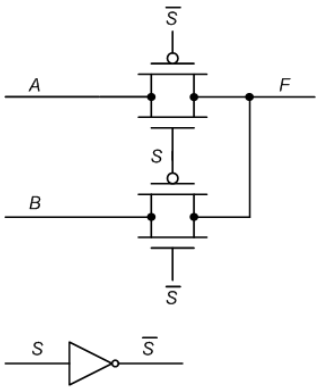
EECS150: Homework 5, Transistors, Single-cycle MIPS Processor

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

1. Draw the circuit diagram for a 2:1 Multiplexer using only 6 transistors.

   See Figure 1.

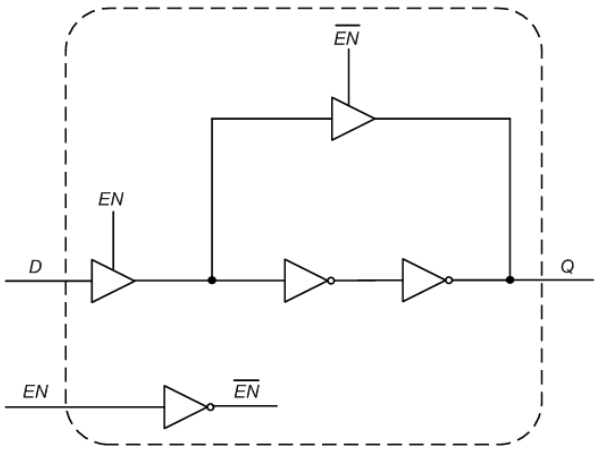
Figure 1 A 2:1 Multiplexer. The control signal `S` is used to control two transmission gates that both connect to the output but the inverter guarantees that only 1 transmission gate is on at a time and thus the output is "selected."



2. Draw the circuit diagram for a positive level sensitive latch using only inverters, and tri-state buffers.

   See Figure 2.

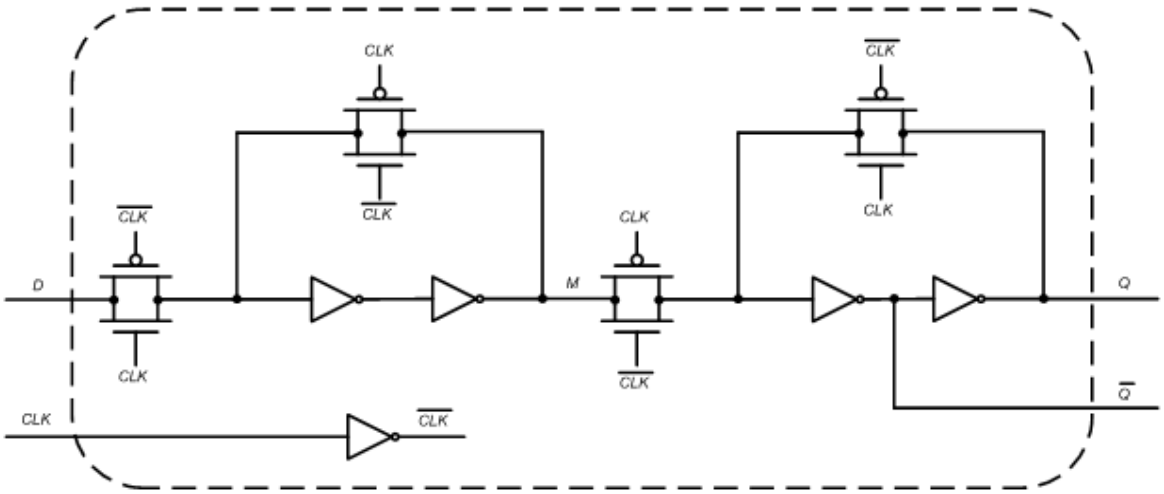
Figure 2 Positive level sensitive latch. To get this from the latch presented in the lecture slides, just replace all the transmission gates with tristate buffers, which are functionally equivalent.



3. Draw a transistor-level diagram showing all the details of a negative edge-triggered flip-flop. Your circuit should have only two inputs, **clk** and **d**, and two outputs, **q** and $\overline{\mathbf{q}}$.

   See Figure 3.

Figure 3 Negative edge-triggered flip-flop. Just cascade a positive level sensitive latch with a negative level sensitive latch. To make this positive edge-triggered, just flip the ordering of the two latches (negative level sensitive first).



4. Draw a transistor-level circuit diagram for a switching circuit with two data inputs **x0** and **x1**, one control input **cross**, and two outputs **y0** and **y1**, with the following function:

```
if (cross == 1)then y0 = x1, y1 = x0 else y0 = x0, y1 = x1
```

   See Figure 4.

5. Build the FSM in exercise 4.22 down to the level of transistors. Please draw your circuit diagram hierarchically (i.e. implement an OR gate using transistors, then start using it in your main circuit).

   We will begin thinking about how to build this FSM by first splitting it into three different parts: the State registers, the output logic, and the NextState logic. The State registers, shown in Figure 5, is easy to build - it is just 2 D-flip flops in parallel, each responsible for holding one bit of the state. Each flip flop is implemented in Figure 6 and note also that they have an **asynchronous reset**.

   The output logic is relatively straightforward - just see whether `State` equals the encoding for `S1` or `S2`, namely whether `State == 2'b01` or `State == 2'b10`. This is implemented in Figure 7.

   The next state logic is a bit tricky. It is probably easiest to begin with a truth table, such as the one in Table 1. Note that the NextState function has four inputs (`State[1:0]`, `A`, and `B`) and two outputs (`NextState[1:0]`).

   With the truth table in hand, we can now use *Logisim* to help us quickly draw the equivalent circuit, shown in Figure 8.

   We can implement two-input OR and AND gates by just using NOR or NAND gates followed by an inverter, shown in Figure 9.

