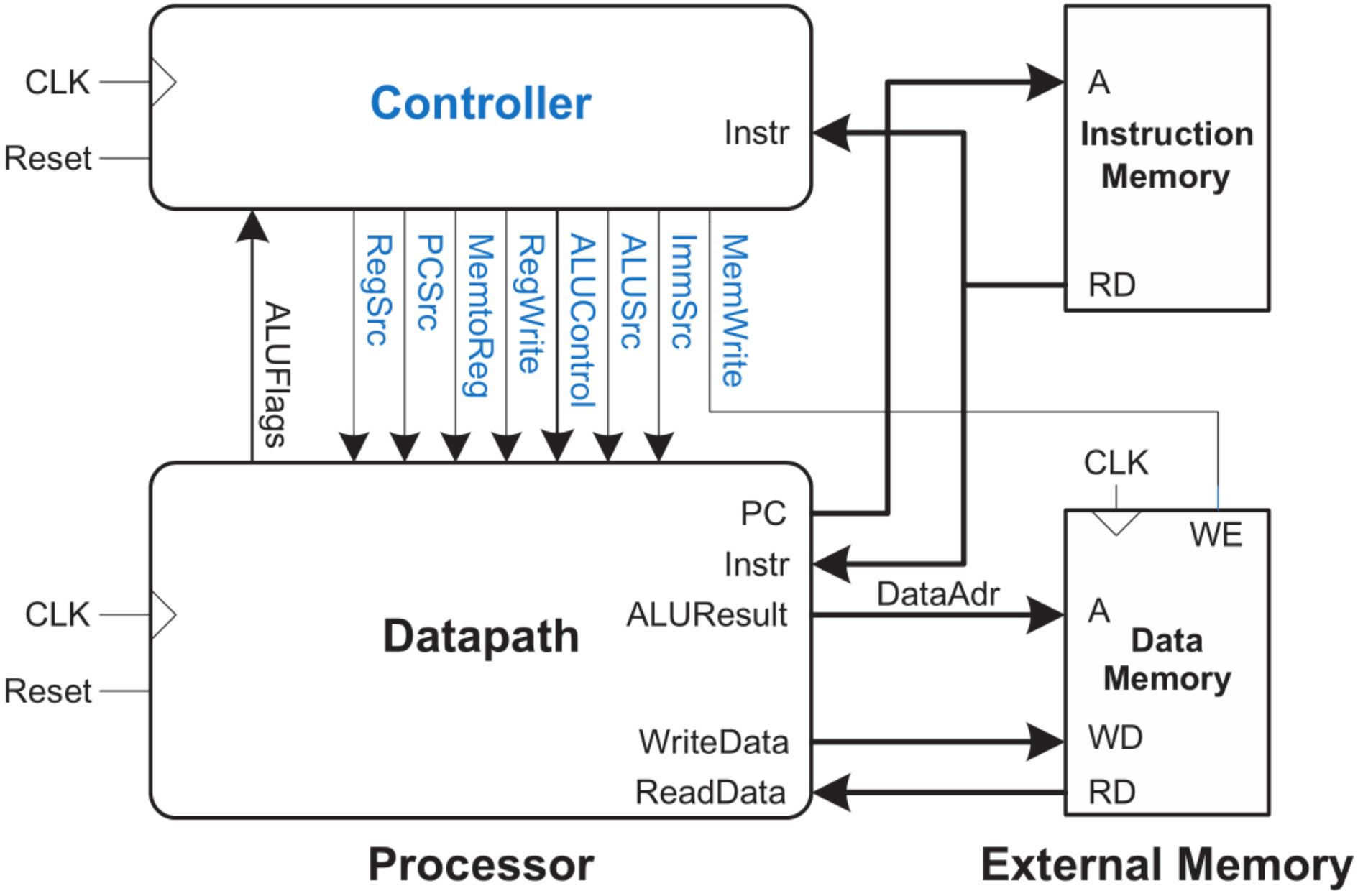
Computer organization and architecture

Lesson 17

HDL for the single-cycle processor

Single-cycle processor



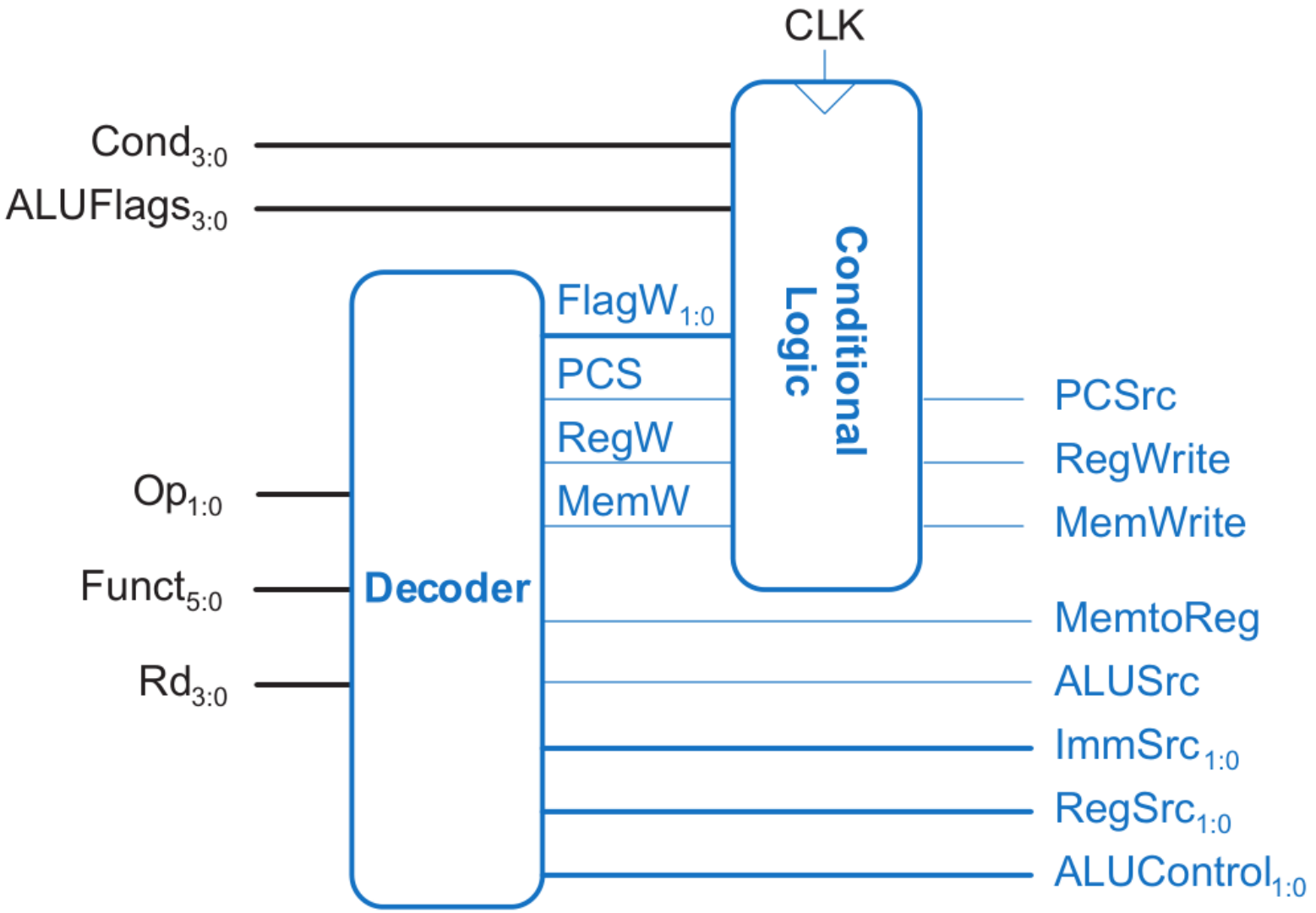
Single-cycle processor

```
module arm (input          clk, reset,
            input [31:0]   Instr, ReadData,
            output         MemWrite,
            output [31:0]  ALUResult, WriteData, PC);

  wire        RegWrite, ALUSrc, MemtoReg, PCSrc;
  wire [1:0]  RegSrc, ImmSrc, ALUControl;
  wire [3:0]  ALUFlags;

  controller c(clk, reset, ALUFlags, Instr[31:12], RegSrc,
               ImmSrc, ALUControl, RegWrite, ALUSrc,
MemWrite,
            MemtoReg, PCSrc);

  datapath dp(clk, reset, RegWrite, ALUSrc, MemtoReg, PCSrc,
