ALU and Pipe-lined CPU Verilog Realization

1.ALU

This project report illustrates how ALU and the five-stage pipe-lined CPU are realized and imitated using the Verilog language. In the first task, we built an Arithmetic Logic Unit (ALU) supporting the following operations:

- add, addi, addu, addiu
- sub, subu
- and, andi, nor, or, ori, xor, xori
- beq, bne, slt, slti, sltiu, sltu
- lw, sw
- sll, sllv, srl, srlv, sra, srav

In this ALU unit, we have two registers, register A with address 00000, and register B with address 00001. Each operation is characterized by its opcode (instruction[31:26]) and (or) function code (instruction[5:0]), and the field rd, rt, rs will store the address of reg A and reg B, respectively. All other fields like sa, immediate, etc. will store their binary representations. Unused fields will be filled with 0.

Therefore, there are 3 inputs (a 32-bit instruction, a 32-bit reg A, a 32-bit reg B), and 2 outputs (a 32-bit register "result" storing the outcome of this ALU operation, and a 3-bit "flags"). The first bit of flags (flags[0]) is the overflow flag, which is 1 if overflow happens in when current operation is add, addi, sub, and 0 otherwise. The second bit of flags (flags[1]) is the negative flag, which is 1 if the condition is true in slt , slti , sltiu , and sltu, and 0 otherwise. The third bit of flags (flags[2]) is the zero flag, which is 1 if the equality sign in beq and bne holds, and 0 otherwise. We denote the ALU realization process in the below chart:

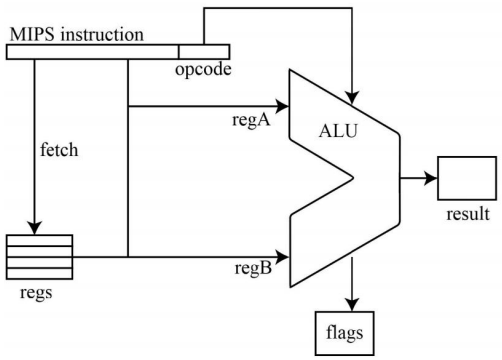


Chart 1. ALU implementation flow chart

We thus define the alu module in file alu.v. The module first extracts the instruction's opcode and funcode to determine which operation it is. It then executes that based on rd, rt, rs, sa, etc., fields, and put the outcome in "result". Meanwhile, it determines whether overflow, negative sign, and zero sign happen and sets the corresponding bit of "flags" to 1 if so.

```
8  module alu(input[31:0] instruction,
9      input[31:0] regA,regB,
10     output reg[31:0] result,
11     output reg[2:0] flags);
12     reg[5:0] opcode, func;
13     reg[31:0] reg0,reg1;
14     reg signed[31:0] reg2,reg3;
15     always @(*)begin
16         opcode = instruction[31:26];
17         func = instruction[5:0];
18         flags = 3'b000;
19         if(opcode==6'b000000 && func==6'b100000) begin //add
20             result = regA+regB;
```

Figure 1. module alu implementation

We also provide the testbench module called test_alu in file test_alu.v to test each of our implemented operations. All operations are tested in the single test_alu.v file, with each operation being tested after one another after 10 time unites time break. Specifically, we compare the correct outputs with the program outputs result and flags, and output “000failed” if they differ. If all operations are implemented correctly, we should only see “test done!” in the terminal. The testbench is run using *make test* in the terminal under the path “src/alu”. Testbench codes are:

```
module test_alu();
    reg[31:0] instruction;
    reg[31:0] regA, regB;
    wire[31:0] result;
    wire[2:0] flags;
    alu dut(.instruction(instruction), .regA(regA), .regB(regB), .result(result), .flags(flags));
    //check instruction one at a time
    initial begin
        instruction=32'b00000000000000001000000000100000; regA=32'h7fffffff; regB=1; #10;//check a
        if (result!=32'h80000000 || flags!=3'b001) $display("000 failed.");
        instruction=32'b00100000000000001111111111111110; regA=-1; regB=0; #10; //check addi
        if (result!=-3 || flags!=3'b000) $display("000 failed.");
```

Figure 2. testbench module test_alu implementation

Our first task in modelling the ALU ends here.

