Solutions for Chapter 5 Exercises

5.1 Combinational logic only: a, b, c, h, i

Sequential logic only: f, g, j

Mixed sequential and combinational: d, e, k

5.2

a. RegWrite = 0: All R-format instructions, in addition to `lw`, will not work because these instructions will not be able to write their results to the register file.

b. ALUop1 = 0: All R-format instructions except subtract will not work correctly because the ALU will perform subtract instead of the required ALU operation.

c. ALUop0 = 0: `beq` instruction will not work because the ALU will perform addition instead of subtraction (see Figure 5.12), so the branch outcome may be wrong.

d. Branch (or PCSrc) = 0: `beq` will not execute correctly. The branch instruction will always be not taken even when it should be taken.

e. MemRead = 0: `lw` will not execute correctly because it will not be able to read data from memory.

f. MemWrite = 0: `sw` will not work correctly because it will not be able to write to the data memory.

5.3

a. RegWrite = 1: `sw` and `beq` should not write results to the register file. `sw` (`beq`) will overwrite a random register with either the store address (branch target) or random data from the memory data read port.

b. ALUop0 = 1: `lw` and `sw` will not work correctly because they will perform subtraction instead of the addition necessary for address calculation.

c. ALUop1 = 1: `lw` and `sw` will not work correctly. `lw` and `sw` will perform a random operation depending on the least significant bits of the address field instead of addition operation necessary for address calculation.

d. Branch = 1: Instructions other than branches (`beq`) will not work correctly if the ALU Zero signal is raised. An R-format instruction that produces zero output will branch to a random address determined by its least significant 16 bits.

e. MemRead = 1: All instructions will work correctly. (Data memory is always read, but memory data is never written to the register file except in the case of `lw`.)

f. MemWrite = 1: Only `sw` will work correctly. The rest of instructions will store their results in the data memory, while they should not.

5.7 No solution provided.

5.8 A modification to the datapath is necessary to allow the new PC to come from a register (Read data 1 port), and a new signal (e.g., JumpReg) to control it through a multiplexor as shown in Figure 5.42.

A new line should be added to the truth table in Figure 5.18 on page 308 to implement the `jr` instruction and a new column to produce the JumpReg signal.

5.9 A modification to the data path is necessary (see Figure 5.43) to feed the shamt field (instruction[10:6]) to the ALU in order to determine the shift amount. The instruction is in R-Format and is controlled according to the first line in Figure 5.18 on page 308.

The ALU will identify the `sll` operation by the ALUop field.

Figure 5.13 on page 302 should be modified to recognize the opcode of `sll`: the third line should be changed to 1X1X0000 0010 (to discriminate the `add` and `ssl` functions), and a new line, inserted, for example, 1X0X0000 0011 (to define `sll` by the 0011 operation code).

5.10 Here one possible `lui` implementation is presented:

This implementation doesn't need a modification to the datapath. We can use the ALU to implement the shift operation. The shift operation can be like the one presented for Exercise 5.9, but will make the shift amount as a constant 16. A new line should be added to the truth table in Figure 5.18 on page 308 to define the new shift function to the function unit. (Remember two things: first, there is no funct field in this command; second, the shift operation is done to the immediate field, not the register input.)

RegDst = 1: To write the ALU output back to the destination register (`$rt`).

ALUSrc = 1: Load the immediate field into the ALU.

MemtoReg = 0: Data source is the ALU.

RegWrite = 1: Write results back.

MemRead = 0: No memory read required.

MemWrite = 0: No memory write required.

Branch = 0: Not a branch.

ALUOp = 11: `sll` operation.

This ALUOp (11) can be translated by the ALU as `shl,ALUI1,16` by modifying the truth table in Figure 5.13 in a way similar to Exercise 5.9.