              RegSrc, ImmSrc, ALUControl, Instr, ReadData,
              ALUFlags, PC, ALUResult, WriteData);
endmodule
```

Top-level module

```
module top(input clk, reset,
           output [31:0] WriteData, DataAdr, Instr,
           output  MemWrite);
  wire [31:0] PC, ReadData;
  // instantiate processor and memories
  arm arm(clk, reset, Instr, ReadData,MemWrite,
          DataAdr, WriteData, PC);
  imem imem(PC, Instr);
  dmem dmem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

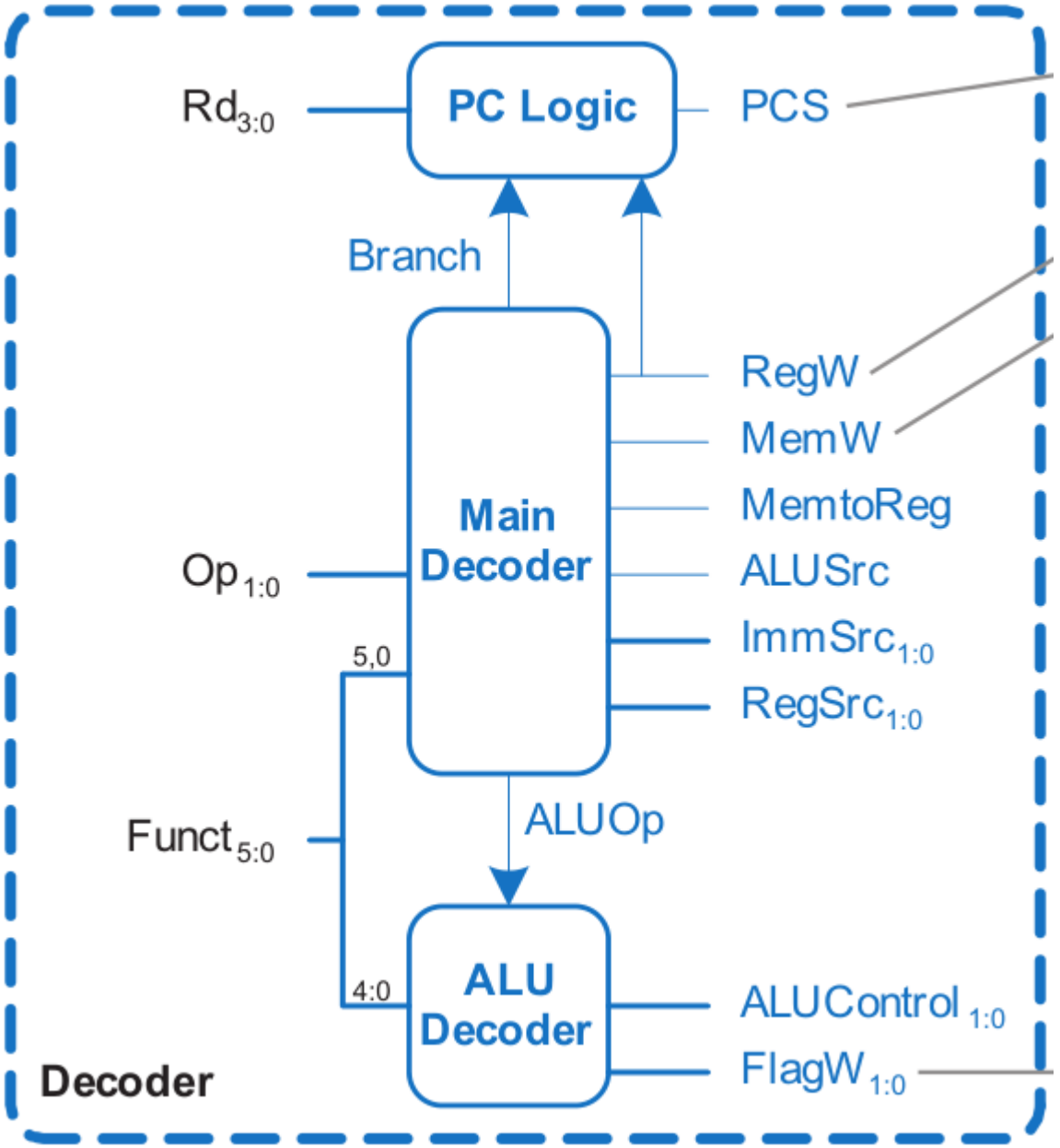
Controller


CLK
Cond3:0
ALUFlags3:0
Op1:0
Funct5:0
Rd3:0
Decoder
FlagW1:0
PCS
RegW
MemW
Conditional Logic
PCSrc
RegWrite
MemWrite
MemtoReg
ALUSrc
ImmSrc1:0
RegSrc1:0
ALUControl1:0

Controller