   How all three pieces connect together to form the FSM is shown in Figure 10.

6. Consider the single-cycle MIPS processor presented in lecture this week. Remember, it executes only the small set of the complete MIPS instruction set: **add**, **sub**, **or**, **slt**, **lw**, **sw**, **beq**.

Figure 4 We can build this circuit first by using the multiplexer-based circuit on the left. The output of each multiplexer is connected to an output. The multiplexer simply selects which input, `X1` or `X2`, for the output to take. Then, we translate this implementation into transistor diagram on the right.

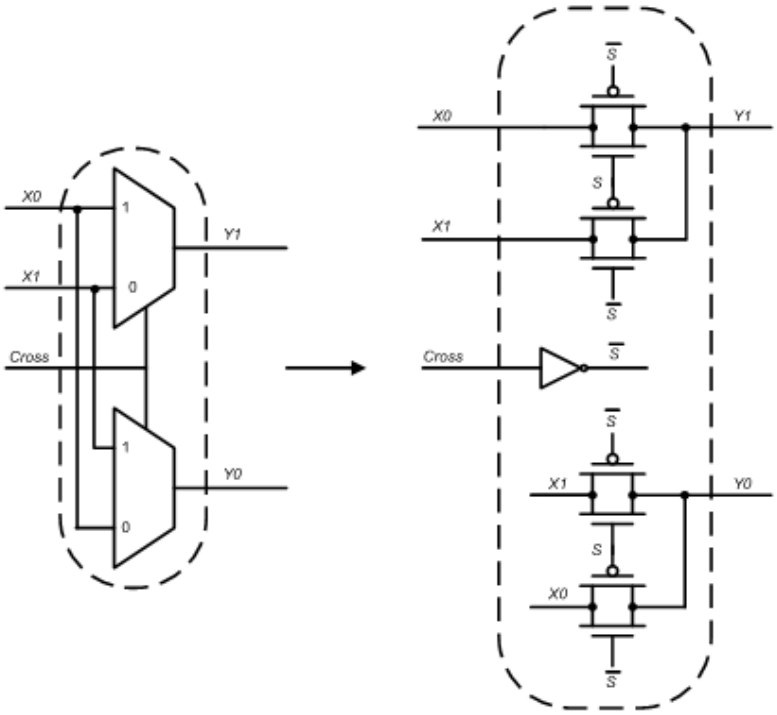


Figure 5 The part of the FSM that stores the State signals. There are 2 D-flip flops, one for each bit of state.

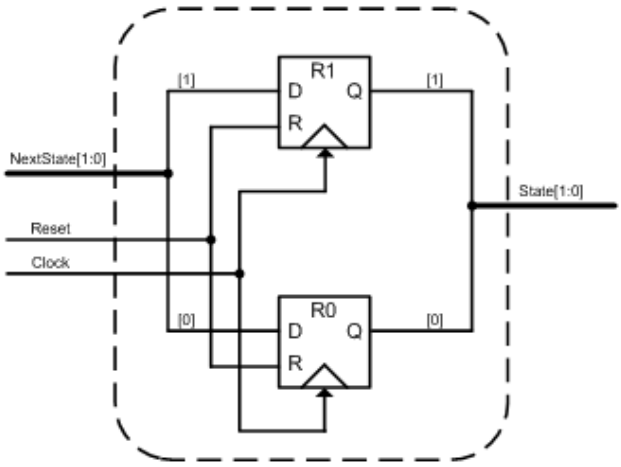


Figure 6 Implementation of a positive-edge triggered flip flop with an asynchronous reset.

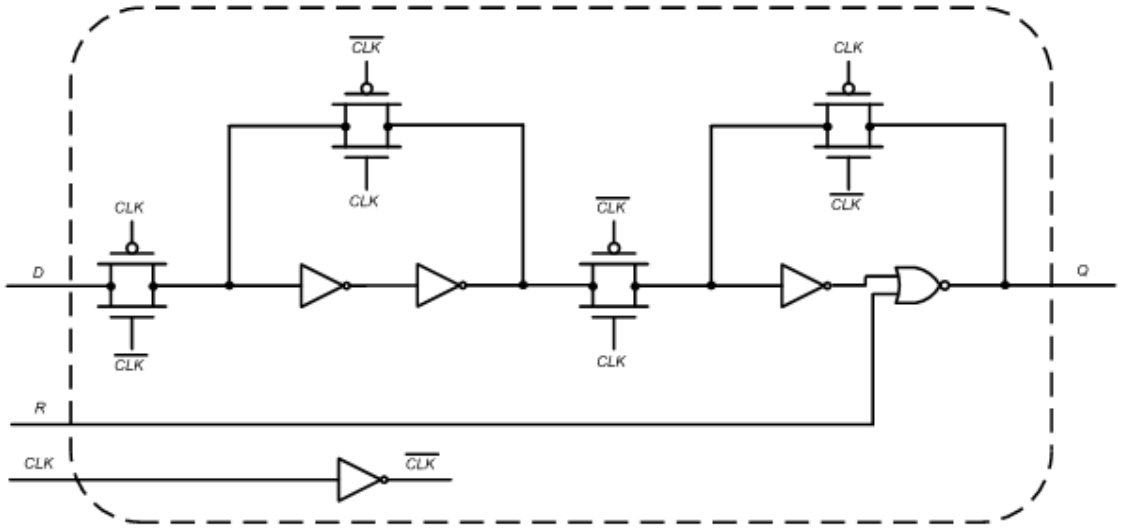
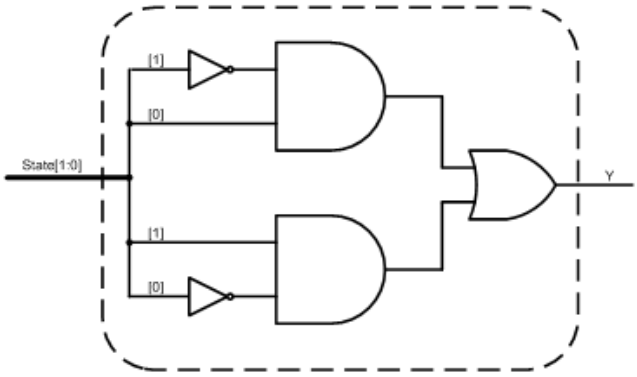


Figure 7 Implementation of the output logic



Assume that it has 1K words of instruction memory and 1K words of data memory. The instruction memory can hold any sequence of valid instructions. Suppose we want to describe this processor as an FSM (i.e. draw out a state transition diagram). How many state bubbles are in the STD? Note: don't bother to actually draw the STD.