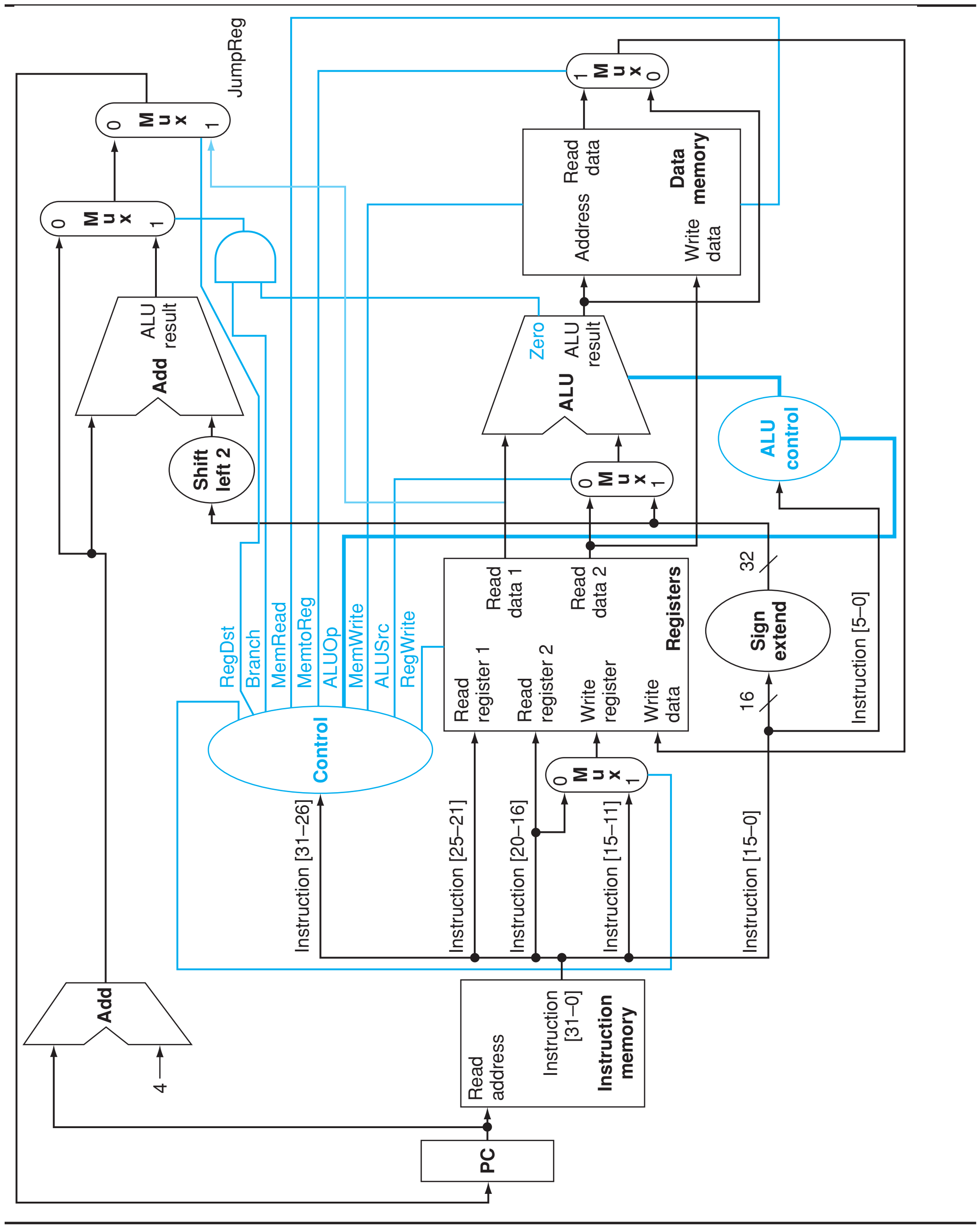
PC
Read address
Instruction [31–0]
Instruction memory
Add
4
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign extend
16
32
Shift left 2
Add
ALU result
ALU
Zero
ALU result
ALU control
Address
Write data
Read data
Data memory
JumpReg
M u x
0
1

