Lab 4: Pipelined Processor

Robert Chen

86266604

3/8/2020

Objective:

The pipelined processor can be broken down into its original 5 components (instruction fetch, instruction decode, execution, memory, writeback) in addition to intermediary registers between each stage that allow for hazard detection and forwarding. Using this pipelined implementation shown in Figure I, instructions can be carried out without any dependencies on the value of other instructions and registers. The result of one component is frequently used as an input to at least one or more other components, with the values stored in a wire, so that new instructions can be loaded in as the previous instruction is still being processed in a further stage.

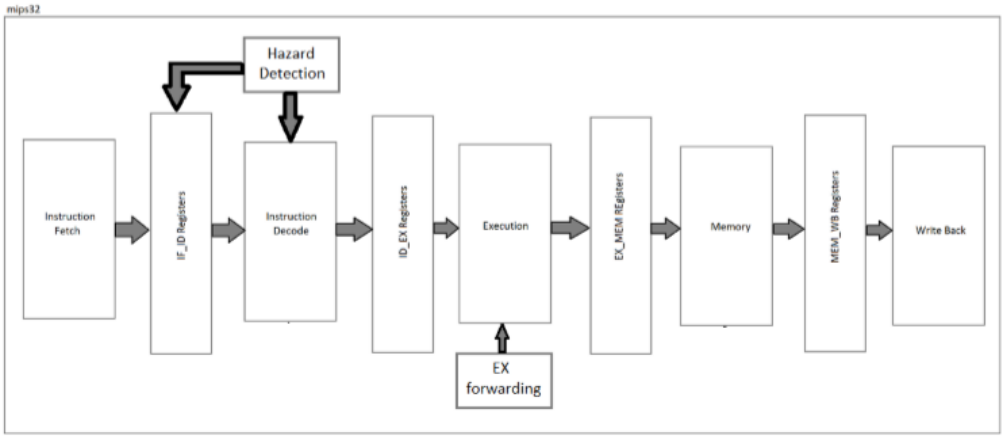


Figure I. Pipelined Processor Structure

Procedure:

IF: For the instruction fetch unit, 2-bit muxes are used to determine the next pc based on whether a branch or jump should be taken or not. Once the next pc is determined, the Data_Hazard input is checked where it returns the next pc if there is no data hazard, or returns the same unchanged pc if there is a data hazard. This is how the pc is “stalled” for one cycle to avoid a data hazard. Otherwise, the pc will just increment for pc+4 as normal and continue to the next instruction.

ID: For the instruction decode unit, the instruction is extracted through the control.v and register file units to know what operation to do and where to store the results. Output data from the memory-writeback unit is used as an input here to the register file. The control unit outputs are also checked against data hazard and control hazard from the forwarding unit to ensure that the values will only defined if there is no hazard.

EX: For the execution stage, the alu_in2_out output will get the value of the data in id_ex_reg2, the write_back_data, or the ex_mem_alu_result depending on the forwarding condition. The alu_result output will get the processed value of a and b, where a is either the data of id_ex_reg1, write_back_data, or ex_mem_alu_result, and b is the immediate value from the instruction. The selections are based on the alu_src output of control.v and the forwarding conditions.

Forwarding: For the forwarding unit, ForwardA and ForwardB can take values of 10 or 11, with 00 as default. These outputs are used in a 4-bit mux as a selector in the execution stage. Forwarding occurs when there is a potential data hazard.

WB: For the write back mux, the write_back_data (output) will be chosen between the alu result or the memory read data from the memory-writeback unit based on the mem-to-reg output. This result is displayed as the main result, and is also used again as an input to the decode stage.

Reg: For all intermediary registers between each stage, all the input wire data were concatenated into port d and outputted at port q. From there, the concatenated q output was extracted back into its individual wires to be used for the next stage. The individual output wires are named differently so that those data can be processed and the original wires can take on new values of the next instruction.

Simulation Results:

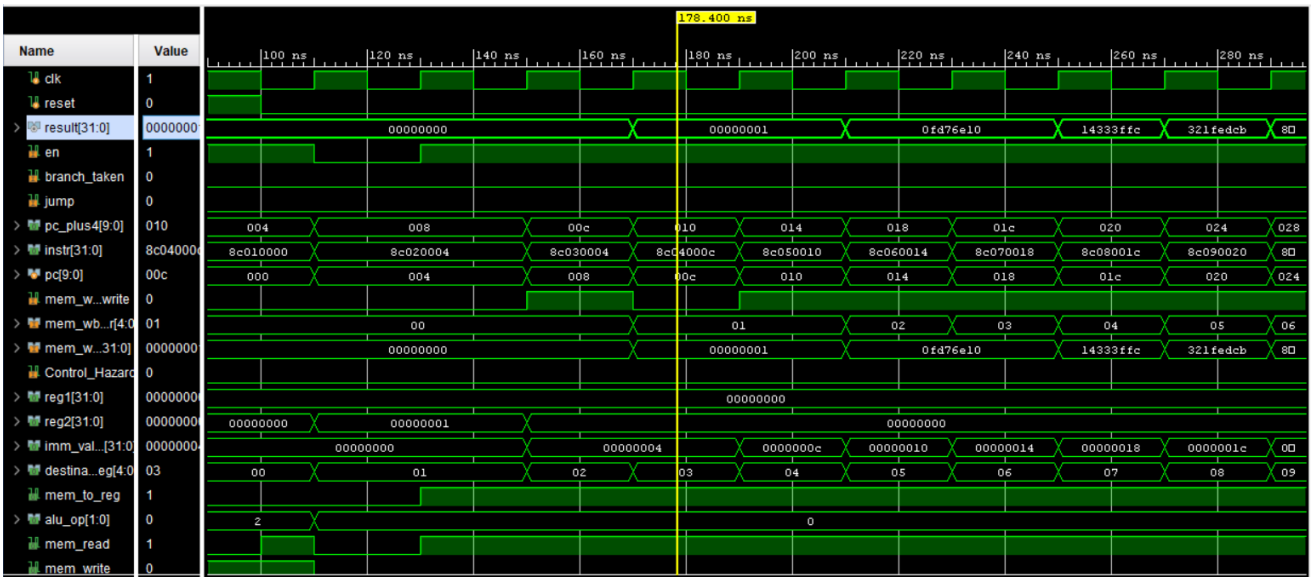
Testbench: (instructions 0-9)