This problem can be done by just enumerating through all the possibilities of values each state element can take.

PC: There are 1K words inside instruction memory, thus, there are $1024 = 2^{10}$ different possible values for the PC.

Data Memory: 1K words inside data memory, each word can take up 2^{32} different values, thus $(2^{32})^{1024} = 2^{32 \cdot 1024}$ possible values for the data memory.

Instruction Memory: This is a tricky one, since we only have 7 possible instructions. Each instruction is also limited by the different possibilities of each field. Lets begin by looking at each one.

add: The RS, RT, and RD fields can each take on 2^5 different values. Thus there are $2^{5 \cdot 3} = 2^{15}$ different **add** instructions.

Table 1 Truth table for the NextState function

ST_1	ST_0	**A**	**B**	NS_1	NS_0
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Figure 8 Implementation of the NextState logic

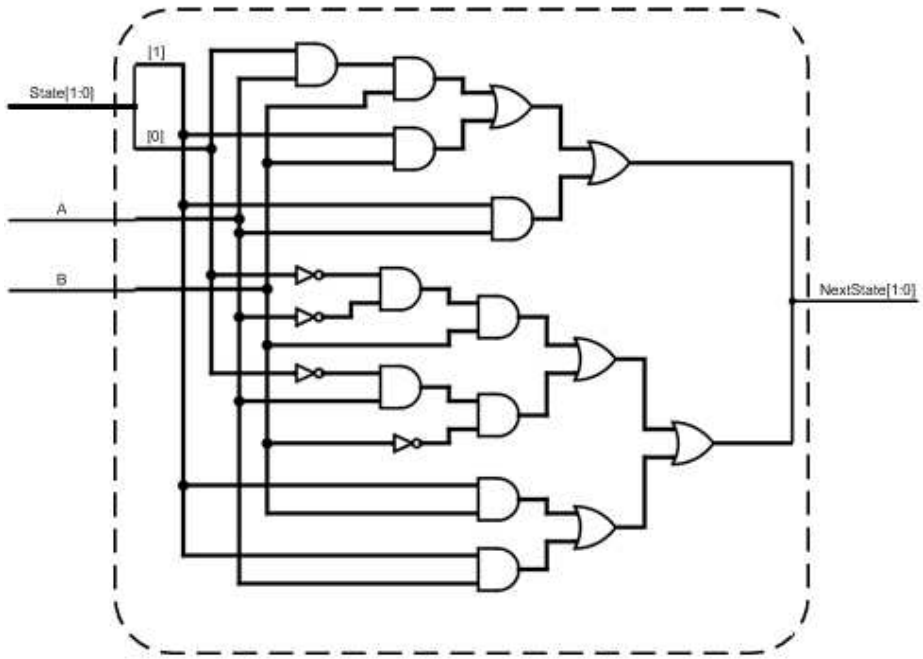


Figure 9 Implementation of an OR (left) and AND (right) gate

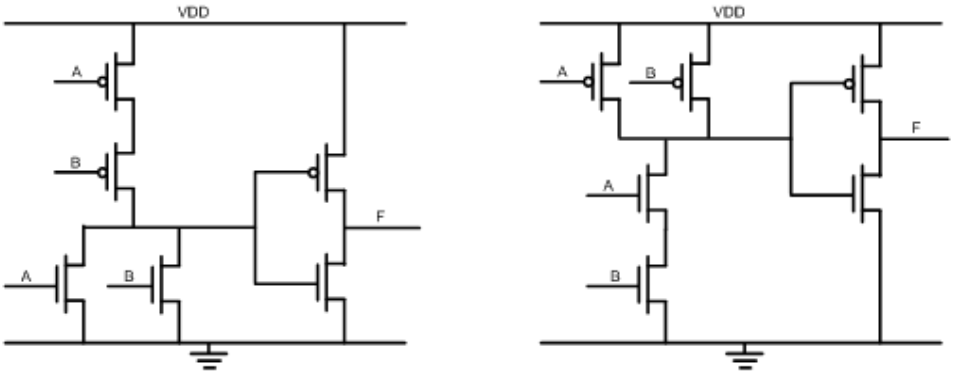
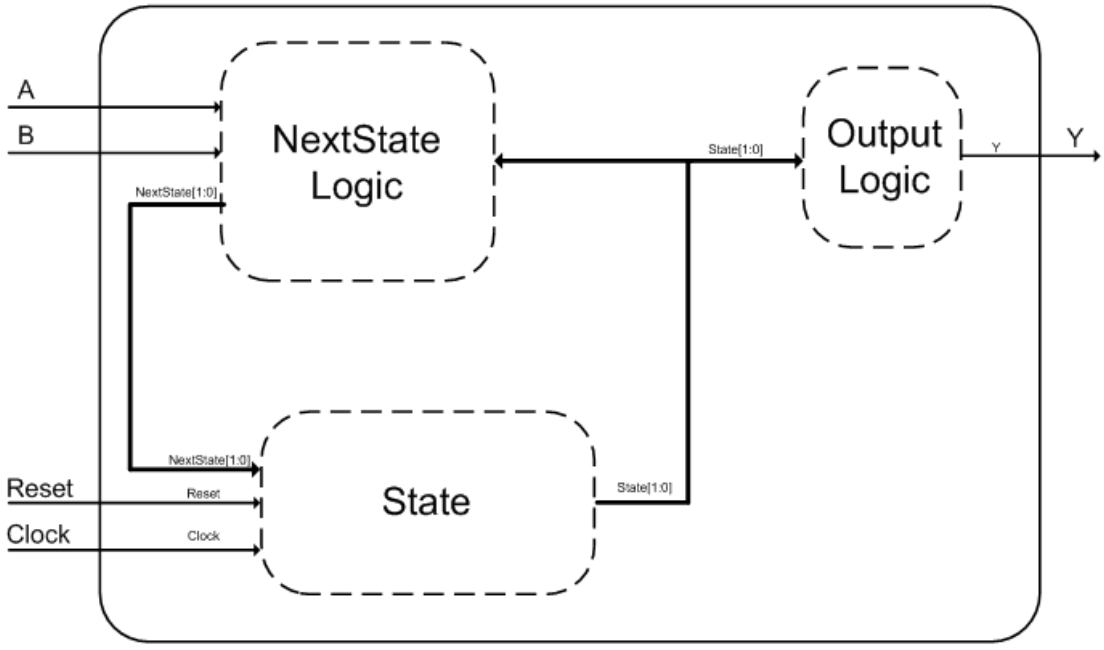


