PROJECT REPORT - IPA

SEQUENTIAL

1. FETCH

- **Function**: Fetches the next instruction from memory and prepares it for decoding and execution.
- **Tasks**:
    - Update the Program Counter (PC) to point to the next instruction.
    - Fetch the instruction from memory using the updated PC.
    - Store the fetched instruction in an Instruction Register.
    - Increment the PC for the next instruction fetch.
- **Key Components**:
    - Program Counter (PC): Tracks the address of the next instruction.
    - Instruction Memory: Stores program instructions.
    - Instruction Register: Holds the fetched instruction for processing.

Operation:

- On the clock signal's positive edge, update the PC and access instruction memory.
- Retrieve the instruction if the PC is valid; otherwise, perform a nop operation.
- Store the fetched instruction in the Instruction Register.
- Increment the PC for the next instruction.
- Ensures continuous instruction fetching and processing for subsequent stages.

The Fetch Stage efficiently manages instruction retrieval, setting the groundwork for decoding, execution, memory access, and writeback stages in the pipeline. This sequential approach enhances processor performance by overlapping instruction execution.

```
module fetch
(
    input clock,
    input [63:0] pc_counter,
    output reg [3:0] icode, ifun, ra, rb,
    output reg [63:0] valc, valp,
    output reg instruction_valid, imem_error, halt
);

reg [7:0] instruction_memory[0:1023];
reg [0:79] current_instruction ;

initial begin
    $readmemb("3.txt", instruction_memory);
    //modif
    icode=0;
    ifun=0;
    halt=0;
    imem_error=0;
    instruction_valid=1;
    ra=0;
    rb=0;
    valc=0;
    valp=0;
end


always@(posedge clock)
begin

    current_instruction[0 :7]=instruction_memory[pc_counter];
    current_instruction[8:15]=instruction_memory[pc_counter+1];
    current_instruction[16:23]=instruction_memory[pc_counter+2];
    current_instruction[24:31]=instruction_memory[pc_counter+3];
    current_instruction[32:39]=instruction_memory[pc_counter+4];
    current_instruction[40:47]=instruction_memory[pc_counter+5];
    current_instruction[48:55]=instruction_memory[pc_counter+6];
    current_instruction[56:63]=instruction_memory[pc_counter+7];
    current_instruction[64:71]=instruction_memory[pc_counter+8];
    current_instruction[72:79]=instruction_memory[pc_counter+9];

    icode=current_instruction[0:3];
    ifun=current_instruction[4:7];
    //instruction_valid=1'b1;
    //modif
    // halt=1'b0;