FIGURE 5.42

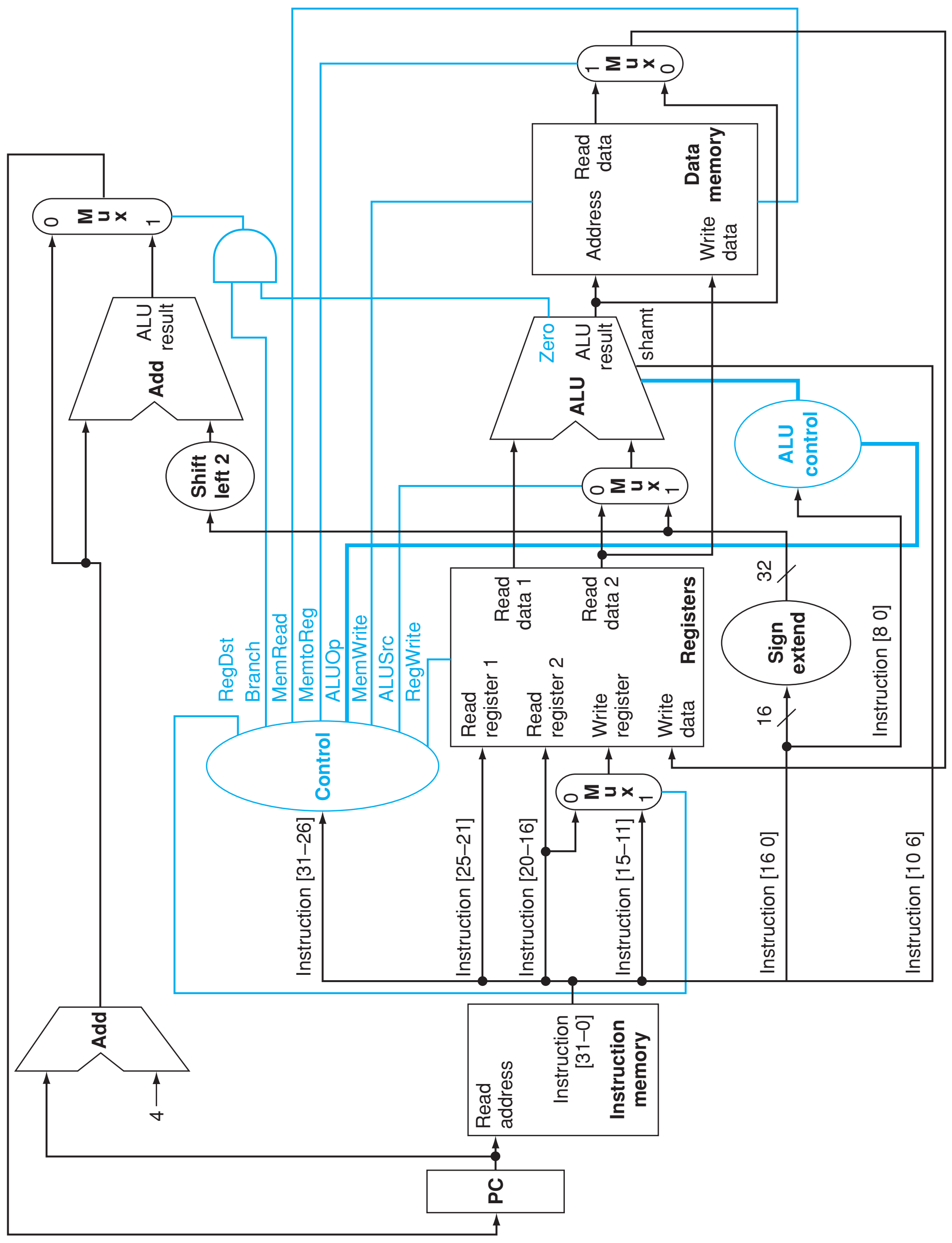
PC
Read address
Instruction [31–0]
Instruction memory
Add
4
Add
ALU result
Shift left 2
M u x
0
1
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [16 0]
Instruction [10 6]
Instruction [8 0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
32
Sign extend
ALU control
ALU
Zero
ALU result
shamt
Address
Read data
Write data
Data memory

FIGURE 5.43

5.11 A modification is required for the datapath of Figure 5.17 to perform the autoincrement by adding 4 to the `$rs` register through an incrementer. Also we need a second write port to the register file because two register writes are required for this instruction. The new write port will be controlled by a new signal, "Write 2", and a data port, "Write data 2." We assume that the Write register 2 identifier is always the same as Read register 1 (`$rs`). This way "Write 2" indicates that there is second write to register file to the register identified by "Read register 1," and the data is fed through Write data 2.

A new line should be added to the truth table in Figure 5.18 for the `l_inc` command as follows:

RegDst = 0: First write to `$rt`.

ALUSrc = 1: Address field for address calculation.

MemtoReg = 1: Write loaded data from memory.

RegWrite = 1: Write loaded data into `$rt`.

MemRead = 1: Data memory read.

MemWrite = 0: No memory write required.

Branch = 0: Not a branch, output from the PCSrc controlled mux ignored.

ALUOp = 00: Address calculation.

Write2 = 1: Second register write (to `$rs`).

Such a modification of the register file architecture may not be required for a multiple-cycle implementation, since multiple writes to the same port can occur on different cycles.

5.12 This instruction requires two writes to the register file. The only way to implement it is to modify the register file to have two write ports instead of one.

5.13 From Figure 5.18, the MemtoReg control signal looks identical to both signals, except for the don't care entries which have different settings for the other signals. A don't care can be replaced by any signal; hence both signals can substitute for the MemtoReg signal.

Signals ALUSrc and MemRead differ in that `sw` sets ALSrc (for address calculation) and resets MemRead (writes memory: can't have a read and a write in the same cycle), so they can't replace each other. If a read and a write operation can take place in the same cycle, then ALUSrc can replace MemRead, and hence we can eliminate the two signals MemtoReg and MemRead from the control system.

Insight: MemtoReg directs the memory output into the register file; this happens only in loads. Because `sw` and `beq` don't produce output, they don't write to the register file (Regwrite = 0), and the setting of MemtoReg is hence a don't care. The important setting for a signal that replaces the MemtoReg signal is that it is set for `lw` (Mem->Reg), and reset for R-format (ALU->Reg), which is the case for the ALUSrc (different sources for ALU identify `lw` from R-format) and MemRead (`lw` reads memory but not R-format).