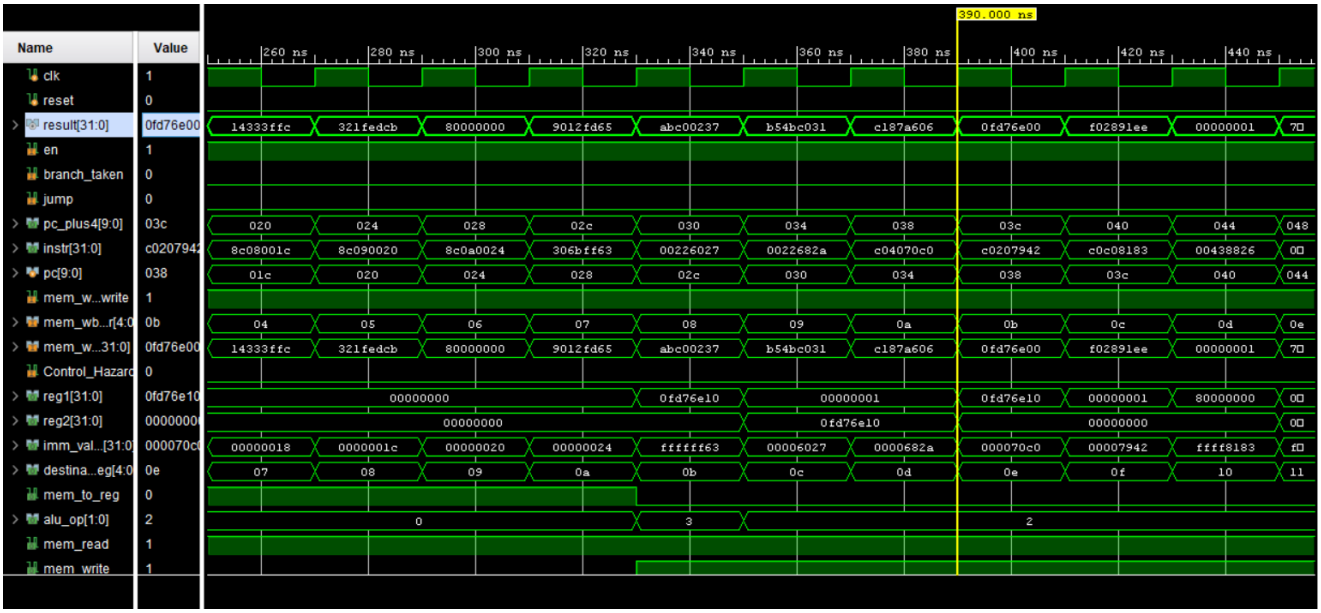
```
        // load to registers 1 to 10
                                                    // instruction          alu result in hex      register content       mem content

        rom[0]  = 32'b10001100000000010000000000000000; // r1  = mem[0]          0                     r1 = 00000001          -
        rom[1]  = 32'b10001100000000010000000000000100; // r2  = mem[4]          1                     r2 = 0fd76e10          -
        rom[2]  = 32'b10001100000000011000000000000100; // r3  = mem[4]          2                     r3 = 0fd76e10          -
        rom[3]  = 32'b10001100000001000000000000001100; // r4  = mem[12]         3                     r4 = 14333ffc          -
        rom[4]  = 32'b10001100000001010000000000010000; // r5  = mem[16]         4                     r5 = 321fedcb          -
        rom[5]  = 32'b10001100000001100000000000010100; // r6  = mem[20]         5                     r6 = 80000000          -
        rom[6]  = 32'b10001100000001110000000000011000; // r7  = mem[24]         6                     r7 = 9012fd65          -
        rom[7]  = 32'b10001100000010000000000000011100; // r8  = mem[28]         7                     r8 = abc00237          -
        rom[8]  = 32'b10001100000010010000000000100000; // r9  = mem[32]         8                     r9 = b54bc031          -
        rom[9]  = 32'b10001100000010100000000000100100; // r10 = mem[36]         9                     r10= c187a606          -
```