Figure 10 Implementation of The FSM



sub: Same as **add**, 2^{15}.

or: Same as **add**, 2^{15}.

slt: Same as **add**, 2^{15}.

sw: The RS and RT fields can each take on 2^5 different values. The IMMED field can take on 2^{12} different values, since there are 1K data words, each of which is 4 bytes (**sw**, **lw** are byte-addressed. Thus, 2^{22} different **sw** instructions.

lw: Same as **sw**, 2^{22}.

beq: Again, RS and RT fields each take on 2^5 different values. However, remember that the branch offset is given in words. Thus, since there is only 1K of instruction memory, there are only 2^{10} possible values for its immediate, meaning only 2^{20} different **beq** instructions.

To get the number of possibilities for **each word** of instruction memory, we have to **sum** all the different possibilities of instructions together - $4 \cdot 2^{15} + 2 \cdot 2^{22} + 2^{20}$. Since there are 1K words, the number of different values that the instruction memory can take is $(2^{4 \cdot 2^{15} + 2 \cdot 2^{22} + 2^{20}})^{1024}$.

The number of state bubbles is just the number of different possibilities for all the state elements multiplied together, namely $2^{10} \cdot 2^{32 \cdot 1024} \cdot (2^{4 \cdot 2^{15} + 2 \cdot 2^{22} + 2^{20}})^{1024}$. This number is many orders of magnitude larger than the predicted total number of atoms in the universe!

7. DDCA 7.3

   (a) `sll`

   Modifications to the ALU: Figure 11.

   Modified ALU Control: Figure 12.

   Modified ALU Decoder: Figure 13.

   Modified Datapath: Figure 14.

Figure 11 ALU Modifications for DDCA 7.3(a)

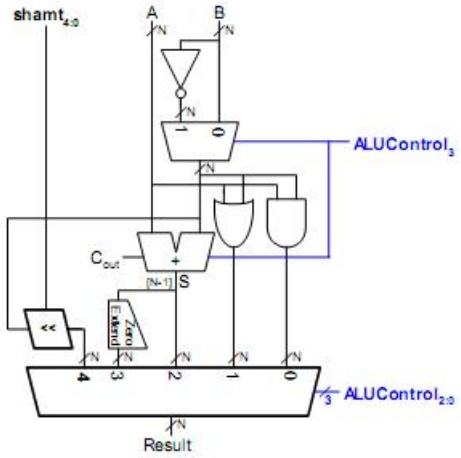


Figure 12 ALU Control Modifications for DDCA 7.3(a)

$ALUControl_{3:0}$	Function
0000	A AND B
0001	A OR B
0010	A + B
0011	not used
1000	A AND $\overline{B}$
1001	A OR $\overline{B}$
1010	A - B
1011	SLT
0100	SLL

   (b) `lui`

   Modified Decoder: Figure 15.

   Modified Datapath: Figure 16.

   (c) `slti`

   Modified ALU Decoder: Figure 17.

   Modified Main Decoder: Figure 18.

Figure 13 ALU Decoder Modifications for DDCA 7.3(a)

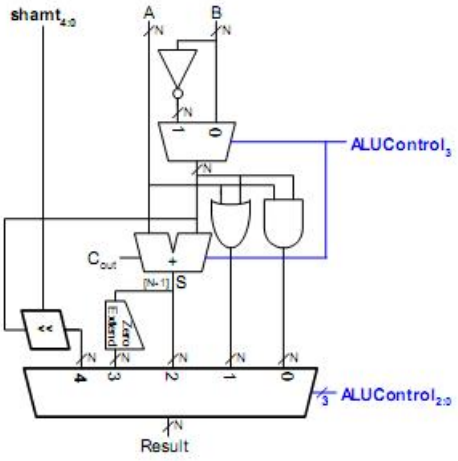


Figure 14 Datapath Modifications for DDCA 7.3(a)