```
module controller(input          clk, reset,
                  input [3:0]    ALUFlags,
                  input [31:12]  Instr,
                  output [1:0]   RegSrc, ImmSrc, ALUControl,
                  output         RegWrite, ALUSrc, MemWrite,
                                 MemtoReg, PCSrc);

  wire [1:0] FlagW;
  wire PCS, RegW, MemW;

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              PCS, RegW, MemW, MemtoReg, ALUSrc, FlagW,
              ImmSrc, RegSrc, ALUControl);

  condlogic cl(clk, reset, PCS, RegW, MemW,
               Instr[31:28], ALUFlags, FlagW,
               PCSrc, RegWrite, MemWrite);
endmodule
```

Decoder


Rd3:0
PC Logic
PCS
Branch
RegW
MemW
Main Decoder
MemtoReg
ALUSrc
Op1:0
ImmSrc1:0
5,0
RegSrc1:0
ALUOp
Funct5:0
ALU Decoder
4:0
ALUControl1:0
FlagW1:0
Decoder

Decoder

```
module decoder(input [1:0]  Op,
               input [5:0]  Funct,
               input [3:0]  Rd,
               output       PCS, RegW, MemW, MemtoReg,
                            ALUSrc,
               output [1:0] FlagW, ImmSrc, RegSrc,
                            ALUControl);

  wire Branch, ALUOp;

  maindecoder mdec(Op, Funct[5], Funct[0], RegSrc,
                   ImmSrc, ALUSrc, MemtoReg, MemW, RegW,
                   Branch, ALUOp);

  ALUdecoder aludec(ALUOp, Funct[4:0], FlagW, ALUControl);

  PClogic pcl(Rd, Branch, RegW, PCS);

endmodule
```

Main decoder

```
module maindecoder(input [1:0]  Op,
                   input        Funct5, Funct0,
                   output [1:0] RegSrc, ImmSrc,
                   output       ALUSrc, MemtoReg, MemW, RegW,
                                Branch, ALUOp);
  reg [9:0] controls;

  always @ (*)
    casex(Op)
      2'b00: if (Funct5) controls = 10'b0001001001; // DP imm
             else        controls = 10'b0000001001; // DP reg
      2'b01: if (Funct0) controls = 10'b0101011000; // LDR
             else        controls = 10'b0011010100; // STR
      2'b10:             controls = 10'b1001100010; // B
      default:           controls = 10'bx; // Unimplemented
    endcase

  assign {Branch, MemtoReg, MemW, ALUSrc, ImmSrc,
          RegW, RegSrc, ALUOp} = controls;
endmodule
```

Main decoder

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

```
always @ (*)
  casex(Op)
    2'b00: if (Funct5) controls = 10'b0001001001; // DP imm
           else        controls = 10'b0000001001; // DP reg
    2'b01: if (Funct0) controls = 10'b0101011000; // LDR
           else        controls = 10'b0011010100; // STR
    2'b10:             controls = 10'b1001100010; // B
    default:           controls = 10'bx; // Unimplemented
  endcase