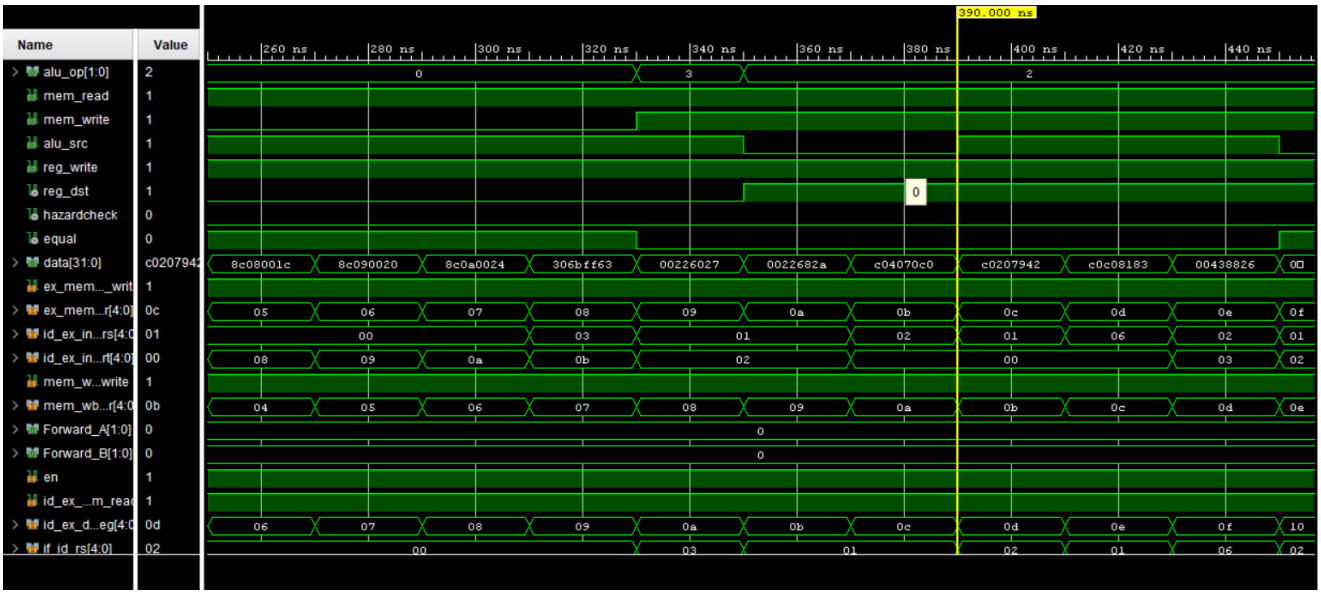
Waveform Data:

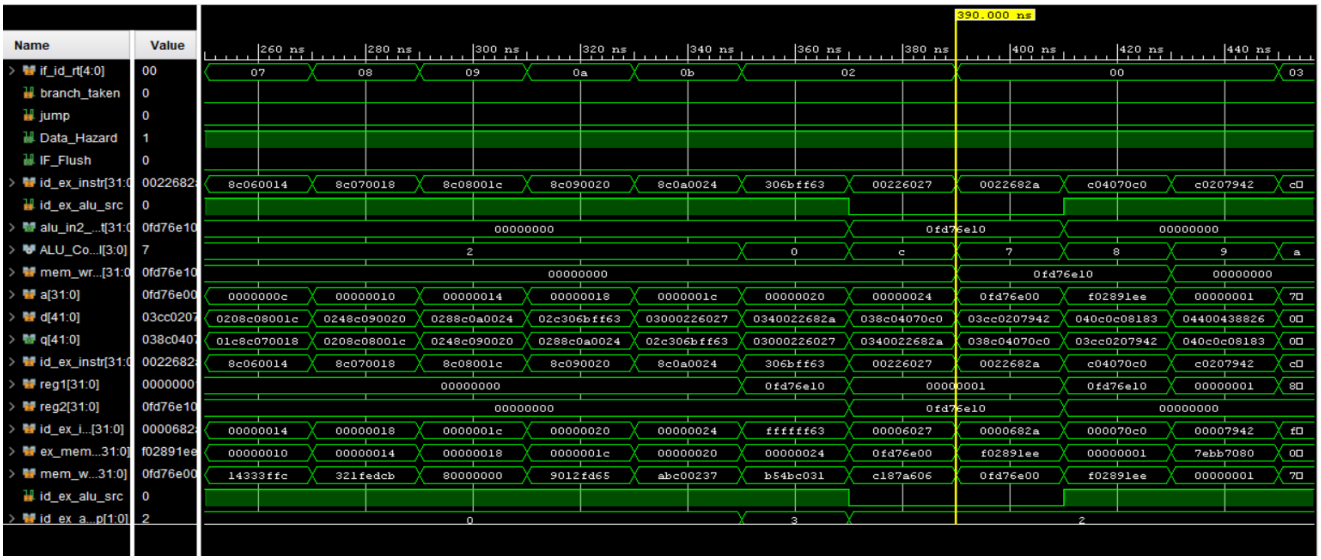
In the waveforms below, the written result matches the register content from the instruction memory.