5.14 swap \$rs,\$rt can be implemented by

```
addi $rd,$rs,0

addi $rs,$rt,0

addi $rt,$rd,0
```

if there is an available register `$rd`

or

```
sw $rs,temp($r0)

addi $rs,$rt,0

lw $rt,temp($r0)
```

if not.

Software takes three cycles, and hardware takes one cycle. Assume *Rs* is the ratio of swaps in the code mix and that the base CPI is 1:

Average MIPS time per instruction = $Rs * 3 * T + (1 - Rs) * 1 * T = (2Rs + 1) * T$

Complex implementation time = $1.1 * T$

If swap instructions are greater than 5% of the instruction mix, then a hardware implementation would be preferable.

5.27 `l_incr $rt,Address($rs)` can be implemented as

```
lw $rt,Address($rs)

addi $rs,$rs,1
```

Two cycles instead of one. This time the hardware implementation is more efficient if the load with increment instruction constitute more than 10% of the instruction mix.

5.28 Load instructions are on the critical path that includes the following functional units: instruction memory, register file read, ALU, data memory, and register file write. Increasing the delay of any of these units will increase the clock period of this datapath. The units that are outside this critical path are the two adders used for PC calculation (PC + 4 and PC + Immediate field), which produce the branch outcome.

Based on the numbers given on page 315, the sum of the the two adder's delay can tolerate delays up to 400 more ps.

Any reduction in the critical path components will lead to a reduction in the clock period.

5.29

a. RegWrite = 0: All R-format instructions, in addition to `lw`, will not work because these instructions will not be able to write their results to the register file.

b. MemRead = 0: None of the instructions will run correctly because instructions will not be fetched from memory.

c. MemWrite = 0: `sw` will not work correctly because it will not be able to write to the data memory.

d. IRWrite = 0: None of the instructions will run correctly because instructions fetched from memory are not properly stored in the IR register.

e. PCWrite = 0: Jump instructions will not work correctly because their target address will not be stored in the PC.

f. PCWriteCond = 0: Taken branches will not execute correctly because their target address will not be written into the PC.

5.30

a. RegWrite = 1: Jump and branch will write their target address into the register file. `sw` will write the destination address or a random value into the register file.

b. MemRead = 1: All instructions will work correctly. Memory will be read all the time, but IRWrite and IorD will safeguard this signal.

c. MemWrite = 1: All instructions will not work correctly. Both instruction and data memories will be written over by the contents of register B.

d. IRWrite = 1: `lw` will not work correctly because data memory output will be translated as instructions.

e. PCWrite = 1: All instructions except jump will not work correctly. This signal should be raised only at the time the new PC address is ready (PC + 4 at cycle 1 and jump target in cycle 3). Raising this signal all the time will corrupt the PC by either ALU results of R-format, memory address of `lw`/`sw`, or target address of conditional branch, even when they should not be taken.

f. PCWriteCond = 1: Instructions other than branches (`beq`) will not work correctly if they raise the ALU's Zero signal. An R-format instruction that produces zero output will branch to a random address determined by their least significant 16 bits.

5.31 RegDst can be replaced by $\overline{\text{ALUSrc}}$, $\overline{\text{MemtoReg}}$, $\overline{\text{MemRead}}$, ALUop1.

MemtoReg can be replaced by $\overline{\text{RegDst}}$, ALUSrc, MemRead, or $\overline{\text{ALUOp1}}$.

Branch and ALUOp0 can replace each other.

5.32 We use the same datapath, so the immediate field shift will be done inside the ALU.

1. Instruction fetch step: This is the same (IR <= Memory[PC]; PC <= PC + 4)

2. Instruction decode step: We don't really need to read any register in this stage if we know that the instruction in hand is a `lui`, but we will not know this before the end of this cycle. It is tempting to read the immediate field into the ALU to start shifting next cycle, but we don't yet know what the instruction is. So we have to perform the same way as the standard machine does.

A <= 0 ($r0); B <= $rt; ALUOut <= PC + (sign-extend(immediate field));

3. Execution: Only now we know that we have a `lui`. We have to use the ALU to shift left the low-order 16 bits of input 2 of the multiplexor. (The sign extension is useless, and sign bits will be flushed out during the shift process.)

ALUOut <= {IR[15-0],16(0)}

4. Instruction completion: Reg[IR[20-16]] = ALUOut.

The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle the FSM will recognize the opcode. We add the Op='lui', a new transition condition from state 1 to a new state 10. In this state we perform the left shifting of the immediate field: ALUSrcA = x, ALUSrcB = 10, ALUOp = 11 (assume this means left shift of ALUSrcB). State 10 corresponds to cycle 3. Cycle 4 will be translated into a new state 11, in which RegDst = 0, RegWrite, MemtoReg = 0. State 11 will make the transition back to state 0 after completion.

As shown above the instruction execution takes 4 cycles.