```

```
48      if(pc_counter>1023)
49      begin
50          imem_error=1'b1;
51      end
52      else
53      begin
54          imem_error=1'b0;
55      end
56
57      if(icode==4'd0 && ifun==4'd0) //halt
58      begin
59          halt=1'b1;
60          valp=pc_counter+64'd1;
61      end
62
63      else if(icode==4'd1 && ifun==4'd0) //nop
64      begin
65          valp=pc_counter+64'd1;
66      end
67
68      else if(icode==4'd2 && (ifun==4'd0 || ifun==4'd1 || ifun==4'd2 ||
69              ifun==4'd6)) //cmovxx
70      begin
71          ra=current_instruction[8:11];
72          rb=current_instruction[12:15];
73          valp=pc_counter+64'd2;
74      end
75
76      else if(icode==4'd3 && ifun==4'd0) //irmovq
77      begin
78          ra=current_instruction[8:11];
79          rb=current_instruction[12:15];
80          valc={
81              current_instruction[72:79],
82              current_instruction[64:71],
83              current_instruction[56:63],
84              current_instruction[48:55],
85              current_instruction[40:47],
86              current_instruction[32:39],
87              current_instruction[24:31],
88              current_instruction[16:23]
89          };
90          valp=pc_counter+64'd10;
91      end
92
93      else if(icode==4'd4 && ifun==4'd0) //rmmovq
94      begin
```

```
95          ra=current_instruction[8:11];
96          rb=current_instruction[12:15];
97          valc={
98              current_instruction[72:79],
99              current_instruction[64:71],
100             current_instruction[56:63],
101             current_instruction[48:55],
102             current_instruction[40:47],
103             current_instruction[32:39],
104             current_instruction[24:31],
105             current_instruction[16:23]
106         };
107         valp=pc_counter+64'd10;
108     end
109
110     else if(icode==4'd5 && ifun==4'd0) //mrmovq
111     begin
112         ra=current_instruction[8:11];
113         rb=current_instruction[12:15];
114         valc={
115             current_instruction[72:79],
116             current_instruction[64:71],
117             current_instruction[56:63],
118             current_instruction[48:55],
119             current_instruction[40:47],
120             current_instruction[32:39],
121             current_instruction[24:31],
122             current_instruction[16:23]
123         };
124         valp=pc_counter+64'd10;
125     end
126
127     else if(icode==4'd6 && (ifun==4'd0 || ifun==4'd1 || ifun==4'd2 ||
128     begin
129         ra=current_instruction[8:11];
130         rb=current_instruction[12:15];
131         valp=pc_counter+64'd2;
132     end
133
134     else if(icode==4'd7 && (ifun==4'd0 || ifun==4'd1 || ifun==4'd2 ||
135             ifun==4'd6 )) // jump
136     begin
137                 valc={
138             current_instruction[64:71],
139             current_instruction[56:63],
140             current_instruction[48:55],
141             current_instruction[40:47],
```

```
142             current_instruction[32:39],
143             current_instruction[24:31],
144             current_instruction[16:23],
145             current_instruction[8:15]
146         };
147         valp=pc_counter+64'd9;
148     end
149 
150     else if(icode==4'd8 && ifun==4'd0) // call
151     begin
152                 valc={
153             current_instruction[64:71],
154             current_instruction[56:63],
155             current_instruction[48:55],
156             current_instruction[40:47],
157             current_instruction[32:39],
158             current_instruction[24:31],
159             current_instruction[16:23],
160             current_instruction[8:15]
161         };
162         valp=pc_counter+64'd9;
163     end
164 
165     else if(icode==4'd9 && ifun==4'd0) //return
166     begin
167          valp=pc_counter+64'd1;
168     end
169 
170     else if(icode==4'd10 && ifun==4'd0) // pushq
171     begin
172         ra=current_instruction[8:11];
173         rb=current_instruction[12:15];
174         valp=pc_counter+64'd2;
175     end
176 
177     else if(icode==4'd11 && ifun==4'd0) //popq
178     begin
179         ra=current_instruction[8:11];
180         rb=current_instruction[12:15];
181         valp=pc_counter+64'd2;
182     end
183 
184     else
185     begin
186         instruction_valid=1'b0;
187         //modif
188         $finish;
189     end
```

2. DECODE

Function: Decodes the fetched instruction to determine the operation to be performed.

Tasks:

- Interpret the opcode and operands of the instruction.
- Identify the registers involved in the instruction.
- Access register values if needed for the instruction.
- Prepare control signals for the execution stage based on the decoded instruction.

Key Components:

- Instruction Opcode: Specifies the operation to be executed.
- Operands: Data or addresses on which the operation acts.
- Register File: Stores register values for instruction execution.
- Control Unit: Generates control signals for the execution stage.

Operation:

- Receive the fetched instruction and extract the opcode and operands.
- Determine the registers involved in the instruction for data access.
- Access register values from the Register File if required for the instruction.
- Generate control signals based on the decoded instruction for the execution stage.
- Prepares the instruction for execution by providing necessary data and control signals.

The Decode Stage plays a crucial role in understanding the fetched instruction and setting the stage for its execution. By decoding instructions efficiently, the processor can accurately execute operations and advance the instruction through the pipeline stages for processing.

```
module decode (
    input clock,
    input [3:0] icode, ra, rb,
    input [63:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14,
    output reg [63:0] vala, valb
);

    reg [63:0] register_file [0:14];

    always @(*) begin

        if(clock) begin

        register_file[0] = r0;
        register_file[1] = r1;
        register_file[2] = r2;
        register_file[3] = r3;
        register_file[4] = r4;
        register_file[5] = r5;
        register_file[6] = r6;
        register_file[7] = r7;
        register_file[8] = r8;
        register_file[9] = r9;
        register_file[10] = r10;
        register_file[11] = r11;
        register_file[12] = r12;
        register_file[13] = r13;
        register_file[14] = r14;

        case (icode)

            4'd2: // cmovxx
                begin
                    vala = register_file[ra];
                end

            4'd4: // rmmovq
                begin
                    vala = register_file[ra];
                    valb = register_file[rb];
                end

            4'd5: // mrmovq
                begin
                    valb = register_file[rb];
                end

            4'd6: // opq
                begin
```

```
            4'd6: // opq
                begin
                    vala = register_file[ra];
                    valb = register_file[rb];
                end

            4'd8: // call
                begin
                    valb = register_file[4]; // rsp
                end

            4'd9: // return
                begin
                    vala = register_file[4]; // rsp
                    valb = register_file[4]; // rsp
                end

            4'd10: // pushq
                begin
                    vala = register_file[ra];
                    valb = register_file[4]; // rsp
                end

            4'd11: // popq
                begin
                    vala = register_file[4]; // rsp
                    valb = register_file[4]; // rsp
                end

        endcase
        end
    end
endmodule
```