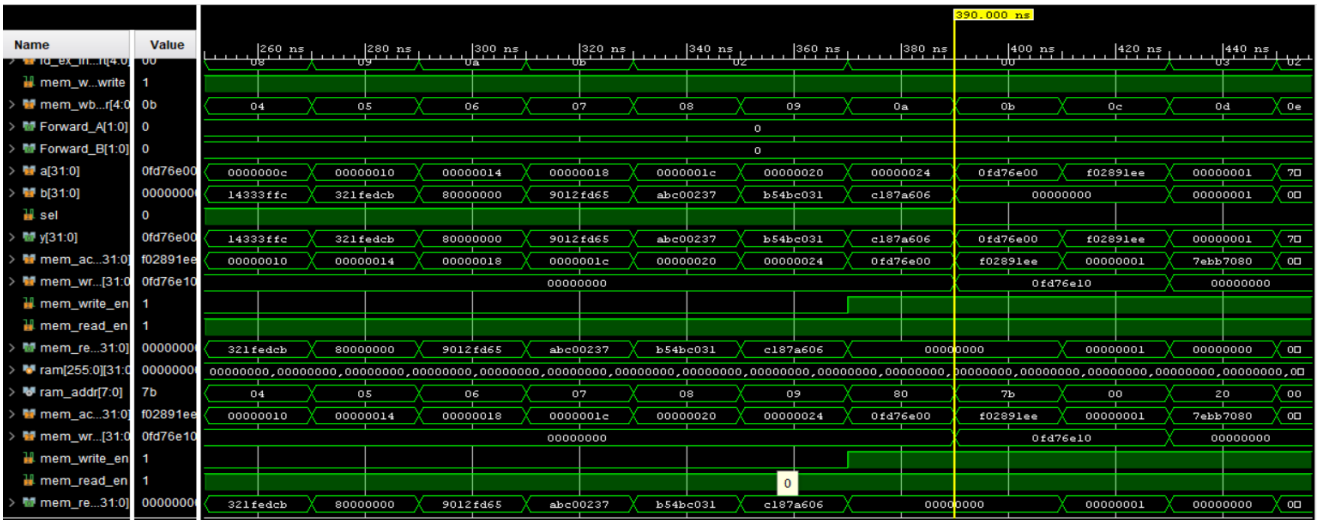




The waveforms below are for instructions 0-9 and show the inputs/outputs of other stages used to determine the resultant data.


Name
Value
alu_op[1:0]
mem_read
mem_write
alu_src
reg_write
reg_dst
hazardcheck
equal
data[31:0]
ex_mem..._writ
ex_mem...r[4:0]
id_ex_in...rs[4:0]
id_ex_in...rt[4:0]
mem_w...write
mem_wb...r[4:0]
Forward_A[1:0]
Forward_B[1:0]
en
id_ex_...m_reac
id_ex_d...eg[4:0]
if_id_rs[4:0]
390.000 ns
260 ns
280 ns
300 ns
320 ns
340 ns
360 ns
380 ns
400 ns
420 ns
440 ns


Name
Value
if_id_rt[4:0]
branch_taken
jump
Data_Hazard
IF_Flush
id_ex_instr[31:0]
id_ex_alu_src
alu_in2_...t[31:0]
ALU_Co...l[3:0]
mem_wr...[31:0]
a[31:0]
d[41:0]
q[41:0]
id_ex_instr[31:0]
reg1[31:0]
reg2[31:0]
id_ex_i...[31:0]
ex_mem...31:0]
mem_w...31:0]
id_ex_alu_src
id_ex_a...p[1:0]
390.000 ns
260 ns
280 ns
300 ns
320 ns
340 ns
360 ns
380 ns
400 ns
420 ns
440 ns