2. Five-stage Pipe-lined CPU

2.1 Modelling the five-stage pipe-lined CPU(test cases outcomes are talked in 2.2)

In the second task, we imitate a five-stage pipe-lined CPU’s working process. We realize it using four sub-modules and one top-module. The four sub-modules are IF(instruction fetch, corresponding to the first CPU cycle), REG(register read and write, corresponding to the second and last CPU cycle), ALU(corresponding to the third CPU cycle), and MeM (memory read and write, corresponding to the fourth CPU cycle). The top-module PipelinedCPU is used to instantiate all the four sub-modules using the input CLK(represent clock) from the testbench test_cpu and the sub-module themselves’ inputs and outputs.

In the following parts, we will provide illustration to the realization of each sub-module, the top-module, and the testbench, one by one. In the first sub-module IF, we fulfill the first CPU cycle, where CPU fetches one 32-bit instruction at the newly given PC address at each rising edge of the clock from the file CPU_instruction.bin.

```
module IF(input CLOCK,jump2,branch2,input[31:0]im,outp
  integer fd,c1,c2;
  integer pc=0;
  always @(posedge CLOCK)begin
    fd = $fopen("CPU_instruction.bin", "rb");
    if(branch2==1'b1)begin
      pc=(pc/33+im-1)*33;
    end
    if(jump2==1'b1)begin
```

Figure 3. Sub-module IF implementation

In the second sub-module REG, at each falling edge of the clock, CPU decodes the 32-bit instruction coming from the module IF into its opcode, function code, sa, signed-extended immediate, zero-extended immediate, target, rd, rt, rs, etc., and output them to the third sub-module ALU. This fulfills the second CPU cycle.

In the same module, at each rising edge of the clock, data coming from the third or fourth sub-module is written back to a specific register if needed. This fulfills the fifth CPU cycle.

```
19  module REG(input CLOCK,memread,memwrite,jump,branch,R,I,input[31:0]instru
20    reg[31:0] register[31:0];
21    integer j,a1,a2;
22    initial begin
23      for (j=0;j<32;j=j+1)begin
24        register[j]=0;
25      end
26    end
```

Figure 4. Sub-module REG implementation

In the third sub-module ALU, at each rising edge of the clock, we execute the corresponding operation recognized by the opcode and function code inputted from the second sub-module. Specifically, we consider arithmetic instructions add, addu, addi, addiu, sub, subu, logical instructions and, andi, nor, or, ori, xor, xori, shifting instructions sll, sllv, srl, srlv, sra, srav, branch/jump instructions beq, bne, slt, j, jr, jal, and data transfer instructions lw, sw. Corresponding processed outcomes are outputted in registers res (for arithmetic, logical, and shifting instructions), offset, jump, branch (for jump/branch instructions), loc1, loc2, memread, memwrite (for data transfer instructions). This fulfills the third CPU cycle.

```
82  module ALU(input CLOCK,memread2,input[5:0]opcode,funcode,input[31:0]rd,rt,rs,
83    always @(posedge CLOCK) begin
84      memread=1'b0;
85      memwrite=1'b0;//reset
86      jump=1'b0;
87      branch=1'b0;
88      offset=32'h00000000;
```

Figure 5. Sub-module ALU implementation

In the fourth sub-module MeM, at each falling edge of the clock, data from ALU is stored to the corresponding location in the memory data.bin (sw instruction) or CPU fetches a piece of data from the indicated location in the memory data.bin (lw instruction). This fulfills the fourth CPU cycle.

```
230  module MeM(input CLOCK,memread,memwrite,termi,input[31:0]loc
231    integer i,j,b1;
232    reg[31:0] memory[511:0];
233    initial begin
234      for(j=0;j<512;j=j+1)begin
235        memory[j]=0;
236      end
237    end
238    always @(negedge CLOCK) begin
239      if(memread==1'b1)begin//lw
240        rtloc3=rtloc2;
241        memread2=memread;
242        data=memory[loc1];
```