3. EXECUTE

- **Function**: Executes the decoded instruction by performing the specified operation on the provided data.
- **Tasks**:
    - Perform arithmetic, logic, or data manipulation operations based on the instruction.
    - Calculate the result of the operation using the provided data.
    - Handle branch instructions by evaluating conditions and updating the Program Counter (PC) if necessary.
    - Set flags or status indicators based on the result of the operation.
- **Key Components**:
    - Arithmetic Logic Unit (ALU): Executes arithmetic and logic operations.
    - Data Inputs: Values or operands required for the operation.
    - Branch Condition Logic: Evaluates conditions for branching instructions.
    - Flags Register: Stores status flags like zero, sign, overflow, etc.

Operation:

- Receive the decoded instruction and necessary data inputs.
- Execute the specified operation using the ALU or dedicated hardware.
- Calculate the result of the operation based on the provided data.
- Evaluate branch conditions for conditional jumps and update the PC accordingly.
- Set flags or status indicators based on the result of the operation for further processing.

The Execute Stage is responsible for the actual computation and manipulation of data as instructed by the decoded instruction. By performing operations accurately and efficiently, the processor progresses towards completing the instruction execution cycle within the pipeline architecture.

```v
`include "./ALU/ALU.v"
`include "./ALU/decoder.v"
`include "./ALU/fulladder.v"
`include "./ALU/subt.v"
`include "./ALU/XOR.v"
`include "./ALU/adder.v"
`include "./ALU/AND.v"

module execute(
    input clock,
    input signed [63:0] vala, valb, valc,
    input [3:0] icode, ifun,
    output reg [63:0] vale,
    output reg condition_cnd,
    output reg overflow_flag,
    output reg sign_flag,
    output reg zero_flag
);

// i/o for alu
reg signed [63:0] input_A, input_B;
reg c0, c1;
wire overflow;
wire signed [63:0] output_alu;

ALU alu0(
    .c0(c0),
    .c1(c1),
    .a(input_A),
    .b(input_B),
    .output_alu(output_alu),
    .bit_overflow(overflow)
    );

initial begin
    overflow_flag = 1'b0;
    sign_flag = 1'b0;
    zero_flag = 1'b0;
    condition_cnd = 1'b0;
end

always @(*) begin

    if(clock==1)
    begin
        //test
        condition_cnd=0;
```

```
48
49          case (icode)
50
51              4'd2:
52                  begin
53                      {c1, c0} = 2'b00;
54                      input_A = vala;
55                      input_B = 64'd0;
56                      vale=output_alu;
57                      if(ifun==4'd0)
58                      begin
59                          condition_cnd=1'b1;
60                      end
61                      else if(ifun==4'd1)
62                      begin
63                          condition_cnd=((sign_flag^overflow_flag) | zero_flag);
64                      end
65                      else if(ifun==4'd2)
66                      begin
67                          condition_cnd=(sign_flag^overflow_flag);
68                      end
69                      else if(ifun==4'd3)
70                      begin
71                          condition_cnd=zero_flag;
72                      end
73                      else if(ifun==4'd4)
74                      begin
75                          if(zero_flag==0)
76                          begin
77                              condition_cnd=1'b1;
78                          end
79                      end
80                      else if(ifun==4'd5)
81                      begin
82                          if((sign_flag^overflow_flag)==0)
83                              condition_cnd=1'b1;
84                      end
85                      else if(ifun==4'd6)
86                      begin
87                          if(((sign_flag^overflow_flag) | zero_flag)==0)
88                              condition_cnd=1'b1;
89                      end
90                  end
91
92              4'd3:
93                  begin
94                      {c1, c0} = 2'b00;
```

```
94                      {c1, c0} = 2'b00;
95                      input_A = valc;
96                      input_B = 64'd0;
97                      vale=output_alu;
98                  end
99
100             4'd4:
101                 begin