assign {Branch, MemtoReg, MemW, ALUSrc, ImmSrc,
        RegW, RegSrc, ALUOp} = controls;
```

ALUDecoder

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

ALUDecoder

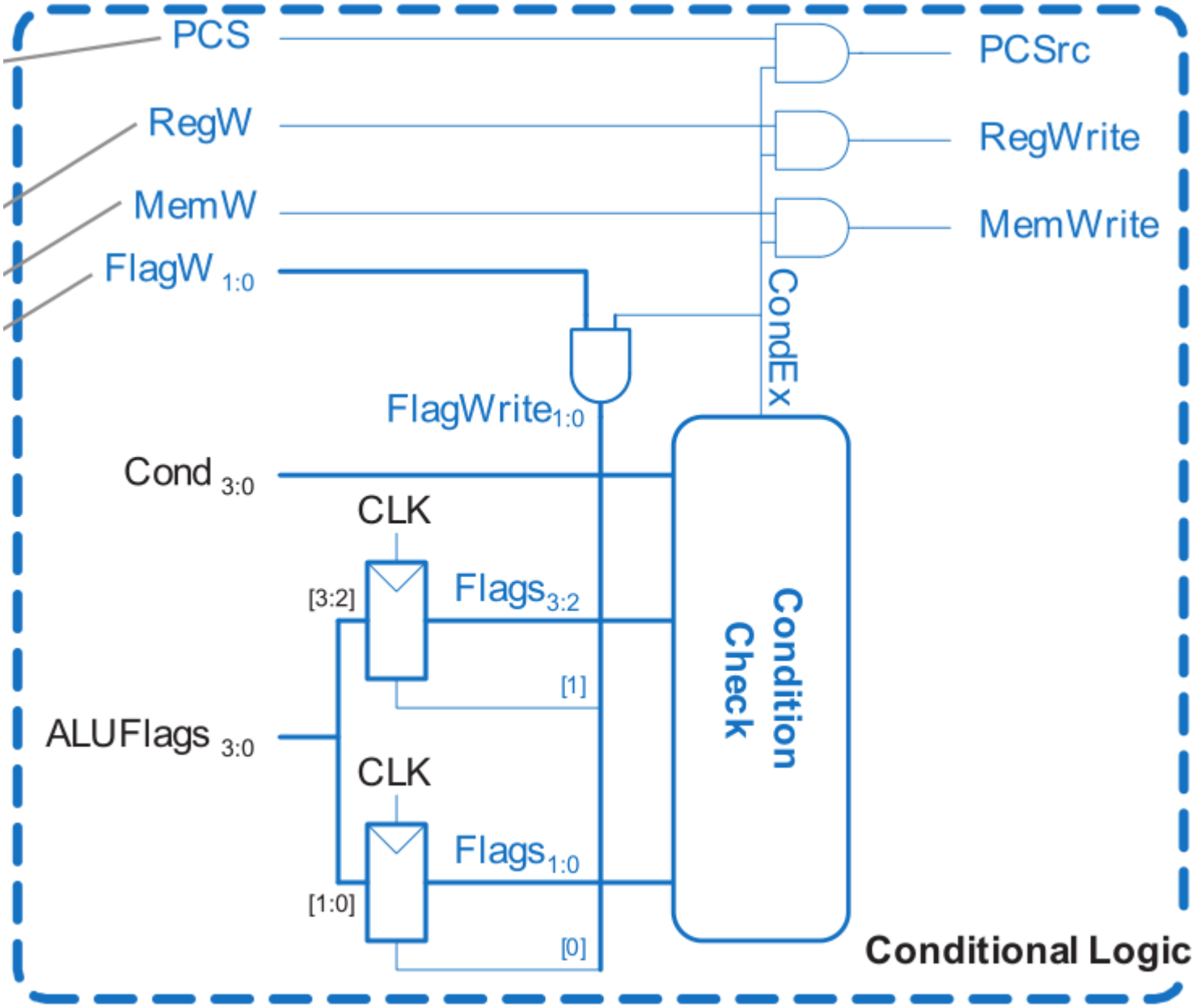
```
module ALUdecoder(input            ALUOp,
                  input [4:0]      Funct,
                  output reg [1:0] FlagW, ALUControl);
  always @ (*)
    if (ALUOp)
   begin // which DP Instr?
      case(Funct[4:1])
        4'b0100: ALUControl = 2'b00; // ADD
        4'b0010: ALUControl = 2'b01; // SUB
        4'b0000: ALUControl = 2'b10; // AND
        4'b1100: ALUControl = 2'b11; // ORR
        default: ALUControl = 2'bx; // unimplemented
      endcase
      FlagW[1] = Funct[0];
      FlagW[0] = Funct[0] &
                 (ALUControl == 2'b00 | ALUControl == 2'b01);
    end else begin
      ALUControl = 2'b00; // non-DP instr
      FlagW = 2'b00; // don't update Flags
    end
endmodule
```

PC Logic

```
PCS = ((Rd == 15) & RegW) | Branch
```

```
module PClogic(input [3:0] Rd,
               input       Branch, RegW,
             output      PCS);
  assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