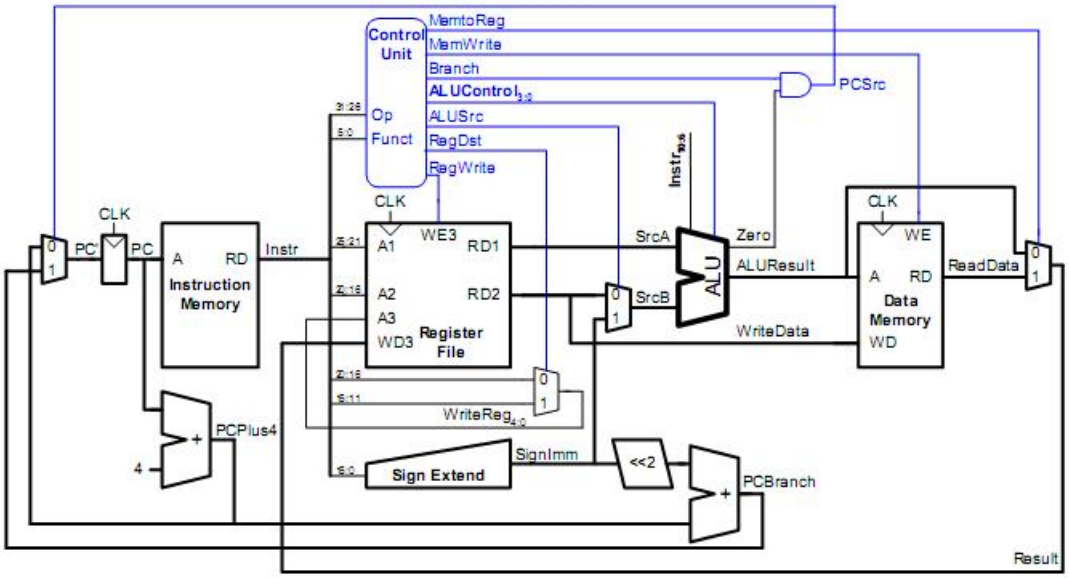


Figure 15 Decoder Modifications for DDCA 7.3(b)

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	00	0	0	0	10
lw	100011	1	0	01	0	0	1	00
sw	101011	0	X	01	0	1	X	00
beq	000100	0	X	00	1	0	X	01
lui	**001111**	**1**	**0**	**10**	**0**	**0**	**0**	**00**

Figure 16 Datapath Modifications for DDCA 7.3(b)

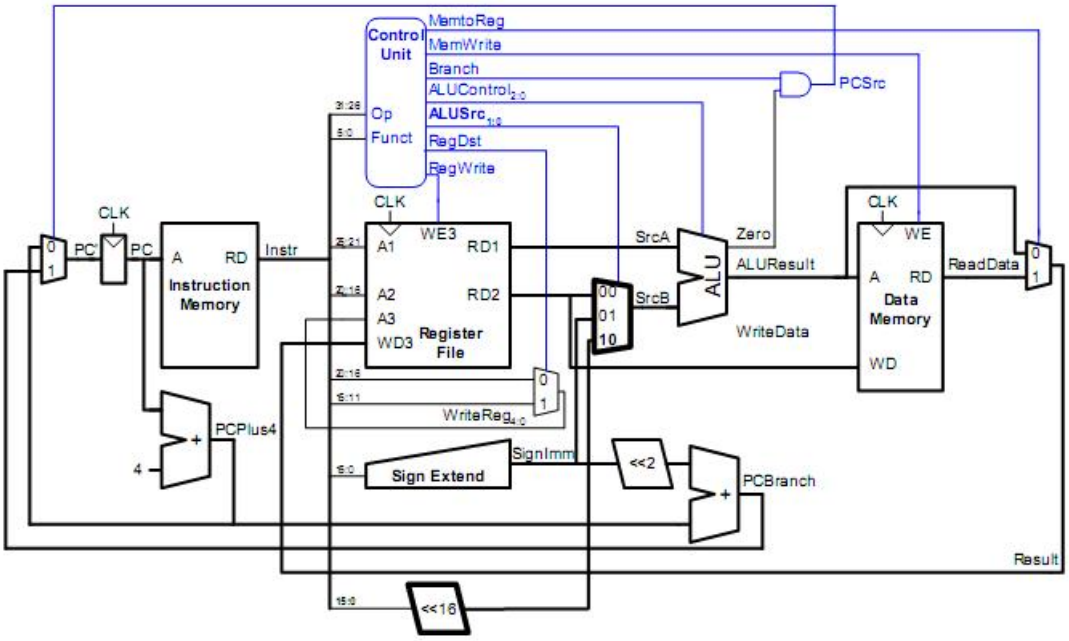


Figure 17 ALU Decoder Modifications for DDCA 7.3(c)

ALUOp	Funct	ALUControl
00	X	010 (add)
01	X	110 (subtract)
10	100000 (add)	010 (add)
10	100010 (sub)	110 (subtract)
10	100100 (and)	000 (and)
10	100101 (or)	001 (or)
10	101010 (slt)	111 (set less than)
11	**X**	**111 (set less than)**

Figure 18 Main Decoder Modifications for DDCA 7.3(c)

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
slti	**001010**	**1**	**0**	**1**	**0**	**0**	**0**	**11**

(d) `blez`

Modified ALU: Figure 19.

Modified Datapath: Figure 20.

Modified Decoder: Figure 21.