102                     {c1, c0} = 2'b00;
103                     input_A = valc;
104                     input_B = valb;
105                     vale=output_alu;
106                 end
107
108             4'd5:
109                 begin
110                     {c1, c0} = 2'b00;
111                     input_A = valc;
112                     input_B = valb;
113                     vale=output_alu;
114                 end
115
116             4'd6:
117                 begin
118                     {c1, c0} = ifun[1:0];
119                     input_A = valb;
120                     input_B = vala;
121                     vale=output_alu;
122
123                     zero_flag = (output_alu==1'b0);
124                     sign_flag = (output_alu[63]);
125                     overflow_flag = overflow;
126                 end
127
128             4'd7:
129                 begin
130                     if(ifun==4'd0)
131                         condition_cnd=1'b1;
132                     else if(ifun==4'd1)
133                         condition_cnd=((sign_flag^overflow_flag) | zero_flag);
134                     else if(ifun==4'd2)
135                         condition_cnd=(sign_flag^overflow_flag);
136                     else if(ifun==4'd3)
137                         condition_cnd=zero_flag;
138                     else if(ifun==4'd4)
139                     begin
140                         if(zero_flag==0)
```

```
141                             condition_cnd=1'b1;
142                         end
143                         else if(ifun==4'd5)
144                         begin
145                             if((sign_flag^overflow_flag)==0)
146                                 condition_cnd=1'b1;
147                         end
148                         else if(ifun==4'd6)
149                         begin
150                             if(((sign_flag^overflow_flag) | zero_flag)==0)
151                                 condition_cnd=1'b1;
152                         end
153                     end
154 
155                 4'd8:
156                     begin
157                         {c1, c0} = 2'b01;
158                         input_A = valb;
159                         input_B = 64'd8;
160                         vale=output_alu;
161                     end
162 
163                 4'd9:
164                     begin
165                         {c1, c0} = 2'b00;
166                         input_A = 64'd8;
167                         input_B = valb;
168                         vale=output_alu;
169                     end
170 
171                 4'd10:
172                     begin
173                         {c1, c0} = 2'b01;
174                         input_A = valb;
175                         input_B = 64'd8;
176                         vale=output_alu;
177                     end
178 
179                 4'd11:
180                     begin
181                         {c1, c0} = 2'b00;
182                         input_A = 64'd8;
183                         input_B = valb;
184                         vale=output_alu;
185                     end
186             endcase
187         end
188 end
```

4.MEMORY

- **Function**: Handles data read and write operations to and from memory as required by the instruction.
- **Tasks**:
    - Read data from memory for load operations or write data to memory for store operations.
    - Calculate memory addresses based on the instruction and provided data.
    - Update memory contents based on the operation being performed.
    - Prepare data for the next stage or writeback to registers.
- **Key Components**:
    - Data Memory: Stores data values for read and write operations.
    - Memory Address Calculation: Determines the memory location to access.
    - Data Input/Output: Data values to be written to or read from memory.
    - Memory Data Register: Holds the data read from or written to memory.

Operation:

- Receive the necessary data inputs and instruction type to determine read or write operation.
- Calculate the memory address based on the instruction and provided data values.
- Read data from memory for load operations or write data to memory for store operations.
- Update memory contents based on the operation being performed.
- Prepare the data read from memory for the next stage or writeback to registers as required.