5.33 This solution can be done by modifying the data path to extract and shift the immediate field outside the ALU. Once we recognize the instruction as `lui` (in cycle 2), we will be ready to store the immediate field into the register file the next cycle. This way the instruction takes 3 cycles instead of the 4 cycles of Exercise 5.26.

1. Instruction fetch step: Unchanged.
2. Instruction decode: Also unchanged, but the immediate field extraction and shifting will be done in this cycle as well.

3. Now the final form of the immediate value is ready to be loaded into the register file. The MemtoReg control signal has to be modified in order to allow its multiplexor to select the immediate upper field as the write data source. We can assume that this signal becomes a 2-bit control signal, and that the value 2 will select the immediate upper field.

Figure 5.44 plots the modified datapath.

The first two cycles are identical to the FSM of Figure 5.38. By the end of the second cycle, the FSM will recognize the opcode. We add the Op = 'lui', a new transition condition from state 1 to a new state 10. In this state we store the immediate upper field into the register file by these signals: RedDst = 0, RegWrite, MemtoReg = 2. State 10 will make the transition back to state 0 after its completion.

5.34 We can use the same datapath.

1. Instruction fetch: Unchanged (IR <= Memory[PC]; PC<= PC + 4).
2. Instruction decode: Unchanged (A <= Reg[IR[25-21]]; B<=REG[IR[20-16]]; ALUOut<=PC+(sign-extend(IR[15-0])<<2).
3. Load immediate value from memory (MDR <= Memory[PC]; PC <= PC + 4).
4. Complete instruction (Reg[IR[20-16]] (dependent on instruction format) <= MDR).

The first two cycles are identical to the FSM of Figure 5.38.

We add the Op='ldi', a new transition condition from state 1 to a new state 10. In this state we fetch the immediate value from memory into the MDR: MemRead, ALUSrcA = 0, IorD = 0, MDWrite, ALUSrcB = 01, ALUOp = 00, PCWrite, PCSource = 00.

FSM then makes the transition to another new state 11.

In this state we store the MDR into the register file by these signals: RedDst = 0 (actually depends on the instruction format), RegWrite, MemtoReg = 1.

State 11 will make the transition back to state 0 after its completion.

Four cycles to complete this instruction, in which we have two instruction memory accesses.

5.35 Many solutions are possible. In all of them, a multiplexor will be needed as well as a new control signal (e.g., RegRead) to select which register is going to be read (i.e., using `IR[25-11]` or `IR[20-16]`). One simple solution is simply to add a write signal to A and break up state 1 into two states, in which A and B are read. It is possible to avoid adding the write signal to A if B is read first. Then A is

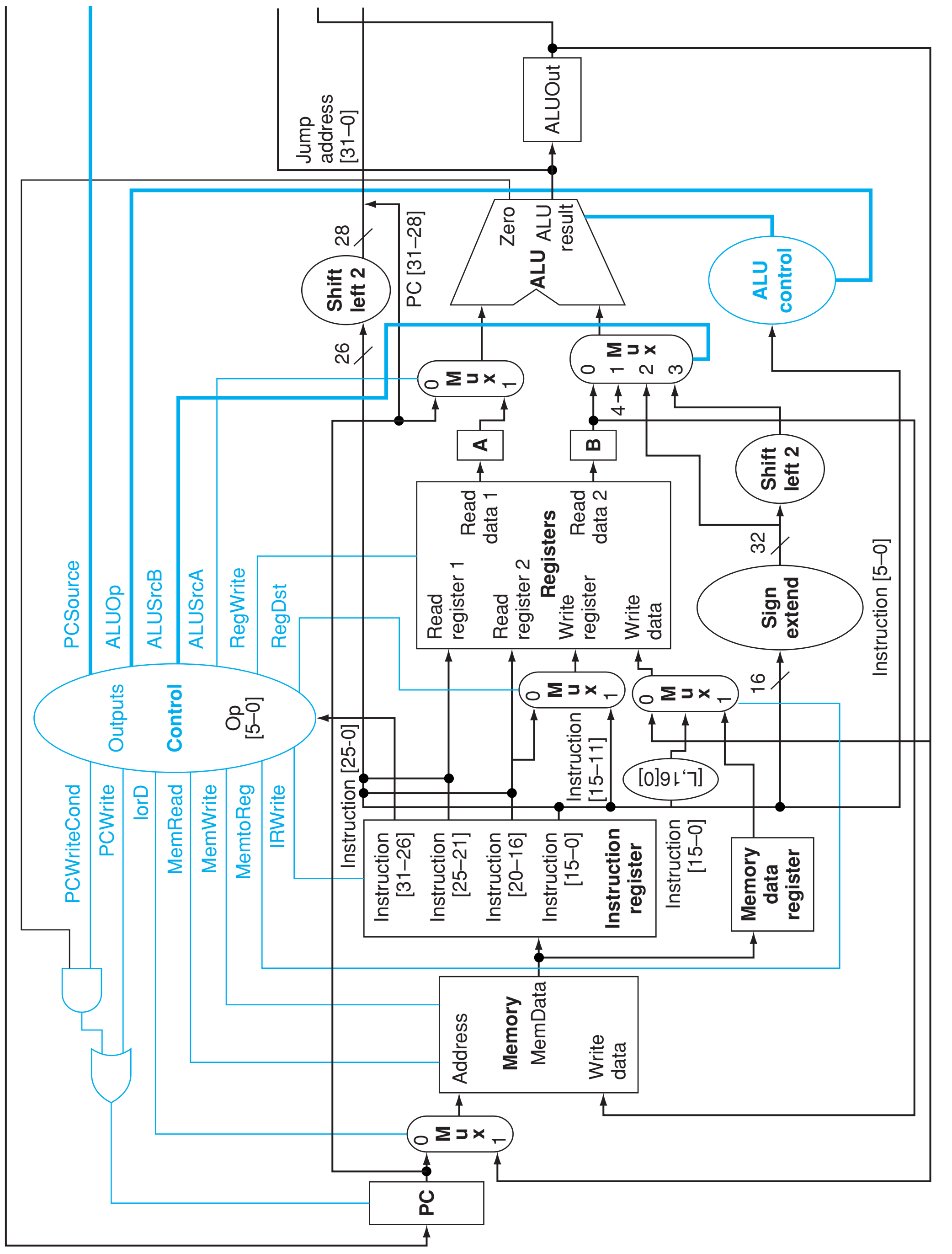
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Outputs
Control
Op [5–0]
PC
Memory
Address
MemData
Write data
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Memory data register
Instruction [25–0]
Instruction [15–11]
[L, 16[0]
Registers
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
A
B
Sign extend
16
32
Shift left 2
Instruction [5–0]
ALU control
ALU
Zero
ALU result
ALUOut
Shift left 2
26
28
PC [31–28]
Jump address [31–0]
M u x
0
1
2
3
4