Conditional Logic

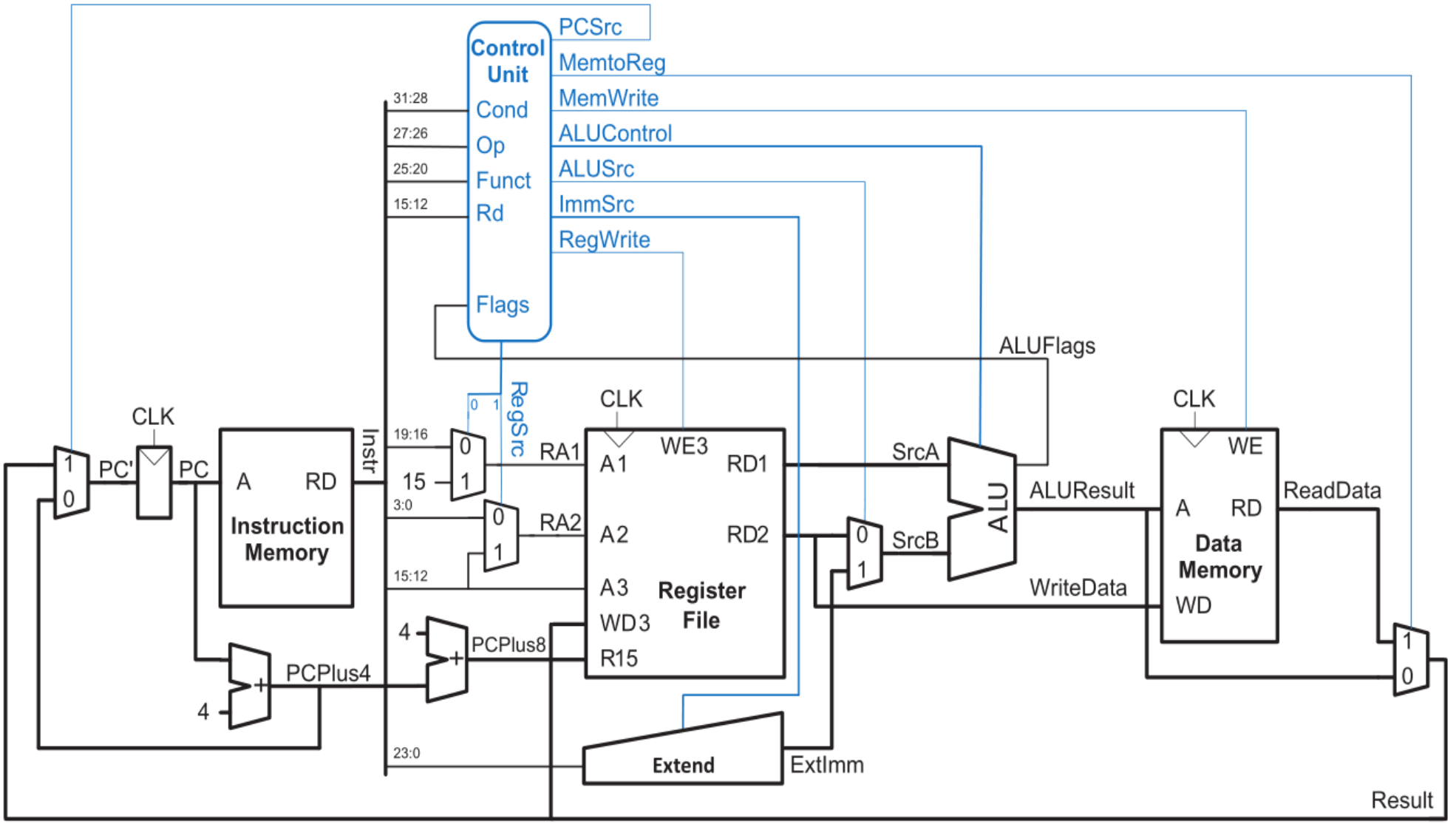


```
module condlogic(input        clk, reset, PCS, RegW, MemW,
                 input [3:0] Cond, ALUFlags,
                 input [1:0] FlagW,
                 output      PCSrc, RegWrite, MemWrite);
  wire [1:0] FlagWrite;
  wire [3:0] Flags;
  wire CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  condcheck cc(Cond, Flags, CondEx);

  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite = RegW & CondEx;
  assign MemWrite = MemW & CondEx;
  assign PCSrc = PCS & CondEx;
endmodule
```

```
module condcheck(input [3:0] Cond, Flags,
                 output reg  CondEx);
  wire neg, zero, carry, overflow, ge;
  assign {neg,zero,carry,overflow}=Flags;
  assign ge = (neg == overflow);
  always @ (*)
    case(Cond)
      4'b0000: CondEx = zero; // EQ
      4'b0001: CondEx = ~zero; // NE
      4'b0010: CondEx = carry; // CS
      4'b0011: CondEx = ~carry; // CC
      4'b0100: CondEx = neg; // MI
      4'b0101: CondEx = ~neg; // PL
      4'b0110: CondEx = overflow; // VS
      4'b0111: CondEx = ~overflow; // VC
      4'b1000: CondEx = carry & ~zero; // HI
      4'b1001: CondEx = ~(carry & ~zero);//LS
      4'b1010: CondEx = ge; // GE
      4'b1011: CondEx = ~ge; // LT
      4'b1100: CondEx = ~zero & ge; // GT
      4'b1101: CondEx = ~(~zero & ge); // LE
      4'b1110: CondEx = 1'b1; // Always
      default: CondEx = 1'bx; // undefined
    endcase
endmodule
```

cond	Mnemonic	CondEx
0000	EQ	Z
0001	NE	$\overline{Z}$
0010	CS/HS	C
0011	CC/LO	$\overline{C}$
0100	MI	N
0101	PL	$\overline{N}$
0110	VS	V
0111	VC	$\overline{V}$
1000	HI	$\overline{Z}C$
1001	LS	Z OR $\overline{C}$
1010	GE	$\overline{N \oplus V}$
1011	LT	$N \oplus V$
1100	GT	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Z OR $(N \oplus V)$
1110	AL	Ignored

Datapath


Control Unit
Cond
Op
Funct
Rd
Flags
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
ALUFlags
31:28
27:26
25:20
15:12
RegSrc
CLK
PC'
PC
A
RD
Instruction Memory
Instr
19:16
15
3:0
15:12
RA1
RA2
A1
A2
A3
WD3
R15
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
ALUResult
WriteData
WE
WD
Data Memory
ReadData
PCPlus4
PCPlus8
4
Extend
ExtImm
23:0
Result