The Memory Stage manages data transfer between the processor and memory, ensuring accurate read and write operations based on the instruction's memory access requirements. By efficiently handling memory operations, the processor progresses towards completing the instruction execution cycle within the pipeline architecture.

```
module memory(
    input clock,
    input [3:0] M_icode,M_dste,M_dstm,
    input [63:0] M_vala,M_vale,
    input [1:0] M_status,
    output reg [1:0] m_status,
    output reg [3:0] m_icode,m_dste,m_dstm,
    output reg memory_block_error,
    output reg [63:0] m_vale,m_valm,written_mem
);


reg [63:0] memory_register [0:1023];

initial begin
    memory_block_error=0;
end

always@(*)
begin
    if(M_icode==4'd5) //mrmovq
    begin
        if(M_vale>1023)
            memory_block_error=1;
        else
            m_valm=memory_register[M_vale];
    end

    if(M_icode==4'd9) //return
    begin
        if(M_vala>1023)
            memory_block_error=1;
        else
            m_valm=memory_register[M_vala];
    end

    if(M_icode==4'd11) //popq
    begin
        if(M_vala>1023)
            memory_block_error=1;
        else
            m_valm=memory_register[M_vala];
    end

    m_vale=M_vale;
    m_icode=M_icode;
    m_dste=M_dste;
    m_dstm=M_dstm;
```

```
end

// always@(*)
// begin
// end

// always@(negedge clock)
always@(*)
begin
    if(M_vale>1023)
        memory_block_error=1;
    else
    begin
        if(M_icode==4'd4) //rmmovq
        begin
            memory_register[M_vale]=M_vala;
            written_mem=memory_register[M_vale];
        end

        if(M_icode==4'd8) //call
        begin
            memory_register[M_vale]=M_vala;
            written_mem=memory_register[M_vale];
        end

        if(M_icode==4'd10) //pushq
        begin
            memory_register[M_vale]=M_vala;
            written_mem=memory_register[M_vale];
        end
    end
end

always@(*)
begin
    if(memory_block_error==1)
        m_status=3;
    else
        m_status=M_status;
end

endmodule
```

5.WRITEBACK

Function: Writes the results of the executed instruction back to the appropriate destination registers or memory locations.

- **Tasks**:
  - Determine the destination for the computed result based on the instruction type.
  - Update register values or memory locations with the computed data.
  - Handle register file writes and stack pointer updates.
  - Prepare data for further processing or completion of the instruction.
- **Key Components**:
  - Register File: Stores register values for the processor.
  - Stack Pointer (rsp): Manages stack operations and memory addresses.
  - Data Registers: Hold the computed results of the executed instruction.
  - Control Signals: Direct the data flow to the appropriate destinations.

Operation:

- Receive the computed results (valE or valM) from the previous stage.
- Determine the destination register or memory location based on the instruction type.
- Update the register file with the computed data for instructions like mov, add, sub, etc.
- Manage stack pointer updates for push, pop, call, and return instructions.
- Prepare the data for further processing or completion of the instruction cycle.

The Write Back Stage finalises the instruction execution cycle by updating register values or memory locations with the computed results. By efficiently handling data writes and updates, the processor ensures the correct flow of information and progress towards completing the instruction's operation within the pipeline architecture.

```
module writeback(
    input clock,
    input [63:0]vale,valm,
    input condition_cnd,
    input[3:0] ra,rb,icode,
    output reg [63:0] register0,register1,register2,register3,register4,register5,register6,register7,register8,register9,register10,register11,register12,register13,register14
);

reg[63:0] register_file[0:14];

always@(negedge clock)
begin

    if(icode==4'd2)//cmovxx
        begin
            if(condition_cnd==1)
            begin
                register_file[rb]=vale;
            end
        end

    else if(icode==4'd3) //irmovq
        begin
            register_file[rb]=vale;
        end

    else if(icode==4'd5) //mrmovq
        begin
            register_file[ra]=valm;
        end

    else if(icode==4'd6) //opq
        begin
            register_file[rb]=vale;
        end

    else if(icode==4'd8) //call
        begin
            register_file[4]=vale;
        end

    else if(icode==4'd9) //return
        begin
            register_file[4]=vale;
        end

    else if(icode==4'd10) //pushq
        begin
            register_file[4]=vale;
        end

    else if(icode==4'd11) //popq
        begin
            register_file[4]=vale;
            register_file[ra]=valm;
        end

    register0=register_file[0];
    register1=register_file[1];
    register2=register_file[2];
    register3=register_file[3];
    register4=register_file[4];
    register5=register_file[5];
    register6=register_file[6];
    register7=register_file[7];
    register8=register_file[8];
    register9=register_file[9];
    register10=register_file[10];
    register11=register_file[11];
    register12=register_file[12];
    register13=register_file[13];
    register14=register_file[14];
end

endmodule
```

6.PC UPDATE

- **Function**: Determines the next value of the Program Counter (PC) after the execution of an instruction.
- **Tasks**:
    - Calculate the updated PC value based on the instruction type and execution conditions.
    - Handle branch instructions by updating the PC with the target address or the next sequential address.
    - Manage control flow changes due to conditional jumps, calls, and returns.
    - Prepare the PC value for the next instruction fetch stage.
- **Key Components**:
    - Program Counter (PC): Stores the address of the next instruction to be fetched.
    - Branch Condition Logic: Evaluates conditions for branching instructions.
    - Control Signals: Direct the flow of instructions based on execution outcomes.
    - Target Address: Specifies the address to jump to for branch instructions.