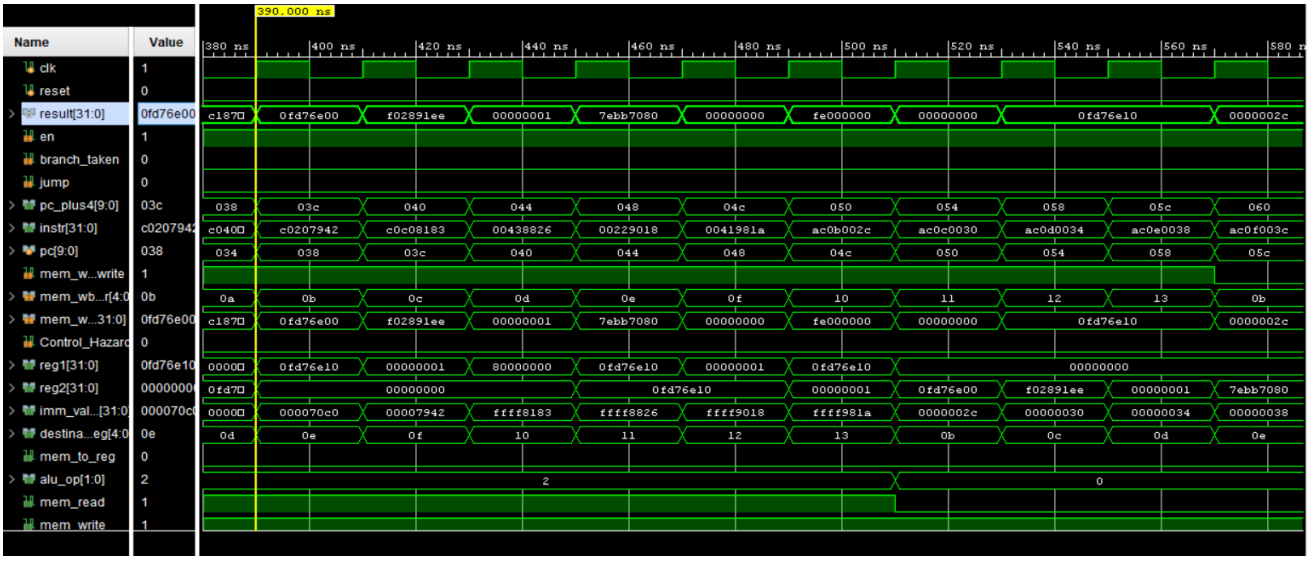

Name
Value
mem_w...write
mem_wb...r[4:0]
Forward_A[1:0]
Forward_B[1:0]
a[31:0]
b[31:0]
sel
y[31:0]
mem_ac...31:0]
mem_wr...[31:0]
mem_write_en
mem_read_en
mem_re...31:0]
ram[255:0][31:0]
ram_addr[7:0]
mem_ac...31:0]
mem_wr...[31:0]
mem_write_en
mem_read_en
mem_re...31:0]
390.000 ns
260 ns
280 ns
300 ns
320 ns
340 ns
360 ns
380 ns
400 ns
420 ns
440 ns

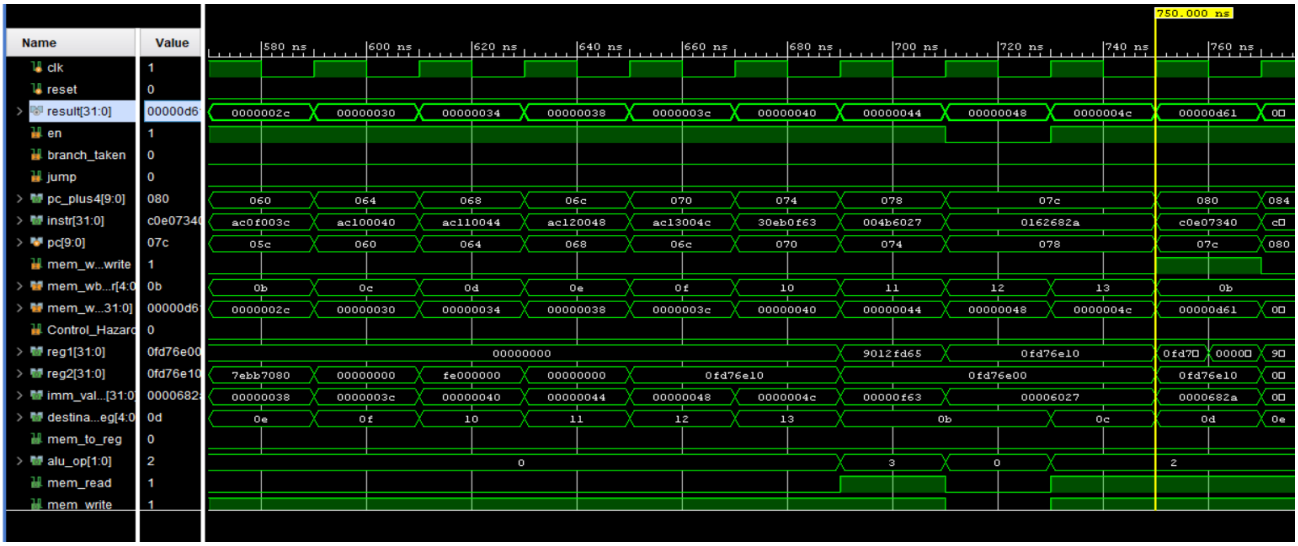
Testbench: (instructions 10-27)

```
// no dependency test, no jump
rom[10] = 32'b00110000011010111111111101100011; // andi r11,r3,#ff63        0fd76e00          r11= 0fd76e00          -
rom[11] = 32'b00000000001000100110000000100111; // nor  r12,r1,r2           f02891ee          r12= f02891ee          -
rom[12] = 32'b00000000001000100110100000101010; // slt  r13,r1,r2              1              r13= 1                 -
rom[13] = 32'b11000000010000000111000011000000; // sll  r14,r2,#3           7ebb7080          r14= 7ebb7080          -
rom[14] = 32'b11000000001000000111100101000010; // srl  r15,r1,#5                0            r15= 0                 -
rom[15] = 32'b11000000110000001000000110000011; // sra  r16,r6,#6           fe000000          r16= fe000000          -
rom[16] = 32'b00000000001000011100010000100110; // xor  r17,r2,r3           00000000          r17= 00000000          -
rom[17] = 32'b00000000001000101001000000011000; // mult r17,r1,r2           0fd76e10          r18= 0fd76e10          -
rom[18] = 32'b00000000010000011001100000011010; // div  r19,r2,r1           0fd76e10          r19= 0fd76e10          -
// store the result in memory
rom[19] = 32'b10101100000010110000000000101100; // sw mem[r0+11] <= r11        2c                -                  mem[11]= 0fd76e00
rom[20] = 32'b10101100000011000000000000110000; // sw mem[r0+12] <= r12        30                -                  mem[12]= f02891ee
rom[21] = 32'b10101100000011010000000000110100; // sw mem[r0+13] <= r13        34                -                  mem[13]=     1
rom[22] = 32'b10101100000011100000000000111000; // sw mem[r0+14] <= r14        38                -                  mem[14]= 7ebb7080
rom[23] = 32'b10101100000011110000000000111100; // sw mem[r0+15] <= r15        3c                -                  mem[15]=     0
rom[24] = 32'b10101100000100000000000001000000; // sw mem[r0+16] <= r16        40                -                  mem[16]= fe000000
rom[25] = 32'b10101100000100010000000001000100; // sw mem[r0+17] <= r17        44                -                  mem[17]= 00000000
rom[26] = 32'b10101100000100100000000001001000; // sw mem[r0+18] <= r18        48                -                  mem[18]= 0fd76e10
rom[27] = 32'b10101100000100110000000001001100; // sw mem[r0+19] <= r19        4c                -                  mem[19]= 0fd76e10
```