Figure 6. Sub-module MeM implementation

In the top-module PipelinedCPU, we instantiate all four sub-modules using the input clock from the testbench and the inputs and outputs generated by those sub-modules themselves as discussed above. This top-module thus serves as the five-stage CPU as a whole.

```
263  module PipelinedCPU(input CLK);
264    wire[31:0] INSTRUCTION,DATA,RD,RT,RS,IM,ZEROIM,TAR,RES,LOC1,LOC2,LOC
265    wire[5:0] OPCD,FUCD;
266    wire MEMR,MEMW,JP,BCH,TERMI,JP2,BCH2,R,I,MEMR2;
267    IF u_IF(
268      .CLOCK(CLK),
269      .im(IM),
270      .jump2(JP2),
271      .branch2(BCH2),
272      .instruction(INSTRUCTION),
273      .curpc(PC));
274    REG u_REG(
275      .CLOCK(CLK),
276      .memread(MEMR),
```

Figure 7. Top-module PipelinedCPU implementation

One last remaining component is the testbench module test_cpu implemented in the file test_cpu.v, which generates the clock signal, CLK, used in our top-module PipelinedCPU.

CLK changes to 1 and 0 periodically, creating rising edge and falling edge alternatively, so that our pipeline can work correspondingly.

```
Project3 > src > cpu > test_cpu.v
1  module test_cpu();
2      reg CLK;
3      PipelinedCPU dut(.CLK(CLK));
4      always begin
5          CLK=0; #10;
6          CLK=1; #10;
7      end
8  endmodule
```

Figure 8. Testbench module test_cpu implementation

2.2 Test cases results and data hazard handling methods

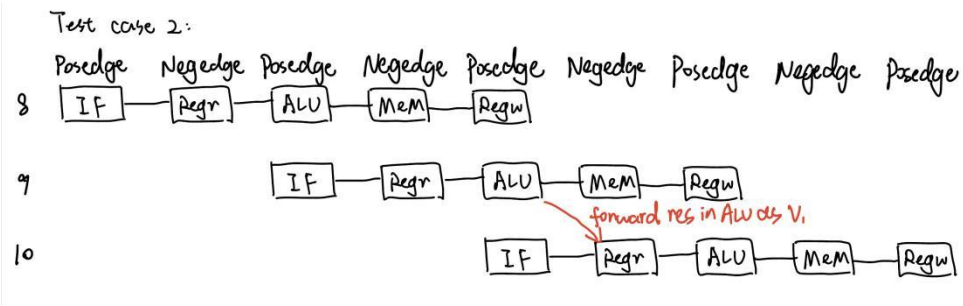
Our implemented CPU has passed all test cases, including the test case without any hazards (test case 1,5,6) and test cases with data hazards (test case 2,3,4,7,8). To run the program, we first need to paste the machine codes needed to be executed in CPU_instruction.bin, open the terminal under the path "src/cpu", and type "make test" in the terminal to run the program. The program output is stored in data.bin, which serves as the outputted CPU memory and can be compared with the correct ones (DATA_RAMx.txt, x=1,2,...,8).

Since we adopt the rising edge and falling edge of clocks for consecutive stages, we will have two stalls between each pipeline (as drawn below). Test case 1 is to test normal functions without any harzards, test case 4 is automatically correct due to the rising edge and falling edge design for Regr and Regw, test case 5 and 6 are to test normal branch and jump instructions without any hazards. We therefore illustrate the hazard handling methods for the remaining test cases (2,3,7,8):

Test case 2 (MEM_to_EX and WB_to_EX data hazards):

```
sub $v0, $a0, $a1 # 4 in $v0 //Line 8
sub $v1, $a1, $a0 # -4 in $v1 //Line 9
sub $a2, $v1, $v0 # -8 in $a2 (if the forward succeeds, $a2 should be 8)//Line 10
```