Operation:

- Receive the necessary inputs such as the current PC value, instruction type, and execution conditions.
- Calculate the updated PC value based on the instruction type and conditions.
- Handle branch instructions by updating the PC with the target address or the next sequential address.
- Manage control flow changes for conditional jumps, calls, and returns by setting the PC accordingly.
- Prepare the updated PC value for the next instruction fetch stage to continue the instruction cycle.

The PC Update Stage plays a critical role in managing the flow of instructions within the processor by determining the next instruction address to be fetched. By accurately updating the PC based on instruction outcomes and conditions, the processor ensures the correct execution sequence and progression through the instruction pipeline.

```
module pc_update(
    input clock,condition_cnd,
    input[63:0] valc,valp,valm,
    input [3:0] icode,
    output reg [63:0] new_pc
);

always@(*)
begin

    if(clock) begin

        if(icode==4'd1 | icode==4'd2 | icode==4'd3
                | icode==4'd4 | icode==4'd5 | icode==4'd6
                | icode==4'd10 | icode==4'd11) // nop,cmovxx,irmovq,rmmovq,mrmovq,opq,pushq,popq
        begin
            new_pc=valp;
        end

        else if(icode==4'd7) //jxx
        begin
            if(condition_cnd==1'b1)
            begin
                new_pc=valc;
            end

            else
            begin
                new_pc=valp;
            end
        end

        else if(icode==4'd8) //call
        begin
            new_pc=valc;
        end

        else if(icode==4'd9) //return
        begin
            new_pc=valm;
        end
    end
end

endmodule
```

WRAPPER MODULE INPUTS AND OUTPUTS

1. ADDING 256+512

```
1    // Reference for 1.txt
2    irmovq $0x100, %rbx
3    irmovq $0x200, %rdx
4    addq %rdx, %rbx
5
```

```
VCD info: dumpfile processor.vcd opened for output.
clock=0, pc_counter=                0, icode= 0, ifun= 0, rA= 0, rB= 0, valA=               x, valB=               x, valC=               0,
clock=1, pc_counter=               10, icode= 3, ifun= 0, rA=15, rB= 3, valA=               x, valB=               x, valC=             256,
clock=0, pc_counter=               10, icode= 3, ifun= 0, rA=15, rB= 3, valA=               x, valB=               x, valC=             256,
clock=1, pc_counter=               20, icode= 3, ifun= 0, rA=15, rB= 2, valA=               x, valB=               x, valC=             512,
clock=0, pc_counter=               20, icode= 3, ifun= 0, rA=15, rB= 2, valA=               x, valB=               x, valC=             512,
clock=1, pc_counter=               22, icode= 6, ifun= 0, rA= 2, rB= 3, valA=             512, valB=             256, valC=             512,
clock=0, pc_counter=               22, icode= 6, ifun= 0, rA= 2, rB= 3, valA=             512, valB=             256, valC=             512,
./fetch.v:188: $finish called at 70 (1s)
processor.v:143: $finish called at 70 (1s)
clock=1, pc_counter=               22, icode= x, ifun= x, rA= 2, rB= 3, valA=             512, valB=             256, valC=             512,
```

```
valP=                0, valE=                x, valM=                x, cnd=0, status=0, r2=                x, r3=                x
valP=               10, valE=              256, valM=                x, cnd=0, status=0, r2=                x, r3=                x
valP=               10, valE=              256, valM=                x, cnd=0, status=0, r2=                x, r3=              256
valP=               20, valE=              512, valM=                x, cnd=0, status=0, r2=                x, r3=              256
valP=               20, valE=              512, valM=                x, cnd=0, status=0, r2=              512, r3=              256
valP=               22, valE=              768, valM=                x, cnd=0, status=0, r2=              512, r3=              256
valP=               22, valE=              768, valM=                x, cnd=0, status=0, r2=              512, r3=              768


valP=               22, valE=              768, valM=                x, cnd=0, status=2, r2=              512, r3=              768
```