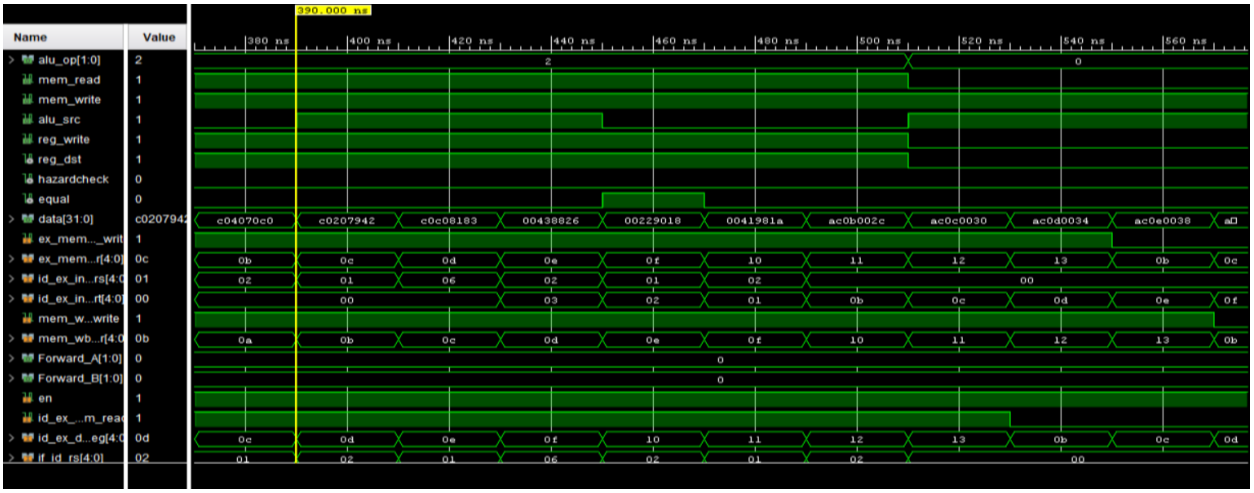
Waveform Data:

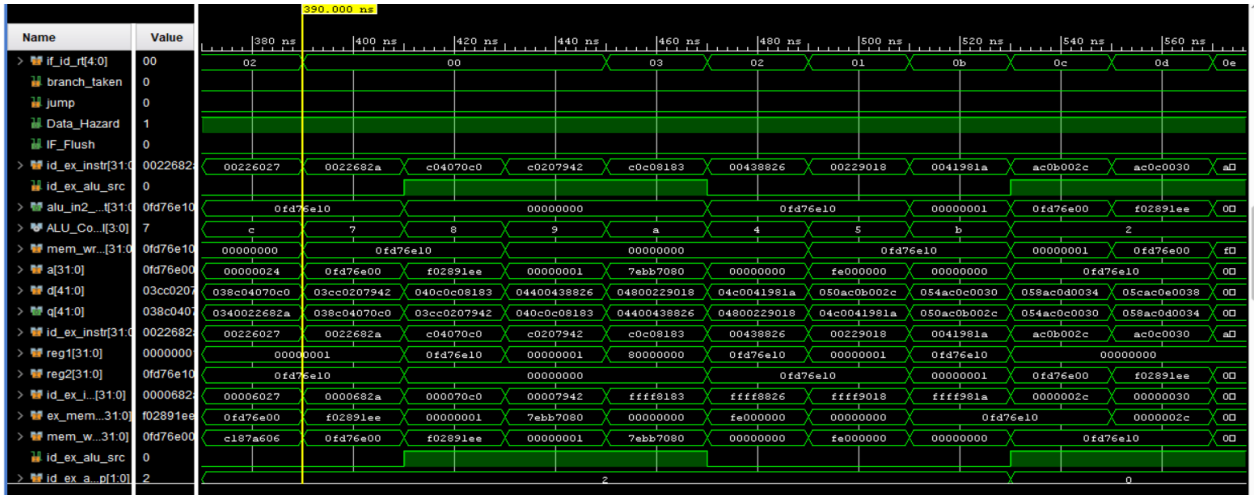
In the waveforms below, the written result matches the alu result from the instruction memory.