Figure 19 ALU Modifications for DDCA 7.3(d)

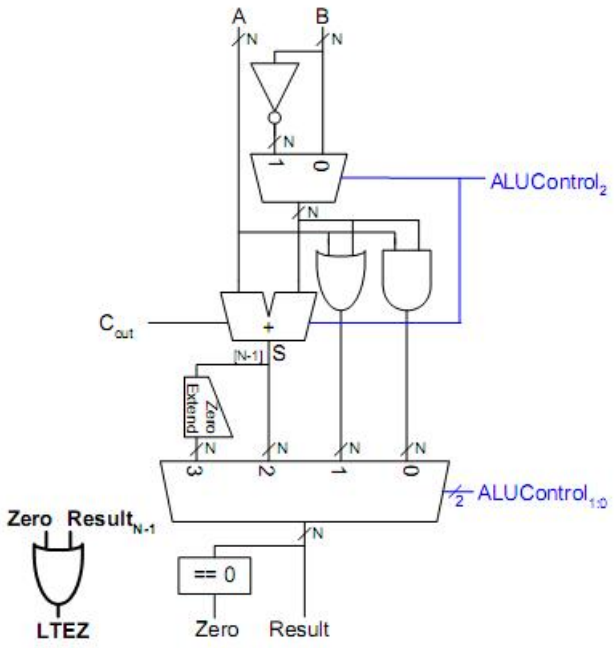


Figure 20 Datapath Modifications for DDCA 7.3(d)

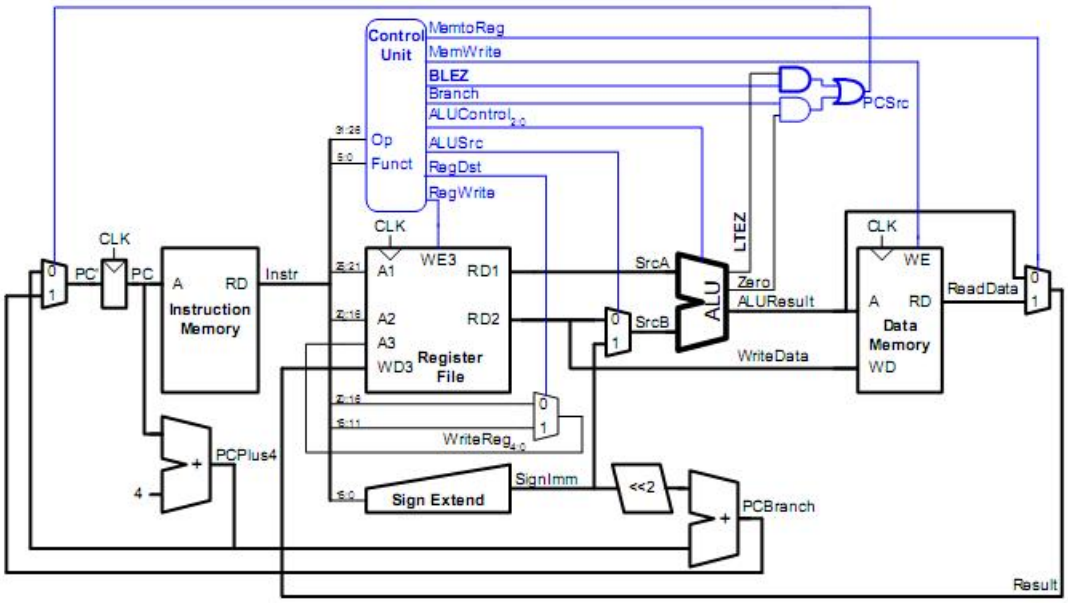


Figure 21 Decoder Modifications for DDCA 7.3(d)

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	BLEZ
R-type	000000	1	1	0	0	0	0	10	**0**
`lw`	100011	1	0	1	0	0	1	00	**0**
`sw`	101011	0	X	1	0	1	X	00	**0**
`beq`	000100	0	X	0	1	0	X	01	**0**
`blez`	**000110**	**0**	**X**	**0**	**0**	**0**	**X**	**01**	**1**

(e) `jal`

Modified Datapath: Figure 22.

Modified Decoder: Figure 23.

Figure 22 Datapath Modifications for DDCA 7.3(e)

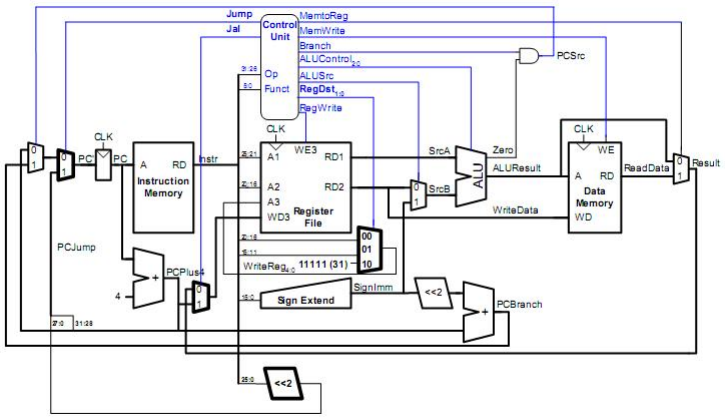


Figure 23 Decoder Modifications for DDCA 7.3(e)

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump	Jal
R-type	000000	1	**01**	0	0	0	0	10	**0**	**0**
`lw`	100011	1	**00**	1	0	0	1	00	**0**	**0**
`sw`	101011	0	**XX**	1	0	1	X	00	**0**	**0**
`beq`	000100	0	**XX**	0	1	0	X	01	**0**	**0**
`addi`	001000	1	**00**	1	0	0	0	00	**0**	**0**
`j`	000010	0	**XX**	X	X	0	X	XX	**1**	**0**
`jal`	**000011**	**1**	**10**	**X**	**X**	**0**	**X**	**XX**	**1**	**1**

(f) `lh`

Modified Datapath: Figure 24.

Modified Decoder: Figure 25.