FIGURE 5.44

read and RegRead is held stable (because A always writes). Alternatively, you could decide to read A first because it may be needed to calculate an address. You could then postpone reading B until state 2 and avoid adding an extra cycle for the load and store instructions. An extra cycle would be needed for the branch and R-type instructions.

5.36 Effective CPI = Sum(operation frequency * operation latency)

MIPS = Frequency/CPIeffective

Instruction	Frequency	M1	M2	M3
Loads CPI	25%	5	4	3
Stores CPI	13%	4	4	3
R-type CPI	47%	4	3	3
Branch/jmp CPI	15%	3	3	3
Effective CPI		4.1	3.38	3
MIPS		976	946	933

From the results above, the penalty imposed on frequency (for all instructions) exceeds the gains attained through the CPI reduction. M1 is the fastest machine.

The more the load instructions in the instruction mix, the more the CPI gain we can get for the M2 and M3 machines. In the extreme case we have all instructions loads, M1 MIPS = 800, M2 MIPS = 300, and M3 MIPS = 933.3, so M3 becomes the best machine in such a case.

5.37 Effective CPI = Sum(operation frequency * operation latency)

MIPS = Frequency/CPIeffective

Instruction	Frequency	2.8 GHz CPI	5.6 GHz CPI	6.4 GHz CPI
Loads CPI	26%	5	6	7
Stores CPI	10%	4	5	6
R-type CPI	49%	4	4	5
Branch/jmp CPI	15%	3	3	4
Effective CPI		4.11	4.47	5.47
MIPS		1167.9	1250	1170.0

The two-cycle implementation increases the frequency, which benefits all instructions, and penalizes only loads and stores. The performance improvement is 7% relative to the original implementation.

Further increase of the clock frequency by increasing the instruction fetch time into two cycles will penalize all instructions and will reduce the performance to about the same as that of the 4.8 GHz base performance. Such implementation hurts the CPI more than the gain it brings through frequency increase and should not be implemented.

5.38

```
slt   $t4, $zero, $t3
      beg  $t4, $zero, exit
cmpr: lw   $t4, 0($t1)
      lw   $t5, 0($t5)
      bne  $t4, $t5, done
      addi $t3, $t3, -1
      addi $t1, $t1, 4
      addi $t2, $t2, 4
      bne  $t3, $zero, cmpr
exit  addi $t1, $zero, $zero
done:
```

To compare two 100-work blocks we'll perform at most one sit 200 loads, 300 adds, and 201 branches = 803 instructions (if the two blocks are equal). Using this chapter's multicycle implementation, this will take 4 cycles for sit 1000 cycles for loads, 1200 cycles for adds, and 603 cycles for branches. The total cycles = 2811 cycles.