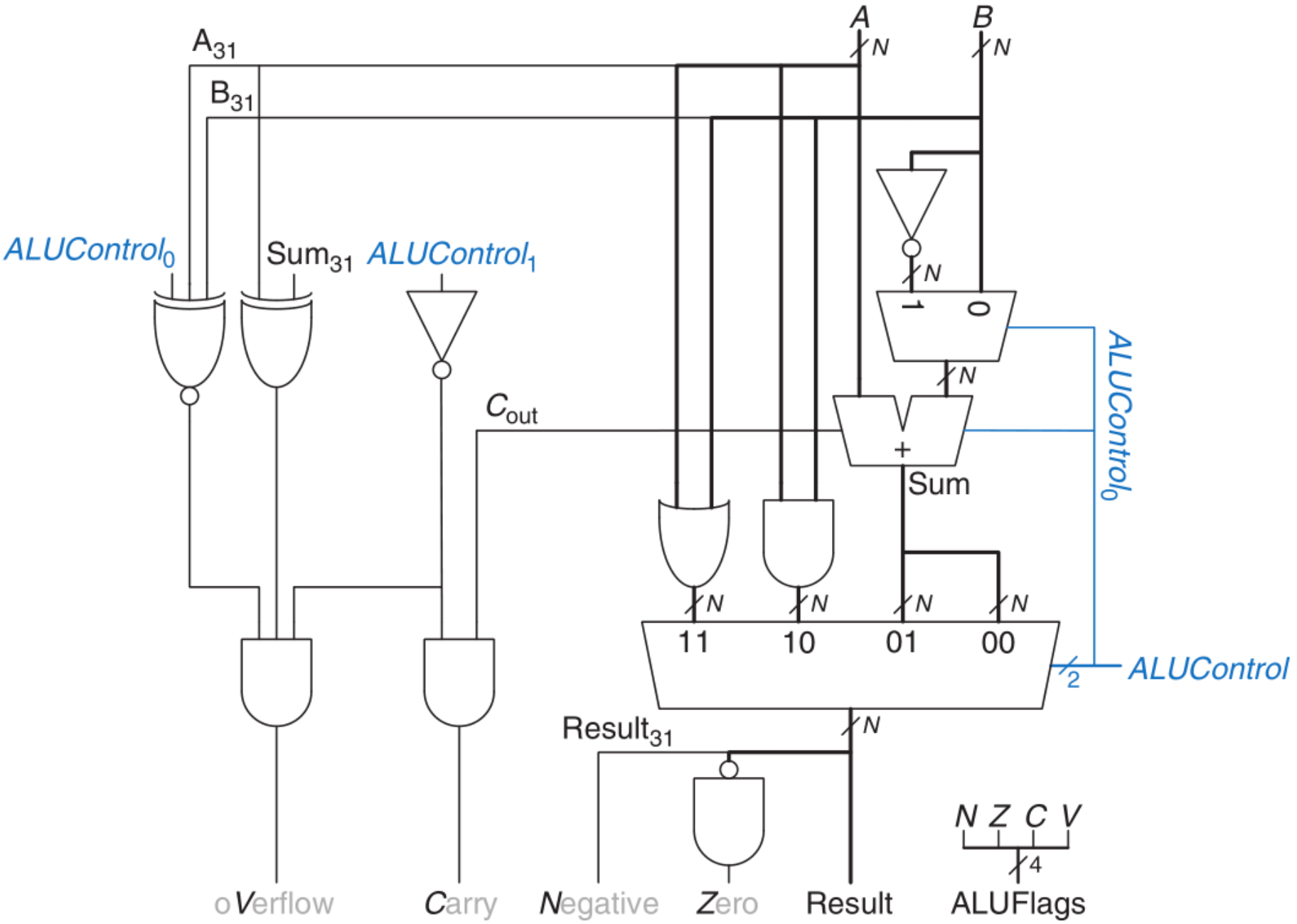
```
module datapath(input clk, reset, RegWrite, ALUSrc, MemtoReg, PCSrc,
                input [1:0]   RegSrc, ImmSrc, ALUControl,
                input  [31:0] Instr, ReadData,
                output [3:0]  ALUFlags,
                output [31:0] PC, ALUResult, WriteData);
  wire [31:0] PCNext, PCPlus4, PCPlus8;
  wire [31:0] ExtImm, SrcA, SrcB, Result;
  wire [3:0] RA1, RA2;
    // next PC logic
  mux2 #(32) pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32) pcreg(clk, reset, PCNext, PC);
  adder #(32) pcadd1(PC, 32'b100, PCPlus4);
  adder #(32) pcadd2(PCPlus4, 32'b100, PCPlus8);
    // register file logic
  mux2 #(4) ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4) ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile rf(clk, RegWrite, RA1, RA2,
             Instr[15:12], Result, PCPlus8, SrcA, WriteData);
  mux2 #(32) resmux(ALUResult, ReadData, MemtoReg, Result);
  extend ext(Instr[23:0], ImmSrc, ExtImm);
    // ALU logic
  mux2 #(32) srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
endmodule
```

ALU

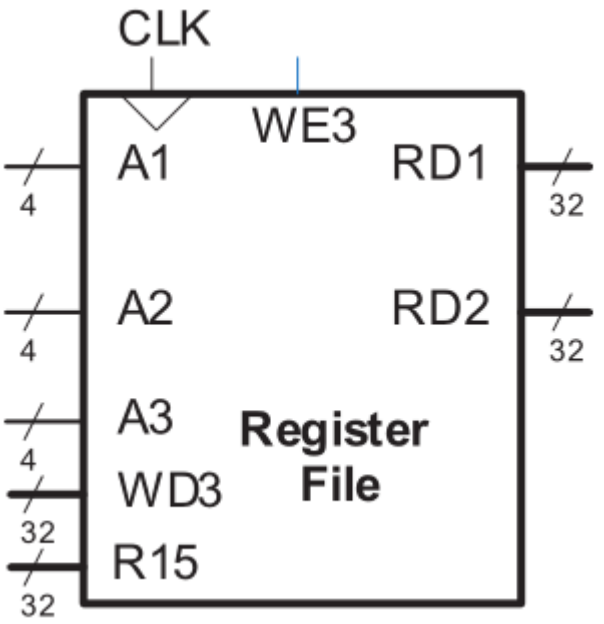
ALUControl	Function
00	Add
01	Subtract
10	AND
11	OR

```
module alu(input [31:0]  SrcA, SrcB,
           input [1:0]   ALUControl,
           output [31:0] ALUResult,
           output [3:0]  ALUFlags);
  wire [31:0] Sum, invSrcB, SignB;
  wire cout;
  assign invSrcB = SrcB ^ {32{1'b1}};

  mux2 #(32) alumux2 (SrcB, invSrcB, ALUControl[0], SignB);
  addercio #(32) aluadd (SrcA, SignB, ALUControl[0], Sum, cout);
  mux4 #(32) alumux4 (Sum, Sum, SrcA & SrcB, SrcA | SrcB,
              ALUControl, ALUResult);

  assign ALUFlags[3] = ALUResult[31]; // N
  assign ALUFlags[2] = ~(|ALUResult); // Z
  assign ALUFlags[1] = ~ALUControl[1] & cout; // C
  assign ALUFlags[0] = ~ALUControl[1] &
                       (Sum[31] ^ SrcA[31]) &
                       ~(ALUControl[0] ^ SrcA[31] ^ SrcB[31]);//V
endmodule
```