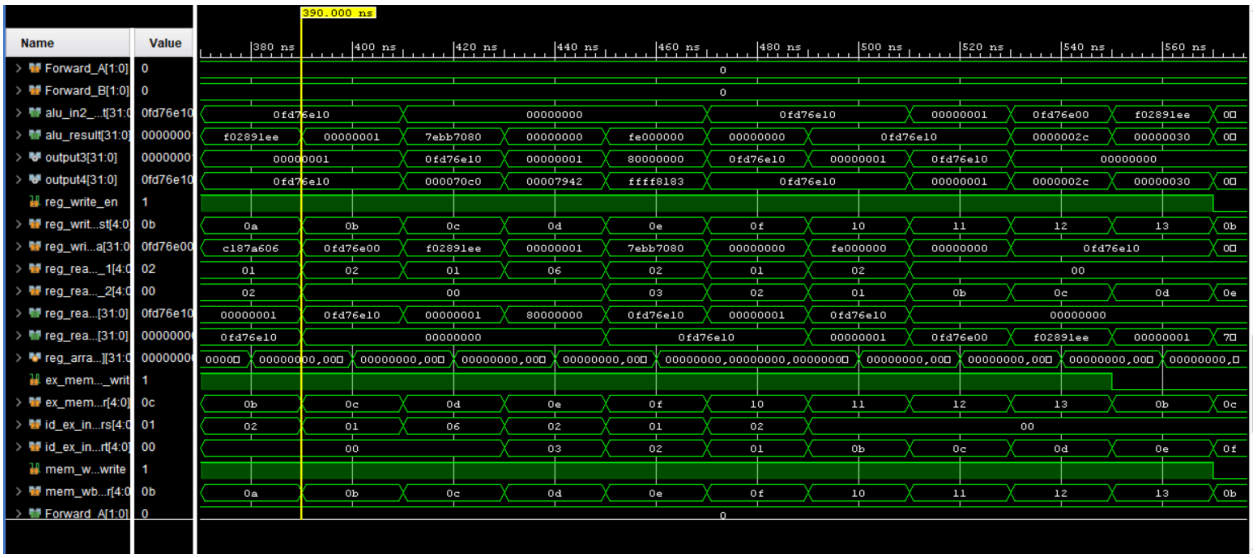


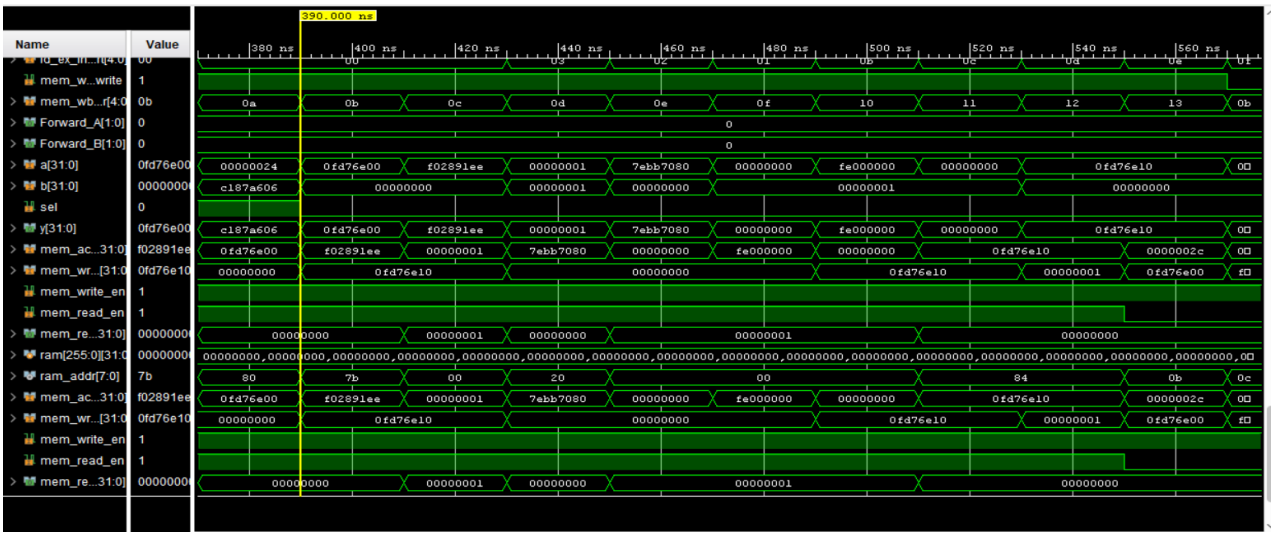


The waveforms below are for instructions 10-18 and show the inputs/outputs of other stages used to determine the resultant data. Similar data for instructions 19-27.