5.39 No solution provided.

5.49 No solution provided.

5.50 The exception cause can be represented through the status "cause" register, which records the reason code for the exception. The instruction position at which the exception occur is identified by saving it in the Exception Program Counter (EPC) register.

Execution can be restarted for some exceptions like overflow, system call request, or external I/O device interrupt by restarting execution at the EPC after handling the exception.

Other exceptions are not restartable and program has to terminate. Examples of this are invalid instructions (which can actually be restartable if defined as NOP by the hardware), power/hardware failure, and divide by zero. In such a case, an error message can be produced, and program termination takes place.

5.51

a. Divide by zero exception can be detected in the ALU in cycle 3, before executing the divide instruction.

b. Overflow can be hardware detected after the completion of the ALU operation. This is done in cycle 4 (see Figure 5.40)

c. Invalid opcode can be detected by the end of cycle 2 (see Figure 5.40).

d. This is an asynchronous exception event that can occur at any cycle. We can design this machine to test for this condition either at a specific cycle (and then the exception can take place only in a specific stage), or check in every cycle (and then this exception can occur at any processor stage).

e. Check for instruction memory address can be done at the time we update the PC. This can be done in cycle 1.

f. Check for data memory address can be done after address calculation at the end of cycle 3.

5.53 No solution provided.

5.57 No solution provided.

5.58 a) will assign the same value (2) to both A and B.

b) will swap A and B (A = 2 and B = 1).

5.59

```
module ALUControl (ALUOp, FuncCode, ALUCtl);
       input ALUOp[1:0], FuncCode[5:0];
                  output ALUCtl[3:0];
                     if(ALUOp == 2'b 00)
                        ALUCtl = 4'b 0010;
                     if(ALUOp == 2'b 01)
                        ALUCtl = 4'b 0110;
                     if(ALUOp == 2'b 10) begin
                        case(FuncCode)
                           6'b 100000: ALUCtl = 4'b 0010;
                           6'b 100010: ALUCtl = 4'b 0110;
                           6'b 100100: ALUCtl = 4'b 0000;
                           6'b 100101: ALUCtl = 4'b 0001;
```

```
                    6'b 101010: ALUCtl = 4'b 0111;
                    default     ALUCtl = 4'b xxxx;
                 endcase
             end
         endmodule
```