ALU


A
B
N
A31
B31
ALUControl0
Sum31
ALUControl1
Cout
1
0
+
Sum
ALUControl0
11
10
01
00
2
ALUControl
Result31
N Z C V
4
ALUFlags
oVerflow
Carry
Negative
Zero
Result

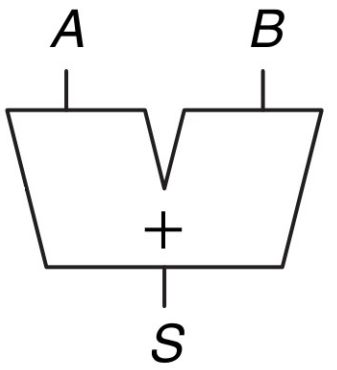
Register file




```
module regfile(input clk, we3,
               input [3:0] ra1, ra2, wa3,
               input [31:0] wd3, r15,
               output [31:0] rd1, rd2);
  reg [31:0] rf[14:0];

  always @(posedge clk)
    if (we3) rf[wa3] <= wd3;

  assign rd1 = (ra1 == 4'b1111) ? r15 : rf[ra1];
  assign rd2 = (ra2 == 4'b1111) ? r15 : rf[ra2];
endmodule
```

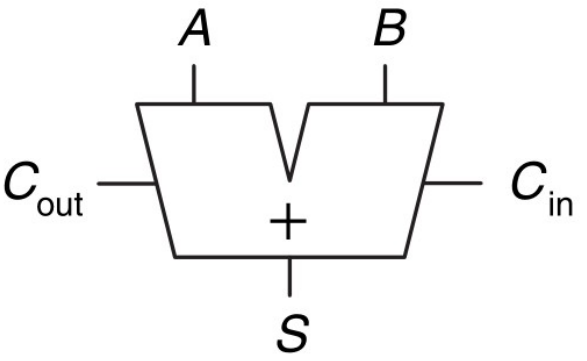
Adder

```
module adder #(parameter WIDTH=8)
              (input [WIDTH-1:0] a, b,
               output [WIDTH-1:0] y);
  assign y = a + b;
endmodule
```



Adder with carry in and carry out

```
module addercio #(parameter WIDTH=8)
                 (input [WIDTH-1:0] a, b,
                  input cin,
                  output [WIDTH-1:0] s,
                  output cout);
  assign {cout, s} = a + b + cin;
endmodule
```



Immediate extension

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for LDR/STR
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for B

```
module extend(input [23:0]      Instr,
              input [1:0]       ImmSrc,
              output reg [31:0] ExtImm);
  always @ (*)
    case(ImmSrc)
      2'b00: ExtImm = {24'b0, Instr[7:0]}; // 8-bit unsigned
      2'b01: ExtImm = {20'b0, Instr[11:0]};//12-bit unsigned
      // 24-bit signed
      2'b10: ExtImm = {{6{Instr[23]}}, Instr[23:0], 2'b00};
      default: ExtImm = 32'bx; // undefined
    endcase
endmodule
```

Resettable flip-flop

```
module flopr #(parameter WIDTH = 8)
              (input clk, reset,
               input [WIDTH-1:0] d,
               output reg [WIDTH-1:0] q);
  always @(posedge clk, posedge reset)
    if (reset) q <= {WIDTH{1'b0}};
    else q <= d;
endmodule
```

Resettable flip-flop with enable

```
module flopenr #(parameter WIDTH = 8)
                (input clk, reset, en,
                 input [WIDTH-1:0] d,
                 output reg [WIDTH-1:0] q);
  always @(posedge clk, posedge reset)
    if (reset) q <= {WIDTH{1'b0}};
    else if (en) q <= d;
endmodule
```

2:1 multiplexer

```
module mux2 #(parameter WIDTH = 8)
             (input [WIDTH-1:0] d0, d1,
              input s,
              output [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

4:1 multiplexer

```
module mux4 #(parameter WIDTH = 8)
             (input [WIDTH-1:0] d0, d1, d2, d3,
              input [1:0] s,
              output [WIDTH-1:0] y);
  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

The testbench loads a program into the memories.

```
ADDR        PROGRAM                 ; COMMENTS                              HEX CODE
00  MAIN    SUB R0, R15, R15        ; R0 = 0                                E04F000F
04          ADD R2, R0, #5          ; R2 = 5                                E2802005
08          ADD R3, R0, #12         ; R3 = 12                               E280300C
0C          SUB R7, R3, #9          ; R7 = 3                                E2437009
10          ORR R4, R7, R2          ; R4 = 3 OR 5 = 7                       E1874002
14          AND R5, R3, R4          ; R5 = 12 AND 7 = 4                     E0035004
18          ADD R5, R5, R4          ; R5 = 4 + 7 = 11                       E0855004
1C          SUBS R8, R5, R7         ; R8 = 11 - 3 = 8, set Flags            E0558007
20          BEQ END                 ; shouldn't be taken                    0A00000C
24          SUBS R8, R3, R4         ; R8 = 12 - 7  = 5                      E0538004
28          BGE AROUND              ; should be taken                       AA000000
2C          ADD R5, R0, #0          ; should be skipped                     E2805000
30 AROUND   SUBS R8, R7, R2         ; R8 = 3 - 5 = -2, set Flags            E0578002
34          ADDLT R7, R5, #1        ; R7 = 11 + 1 = 12                      B2857001
38          SUB R7, R7, R2          ; R7 = 12 - 5 = 7                       E0477002
3C          STR R7, [R3, #84]       ; mem[12+84] = 7                        E5837054
40          LDR R2, [R0, #96]       ; R2 = mem[96] = 7                      E5902060
44          ADD R15, R15, R0        ; PC = PC+8 (skips next)                E08FF000
48          ADD R2, R0, #14         ; shouldn't happen                      E280200E
4C          B END                   ; always taken                          EA000001
50          ADD R2, R0, #13         ; shouldn't happen                      E280200D
54          ADD R2, R0, #10         ; shouldn't happen                      E280200A
58 END      STR R2, [R0, #100]      ; mem[100] = 7                          E5802064
```

The program in the previous page exercises all of the instructions by performing a computation that should produce the correct result only if all of the instructions are functioning correctly.

