] == 0

  // control logic, reg file address generation
  control ctl(reset,irq,inst_addr[31],z,inst_data[31:0],

alufn,asel,bsel,pcsel,ra2,rd1zero,rd2zero,wa,wdsel,werf,mem_we);

  // program counter logic
  pc pc(clk,reset,pcsel,inst_data[15:0],jump_addr,branch_addr,
        inst_addr,pc_plus_4);

  // register file
  regfile regfile(inst_data[20:16],rd1,ra2,rd2,clk,werf,wa,wdata);

  // datapath
  datapath
dp(inst_data[15:0],rd1,rd2,pc_plus_4,branch_addr,mem_rd_data,
```

```
              rd1zero,rd2zero,asel,bsel,wdsel,alufn,
              wdata,mem_addr,jump_addr,mem_wr_data,z);
endmodule
```

```
// Set time units to be nanoseconds.
`timescale 1ns/1ns

module main;
  reg clk,reset,irq;

  wire [31:0] inst_addr;
  wire [31:0] inst_data;
  wire [31:0] mem_addr;
  wire [31:0] mem_wr_data;
  wire [31:0] mem_rd_data;
  wire mem_we;

  reg [31:0] memory[1023:0];

  beta beta(clk,reset,irq,inst_addr,inst_data,
            mem_addr,mem_rd_data,mem_we,mem_wr_data);

  // main memory (2 async read ports, 1 sync write port)
  assign inst_data = memory[inst_addr[13:2]];
  assign mem_rd_data = memory[mem_addr[13:2]];
  always @(posedge clk)
    if (mem_we) memory[mem_addr[13:2]] <= mem_wr_data;

  always #5 clk = ~clk;

  initial begin
    $dumpfile("beta_checkoff.vcd");
    $dumpvars;
    $readmemh("beta_checkoff",memory);
    clk = 0; irq = 0; reset = 1;
    #10
    reset = 0;
    #3000    // 300 cycles
    $finish;
  end
endmodule
```