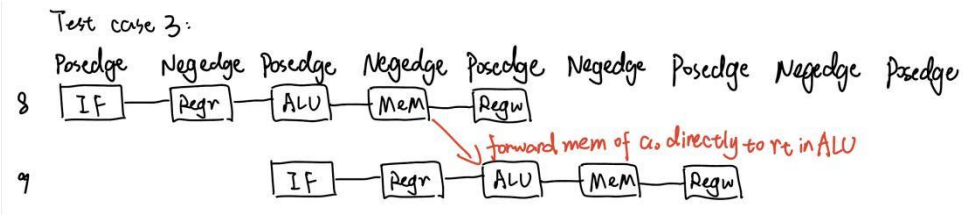
hazard



solution

Test case 3 (lw stall hazard):

Line 8: lw $a0, 4($zero) # $a0 should be 114
Line 9: sub $a1, $v1, $a0 # The hazard of lw stall happens hazard

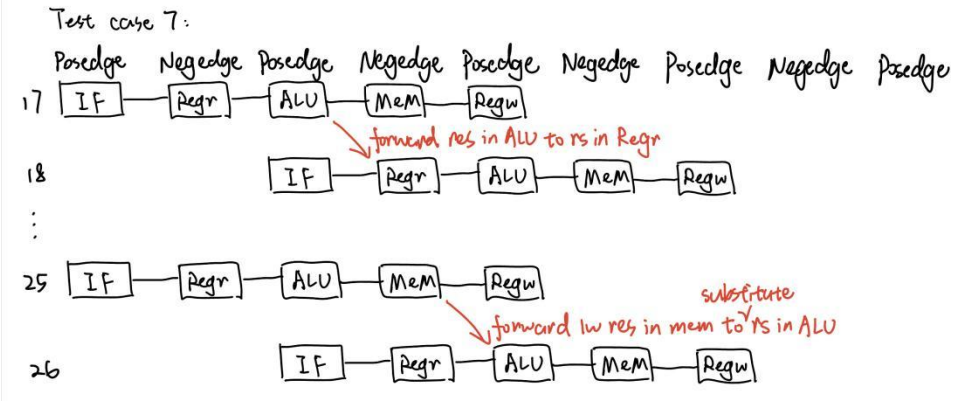


solution

Test case 7 (jr data hazards):

Line 17: addi $v0, $zero, 84
Line 18: jr $v0 # The first jr jr hazard 1

Line 25: lw $k0, 0($zero) # $k0 is 116
Line 26: jr $k0 # The second jr jr hazard 2

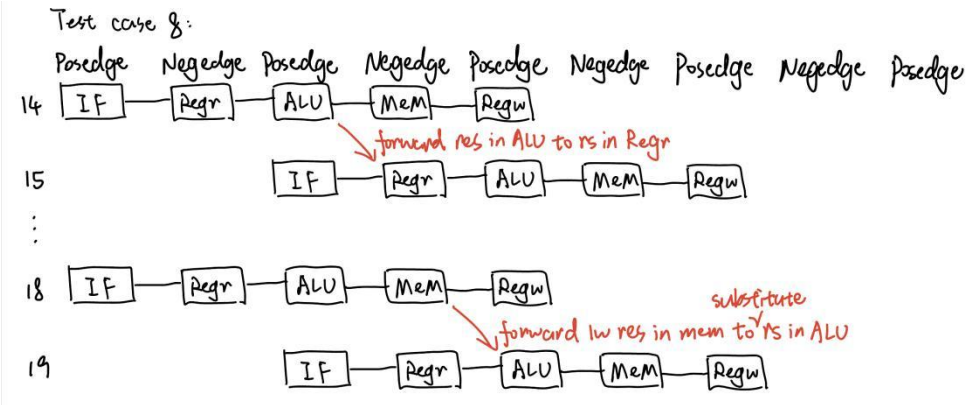


solution

Test case 8 (branch data hazards):

Line 14: slt $a1, $v0, $v1
Line 15: bne $a1, $a0, -5 hazard 1

Line 18: lw $t1, 8($zero)
Line 19: beq $t1, $t2, 2 hazard 2



solution