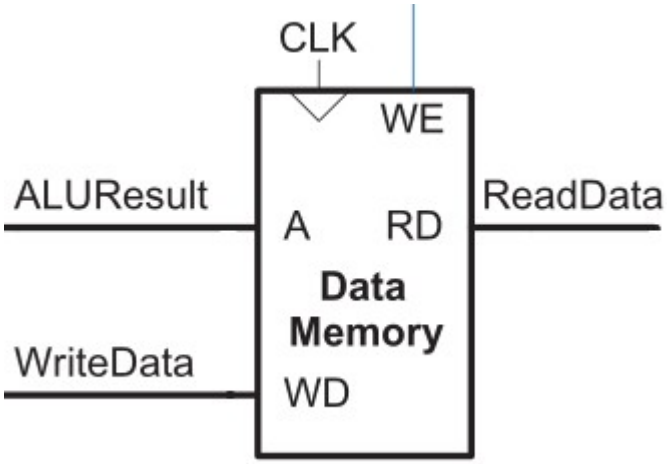
The program will write the value 7 to address 100 if it runs correctly, but it is unlikely to do so if the hardware is buggy.

This is an example of **ad hoc testing**.

The machine code is stored in a hexadecimal file called memfile.dat, which is loaded by the testbench during simulation.

The file consists of the machine code for the instructions, one instruction per line.

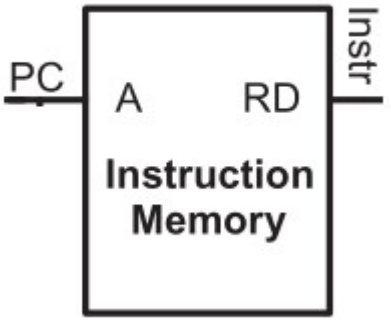
Data memory




```
module dmem(input clk, we,
            input [31:0] a, wd,
            output [31:0] rd);
  reg [31:0] RAM[63:0];
  assign rd = RAM[a[31:2]]; // word aligned
  always @(posedge clk)
    if (we) RAM[a[31:2]] <= wd;
endmodule
```

```
module imem (input [31:0] adr,
             output reg [31:0] rd);
  always @ (adr)
    case (adr)
      32'h00: rd <= 32'hE04F000F;
      32'h04: rd <= 32'hE2802005;
      32'h08: rd <= 32'hE280300C;
      32'h0C: rd <= 32'hE2437009;
      32'h10: rd <= 32'hE1874002;
      32'h14: rd <= 32'hE0035004;
      32'h18: rd <= 32'hE0855004;
      32'h1C: rd <= 32'hE0558007;
      32'h20: rd <= 32'h0A00000C;
      32'h24: rd <= 32'hE0538004;
      32'h28: rd <= 32'hAA000000;
      32'h2C: rd <= 32'hE2805000;
      32'h30: rd <= 32'hE0578002;
      32'h34: rd <= 32'hB2857001;
      32'h38: rd <= 32'hE0477002;
      32'h3C: rd <= 32'hE5837054;
      32'h40: rd <= 32'hE5902060;
      32'h44: rd <= 32'hE08FF000;
      32'h48: rd <= 32'hE280200E;
      32'h4C: rd <= 32'hEA000001;
      32'h50: rd <= 32'hE280200D;
      32'h54: rd <= 32'hE280200A;
      32'h58: rd <= 32'hE5802064;
      32'h5C: rd <= 32'hEAFFFFFE;
    endcase
endmodule
```

Instruction memory



Testbench

```
module testbench();

  reg clk, reset;
  wire [31:0] WriteData, DataAdr, Instr;
  wire MemWrite;

  // instantiate device to be tested
  top dut(clk, reset, WriteData, DataAdr, Instr, MemWrite);

  initial begin  // initialize test
      reset <= 1; #1; reset <= 0;
  end

  always begin // generate clock to sequence tests
     clk = 1; #5; clk = 0; #5;
    if ($time>250) $finish;
  end
  initial
  begin
  $monitor(
       "clk=%b Instr=%h WriteData=%h, DataAdr=%h, MemWrite=%b",
       clk, Instr, WriteData, DataAdr,MemWrite);
  end
endmodule
```

Good links for learning Verilog

http://www.verilogpro.com/

http://vol.verilog.com/

http://www.verilog.com/

https://people.ece.cornell.edu/land/courses/ece5760/Verilog/

http://www.users.miamioh.edu/jamiespa/verilogTown/index.html

https://github.com/verilog-to-routing/vtr-verilog-to-routing

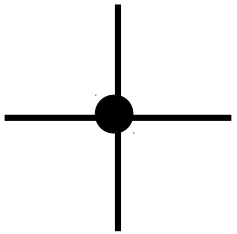
https://www.altera.com/support/training/course/ohdl1120.html

http://www.asic-world.com/verilog/verilinks.html

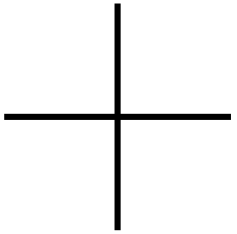
http://www.ece.uvic.ca/~fayez/courses/ceng465/vlogref.pdf

Exercise 17.1

In electronic circuits, the contact between two wires is shown as



The absence of contact (the wires are not connected) is shown as



However, all circuit diagrams in the lessons 14, 15, 16, 17 don't follow this convention (very bad!)

Go through each circuit diagram and mark every contact.

This will be on the quiz and exam.

Exercise 17.2

Modify the HDL code for the single-cycle ARM processor (in the file scp.v in Power Campus) to handle the instructions

- CMP
- TST
- LSL
- CMN
- ADC

Enhance the testbench to test the new instructions.