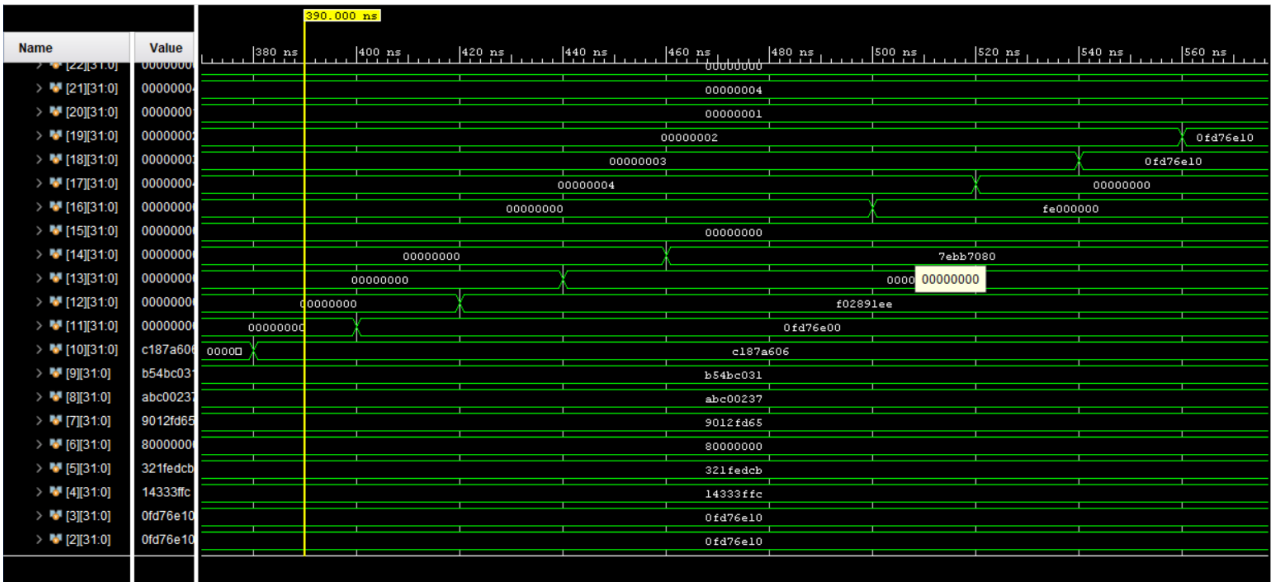




The correct result also displays for instructions beyond 27 and are not shown.

Result Contradiction/Similarity:

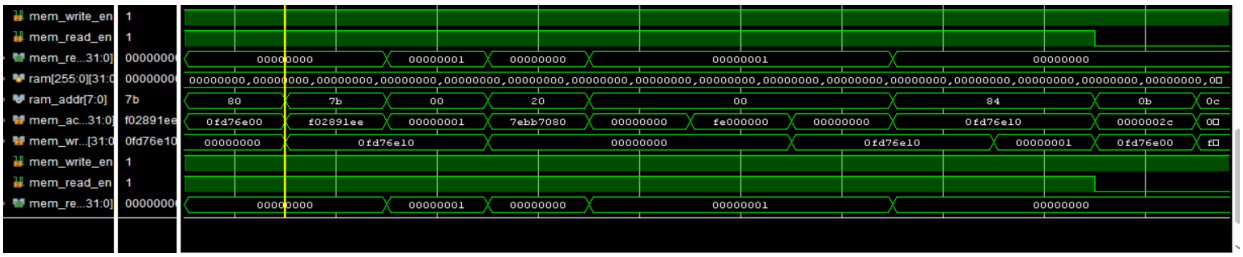
The waveform below shows the content of the reg_array.




```
if(uut.data_memory_unit.ram[11]==32'h0fd76e00) $display("NO DEPENDENCY ANDI    success!\n"); else $display("NO DEPENDENCY ANDI    failed!\n");
if(uut.data_memory_unit.ram[12]==32'hf02891ee) $display("NO DEPENDENCY NOR     success!\n"); else $display("NO DEPENDENCY NOR     failed!\n");
if(uut.data_memory_unit.ram[13]==32'h00000001) $display("NO DEPENDENCY SLT     success!\n"); else $display("NO DEPENDENCY SLT     failed!\n");
if(uut.data_memory_unit.ram[14]==32'h7ebb7080) $display("NO DEPENDENCY SLL     success!\n"); else $display("NO DEPENDENCY SLL     failed!\n");
if(uut.data_memory_unit.ram[15]==32'h00000000) $display("NO DEPENDENCY SRL     success!\n"); else $display("NO DEPENDENCY SRL     failed!\n");
if(uut.data_memory_unit.ram[16]==32'hfe000000) $display("NO DEPENDENCY SRA     success!\n"); else $display("NO DEPENDENCY SRA     failed!\n");
if(uut.data_memory_unit.ram[17]==32'h00000000) $display("NO DEPENDENCY XOR     success!\n"); else $display("NO DEPENDENCY XOR     failed!\n");
if(uut.data_memory_unit.ram[18]==32'h0fd76e10) $display("NO DEPENDENCY MULT    success!\n"); else $display("NO DEPENDENCY MULT    failed!\n");
if(uut.data_memory_unit.ram[19]==32'h0fd76e10) $display("NO DEPENDENCY DIV     success!\n"); else $display("NO DEPENDENCY DIV     failed!\n");
```

From the tb_mips32 file, the test cases are checking if ram[11] = 0fd76e00, and so on. The reg_array waveform above shows that I indeed have stored that value 0fd76e00 (also shown in result in previous waveforms) into reg[11], and so on.

I believe that the tb_mips32 prints out failed cases because the data is being stored into the reg_array instead of the ram, and so the reg_read_data comes out as 0 instead of the expected value. I believe all of the pipelined processor components are working and producing the correct result, but the result is being stored in a different place than the location that tb_mips32 is checking.



From a previous waveform, I found that when reading and writing from memory, the address to write to is 7b and is writing the value 0fd76e10 to ram[7b] instead. It then reads ram[11] as 0 because the value was not stored in ram[11] but in reg_array[11].