Figure 24 Datapath Modifications for DDCA 7.3(f)

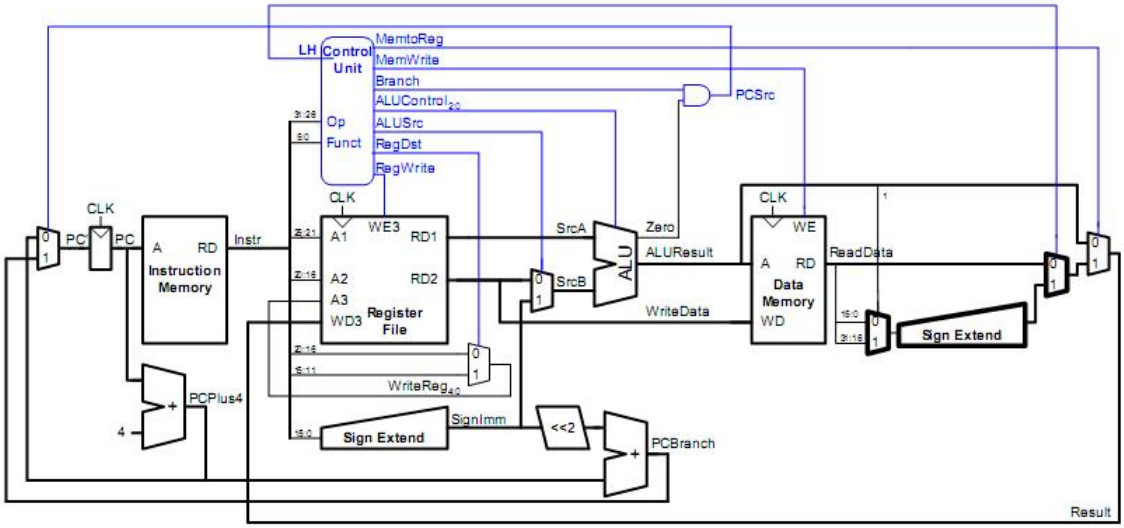


Figure 25 Decoder Modifications for DDCA 7.3(f)

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	LH
R-type	000000	1	01	0	0	0	0	10	**0**
`lw`	100011	1	00	1	0	0	1	00	**0**
`sw`	101011	0	XX	1	0	1	X	00	**0**
`beq`	000100	0	XX	0	1	0	X	01	**0**
`lh`	**100001**	**1**	**00**	**1**	**0**	**0**	**1**	**00**	**1**

8. DDCA 7.4

   Modified Datapath: Figure 26.

   Modified Decoder: Figure 27.

Figure 26 Datapath Modifications for DDCA 7.4

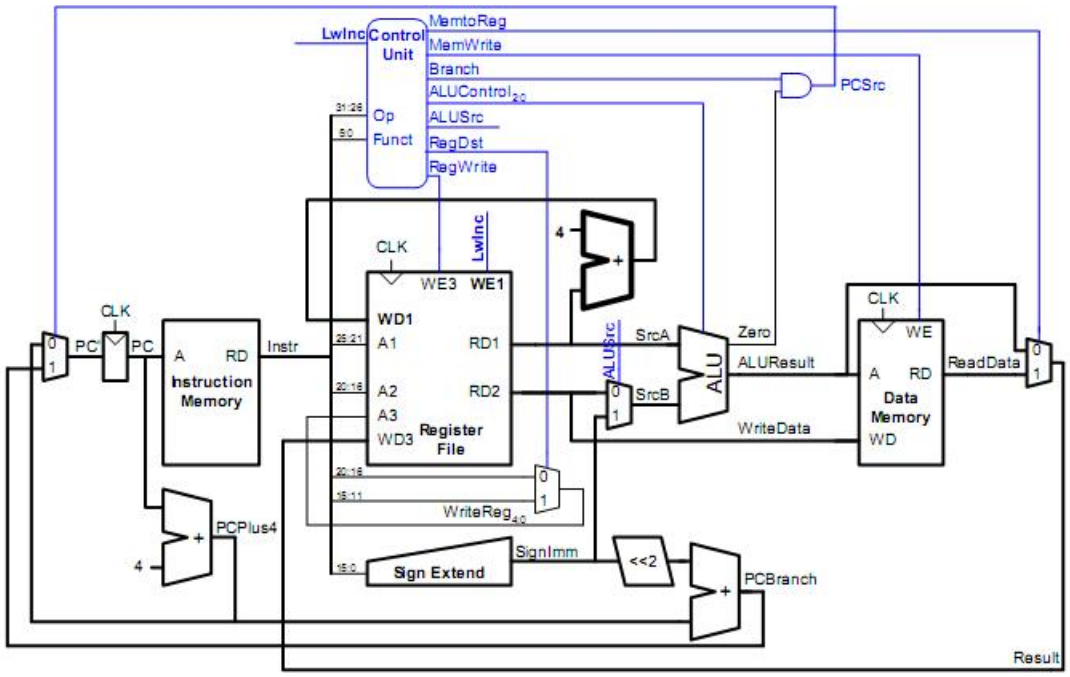


Figure 27 Decoder Modifications for DDCA 7.4

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Lwinc
R-type	000000	1	1	0	0	0	0	10	**0**
`lw`	100011	1	0	1	0	0	1	00	**0**
`sw`	101011	0	X	1	0	1	X	00	**0**
`beq`	000100	0	X	0	1	0	X	01	**0**
`lwinc`		**1**	**0**	**1**	**0**	**0**	**1**	**00**	**1**

9. DDCA 7.10

   Modified Processor Module: Figure 28.

   Modified Controller Module: Figure 29.

   Modified Decoder Module: Figure 30.

   Modified ALU Decoder Module: Figure 31.

   Modified Datapath Module: Figure 32.

   Additional Module: Figure 33.

   Modified ALU Module: Figure 34.