5.60

```
// Register File
module RegisterFile (Read1,Read2,Writereg,Writedata,Regwrite,
Data1Data2,clock);
 input [5:0] Read1,Read2,Writereg;  // the registers numbers to read
or write
 input [31:0] Writedata;     // data to write
 input RegWrite,             // The write control
       clock;                // the clock to trigger writes
 output [31:0] Data1, Data2; // the register values read;
 reg [31:0] RF [31:0];       // 32 registers each 32 bits long
 initial RF[0] = 32'h 00000000; // Initialize all registers to 0
 always begin
   Data1 <= RF[Read1]; Data2 <= RF[Read2];
   // write the register with new value if Regwrite is high
   @(negedge clock) if RegWrite then RF[Writereg] <= WriteData;
 end
endmodule

//ALU Control same as 5.30
module ALUControl (ALUOp, FuncCode, ALUCtl);
input ALUOp[1:0], FuncCode[5:0];
output ALUCtl[3:0];
   if(ALUOp == 2'b 00)
      ALUCtl = 4'b 0010;
   if(ALUOp == 2'b 01)
      ALUCtl = 4'b 0110;
   if(ALUOp == 2'b 10) begin
      case(funct)
         6'b 100000: ALUCtl = 4'b 0010;
         6'b 100010: ALUCtl = 4'b 0110;
         6'b 100100: ALUCtl = 4'b 0000;
         6'b 100101: ALUCtl = 4'b 0001;
         6'b 101010: ALUCtl = 4'b 0111;
         // .... Add more ALU control here
```

```
        default    ALUCtl = 4'b xxxx; //can report an error, or debug
information
      endcase
   end
endmodule

//ALU
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
input [3:0] ALUctl;
input [31:0] A,B;
output [31:0] ALUOut;
output Zero;
assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
always @(ALUctl, A, B) begin //reevaluate if these change
  case (Aluctl)
    0: ALUOut <= A & B;
    1: ALUOut <= A | B;
    2: ALUOut <= A + B;
    6: ALUOut <= A - B;
    7: ALUOut <= A < B ? 1:0;
    // .... Add more ALU operations here
   default: ALUOut <= X;   //can report an error, or debug information
  endcase
 end
endmodule

//2-to-1 Multiplexor
module Mult2to1 (In1,In2,Sel,Out);
input [31:0] In1, In2;
input Sel;
output [31:0] Out;
always @(In1, In2, Sel)
  case (Sel) //a 2->1 multiplexor
    0: Out <= In1;
    default: Out <= In2;
  endcase;
endmodule;
```

```
//This represents every thing in Figure 5.19 on page 309 except the
"control block"
//Which decodes the opcode, and generate the control signals
accordingly
module DataPath(start,RegDst,Branch,MemRead,MemtoReg,ALUOp,Mem-
Write,ALUSrc,RegWrite,opcode,clock)
input [31:0] start;      //PC initial value
input RegDst,Branch,MemRead,MemtoReg,
      ALUOp,MemWrite,ALUSrc,RegWrite,clock;
input [1:0] ALUOp;
output [5:0] opcode;
initial begin           //initialize PC and Memories
    PC = start;
    IMemory = PROGRAM;
    DMemory = DATA;
end
reg [31:0] PC, IMemory[0:1023], DMemory[0:1023];
wire [31:0] SignExtendOffset, PCOffset, PCValue, ALUResultOut,
    IAddress, DAddress, IMemOut, DmemOut, DWriteData, Instruction,
    RWriteData, DreadData, ALUAin, ALUBin;
wire [3:0] ALUctl;
wire Zero;
wire [5:0] WriteReg;
//Instruction fields, to improve code readability
wire [5:0]  funct;
wire [4:0]  rs, rt, rd, shamt;
wire [15:0] offset;

ALUControl alucontroller(ALUOp,Instruction[5:0],ALUctl);
    //ALU control
MIPSALU ALU(ALUctl, ALUAin, ALUBin, ALUResultOut, Zero);
RegisterFile REG(rs, rt, WriteReg,  RWriteData, ALUAin, DWriteData,
clock);
Mult2to1 regdst (rt, rd, RegDst, RegWrite),
     alusrc (DWriteData, SignExtendOffset, ALUSrc, ALUBin),
     pcsrc  (PC+4, PC+4+PCOffset, Branch & Zero, PCValue);
assign {opcode, rs, rt, rd, shamt, funct} = Instruction;
assign offset = Instruction[15:0];
assign SignExtendOffset = {16{offset[15]},offset}; //sign-extend
lower 16 bits;
assign PCOffset = SignExtendOffset << 2;
```

```
always @(negedge clock) begin
  Instruction = IMemory[PC];
  PC <= PCValue;
 end
 always @(posedge clock) begin
  if MemRead
      DreadData <= DMemory[ALUResultOut];
  else if MemWrite
      DMemory[ALUResultOut] <= DWriteData;
  end
 end

endmodule

module MIPS1CYCLE(start);
// Clock
reg clock; // clock is a register
initial clock = 0;
parameter LW = 6b 100011, SW = 6b 101011, BEQ=6b 000100;
input start;
wire [1:0] ALUOp;
wire [5:0] opcode;
wire [31:0] SignExtend;
wire RegDst,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite;
DataPath MIPSDP (start,RegDst,Branch,MemRead,MemtoReg,ALUOp,
MemWrite,ALUSrc,RegWrite,opcode,clock);
  //datapath control
always begin
 #1 clock = ~ clock;  //clock generation
  case(opcode)
    0:   {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,
ALUOp}= 9'b 100100010;//R-Format
    LW:  {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,
ALUOp}= 9'b 011110000;
    SW:  {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,MemWrite,Branch,
ALUOp}= 9'b x1x001000;
    BEQ: {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,Mem-
Write,Branch,ALUOp}= 9'b x0x000101;
    // .... Add more instructions here
    default: $finish;    // end simulation if invalid opcode
  endcase
end
endmodule
```

5.61 We implement the add shift functionality to the ALU using the Verilog code provided in B.22 in Appendix B. The 32-bit multiply execution takes 32 cycles to complete, so the instruction takes a total of 35 cycles. Assume the ALU control recognizes the multiply code correctly.

We follow the CD Verilog code, but we add the following:

```
case(state)
.
.
3: begin         //Execution starts at cycle 3
  state=4
  .
  .
  case(opcode==6'b 0)
    .
    .
    MPYU: begin
// issue load command to the multiplier
  {RegDst,ALUSrc,MemtoReg,RegWrite,MemRead,
MemWrite,Branch,ALUOp}= 9'b 1001000110;//R-Format  same
command. Alu should now recognize the Func Field
    end
  .
  .
35:          // After 32 cycles the multiplication
results are available in the 32-bit Product output of
the ALU. Write the high order and low order words in
this and the next cycle.
  case(opcode==6'b 0) case (IR[5:0])
    .
    .
    MPYU: begin
  Regs[hi]=RegH
    end
```

```
34:
  case(opcode==6'b 0) case (IR[5:0])
    .
    .
    MPYU: begin
  Regs[lo]=RegL
    end
end
```

5.62 We add the divide functionality to the ALU using the code of B.23. The rest of the solution is almost exactly the same as the answer to Exercise 5.61.

5.63 No solution provided

5.64 No solution provided.

5.65 No solution provided.

5.66 No solution provided.