Figure 28 Processor Module Modifications for DDCA 7.10

```
module mipssingle(input         clk, reset,
                  output [31:0] pc,
                  input  [31:0] instr,
                  output        memwrite,
                  output [31:0] aluresult, writedata,
                  input  [31:0] readdata);

  wire        memtoreg;
  wire [1:0]  alusrc;  // LUI
  wire [1:0]  regdst;  // JAL
  wire        regwrite, jump, pcsrc, zero;
  wire [3:0]  alucontrol;  // SLL
  wire        ltez;  // BLEZ
  wire        jal;   // JAL
  wire        lh;    // LH

  controller c(instr[31:26], instr[5:0], zero,
               memtoreg, memwrite, pcsrc,
               alusrc, regdst, regwrite, jump,
               alucontrol,
               ltez,  // BLEZ
               jal,   // JAL
               lh);   // LH
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluresult, writedata, readdata,
              ltez,  // BLEZ
              jal,   // JAL
              lh);   // LH
endmodule
```

Figure 29 Controller Module Modifications for DDCA 7.10

```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc,
                  output [1:0] alusrc,      // LUI
                  output [1:0] regdst,      // JAL
                  output       regwrite,
                  output       jump,
                  output [3:0] alucontrol,  // SLL
                  input        ltez,        // BLEZ
                  output       jal,         // JAL
                  output       lh);         // LH

  wire [1:0] aluop;
  wire       branch;
  wire       blez;  // BLEZ

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             aluop, blez, jal, lh);  // BLEZ, JAL, LH
  aludec  ad(funct, aluop, alucontrol);

  // BLEZ
  assign pcsrc = (branch & zero) | (blez & ltez);

endmodule
```

Figure 30 Decoder Module Modifications for DDCA 7.10

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch,
               output [1:0] alusrc, // LUI
               output [1:0] regdst, // JAL
               output       regwrite,
               output       jump,
               output [1:0] aluop,
               output       blez,   // BLEZ
               output       jal,    // JAL
               output       lh);    // LH

  // increase control width for LUI, BLEZ, JAL, LH
  reg [13:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop,
          blez,   // BLEZ
          jal,    // JAL
          lh}     // LH
          = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 14'b10100000010000;
//Rtype
      6'b100011: controls <= 14'b10001001000000;
//LW
      6'b101011: controls <= 14'b00001010000000;
//SW
      6'b000100: controls <= 14'b00000100001000;
//BEQ
      6'b001000: controls <= 14'b10001000000000;
//ADDI
      6'b000010: controls <= 14'b00000000100000;
//J
      6'b001010: controls <= 14'b10001000011000;
//SLTI
      6'b001111: controls <= 14'b10010000000000;
//LUI
      6'b000110: controls <= 14'b00000000001100;
//BLEZ
      6'b000011: controls <= 14'b11000000100010;
//JAL
      6'b100001: controls <= 14'b10001001000001;
//LH
     default:   controls <= 14'bxxxxxxxxxxxxxx; //???
    endcase
endmodule
```

Figure 31 ALU Decoder Module Modifications for DDCA 7.10

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol);
                  // increase to 4 bits for SLL

  always @(*)
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add
      2'b01: alucontrol <= 4'b1010;  // sub
      2'b11: alucontrol <= 4'b1011;  // slt
      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0010; // ADD
          6'b100010: alucontrol <= 4'b1010; // SUB
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
          6'b000000: alucontrol <= 4'b0100; // SLL
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

Figure 32 Datapath Module Modifications for DDCA 7.10

```
module datapath(input         clk, reset,
                input         memtoreg, pcsrc,
                input [1:0]   alusrc,    // LUI
                input [1:0]   regdst,    // JAL
                input         regwrite, jump,
                input  [3:0]  alucontrol, // SLL
                output        zero,
                output [31:0] pc,
                input  [31:0] instr,
                output [31:0] aluresult, writedata,
                input  [31:0] readdata,
                output        ltez,  // BLEZ
                input         jal,   // JAL
                input         lh);   // LH

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
  wire [31:0] upperimm;  // LUI
  wire [31:0] srca, srcb;
  wire [31:0] result;
  wire [31:0] writeresult;  // JAL
  wire [15:0] half;         // LH
  wire [31:0] signhalf, memdata;     // LH

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 writeresult,
                 srca, writedata);

  mux2 #(32)  wamux(result, pcplus4, jal,
                    writeresult);  // JAL
  mux3 #(5)   wrmux(instr[20:16], instr[15:11], 5'd31,
                    regdst, writereg);  // JAL

  // hardware to support LH
  mux2 #(16)lhmux1(readdata[15:0],
                   readdata[31:16],
                   aluresult[1], half);  // LH
  signext     lhse(half, signhalf);         // LH
  mux2 #(32)  lhmux2(readdata, signhalf, lh,
                     memdata); // LH

  mux2 #(32)  resmux(aluresult, memdata, memtoreg,
                     result); // LH
  signext     se(instr[15:0], signimm);
  upimm       ui(instr[15:0], upperimm);  // LUI

  // ALU logic
  mux3 #(32)  srcbmux(writedata, signimm,
                      upperimm, alusrc,
                      srcb);      // LUI
  alu         alu(srca, srcb, alucontrol,
                  instr[10:6],  // SLL
                  aluresult, zero,
                  ltez); // BLEZ
endmodule
```

Figure 33 Additional Module for DDCA 7.10

```
// upimm module needed for LUI
module upimm(input  [15:0] a,
             output [31:0] y);

  assign y = {a, 16'b0};
endmodule

// mux3 needed for LUI
module mux3 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2,
              input  [1:0]       s,
              output [WIDTH-1:0] y);

  assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

Figure 34 ALU Module Modifications for DDCA 7.10

```
module alu(input [31:0] A, B,
           input [3:0] F, input [4:0] shamt, // SLL
           output reg [31:0] Y, output Zero,
           output ltez);  // BLEZ

  wire [31:0] S, Bout;

  assign Bout = F[3] ? ~B : B;
  assign S = A + Bout + F[3];  // SLL

  always @ ( * )
    case (F[2:0])
      3'b000: Y <= A & Bout;
      3'b001: Y <= A | Bout;
      3'b010: Y <= S;
      3'b011: Y <= S[31];
      3'b100: Y <= (Bout << shamt);  // SLL
    endcase

  assign Zero = (Y == 32'b0);
  assign ltez = Zero | S[31];  // BLEZ

endmodule
```