2. ADDING FIRST 10 NUMBERS

```
irmovq $0, %rax
irmovq $10, %rdx
irmovq $1, %rsi

loop:
    addq %rdx, %rax
    subq %rsi, %rdx
    andq %rdx, %rdx
    jne loop

halt
```

```
VCD info: dumpfile processor.vcd opened for output.
clock=0, pc_counter=                  0, icode= 0, ifun= 0, r0=                  x

clock=1, pc_counter=                 10, icode= 3, ifun= 0, r0=                  x

clock=0, pc_counter=                 10, icode= 3, ifun= 0, r0=                  0

clock=1, pc_counter=                 20, icode= 3, ifun= 0, r0=                  0

clock=0, pc_counter=                 20, icode= 3, ifun= 0, r0=                  0

clock=1, pc_counter=                 30, icode= 3, ifun= 0, r0=                  0

clock=0, pc_counter=                 30, icode= 3, ifun= 0, r0=                  0

clock=1, pc_counter=                 32, icode= 6, ifun= 0, r0=                  0

clock=0, pc_counter=                 32, icode= 6, ifun= 0, r0=                 10

clock=1, pc_counter=                 34, icode= 6, ifun= 1, r0=                 10

clock=0, pc_counter=                 34, icode= 6, ifun= 1, r0=                 10

clock=1, pc_counter=                 36, icode= 6, ifun= 2, r0=                 10

clock=0, pc_counter=                 36, icode= 6, ifun= 2, r0=                 10

clock=1, pc_counter=                 30, icode= 7, ifun= 4, r0=                 10

clock=0, pc_counter=                 30, icode= 7, ifun= 4, r0=                 10

clock=1, pc_counter=                 32, icode= 6, ifun= 0, r0=                 10

clock=0, pc_counter=                 32, icode= 6, ifun= 0, r0=                 19

clock=1, pc_counter=                 34, icode= 6, ifun= 1, r0=                 19

clock=0, pc_counter=                 34, icode= 6, ifun= 1, r0=                 19

clock=1, pc_counter=                 36, icode= 6, ifun= 2, r0=                 19

clock=0, pc_counter=                 36, icode= 6, ifun= 2, r0=                 19

clock=1, pc_counter=                 30, icode= 7, ifun= 4, r0=                 19
```

```
clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                19

clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                19

clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                27

clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                27

clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                27

clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                27

clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                27

clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                27

clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                27

clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                27

clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                34

clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                34

clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                34

clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                34

clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                34

clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                34

clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                34

clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                34

clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                40

clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                40

clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                40

clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                40

clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                40
```

```
 92
 93   clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                40
 94
 95   clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                40
 96
 97   clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                40
 98
 99   clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                45
100
101   clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                45
102
103   clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                45
104
105   clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                45
106
107   clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                45
108
109   clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                45
110
111   clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                45
112
113   clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                45
114
115   clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                49
116
117   clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                49
118
119   clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                49
120
121   clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                49
122
123   clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                49
124
125   clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                49
126
127   clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                49
128
129   clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                49
130
131   clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                52
132
133   clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                52
134
135   clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                52
```

```
136
137   clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                52
138
139   clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                52
140
141   clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                52
142
143   clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                52
144
145   clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                52
146
147   clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                54
148
149   clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                54
150
151   clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                54
152
153   clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                54
154
155   clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                54
156
157   clock=1, pc_counter=                30, icode= 7, ifun= 4, r0=                54
158
159   clock=0, pc_counter=                30, icode= 7, ifun= 4, r0=                54
160
161   clock=1, pc_counter=                32, icode= 6, ifun= 0, r0=                54
162
163   clock=0, pc_counter=                32, icode= 6, ifun= 0, r0=                55
164
165   clock=1, pc_counter=                34, icode= 6, ifun= 1, r0=                55
166
167   clock=0, pc_counter=                34, icode= 6, ifun= 1, r0=                55
168
169   clock=1, pc_counter=                36, icode= 6, ifun= 2, r0=                55
170
171   clock=0, pc_counter=                36, icode= 6, ifun= 2, r0=                55
172
173   clock=1, pc_counter=                45, icode= 7, ifun= 4, r0=                55
174
175   clock=0, pc_counter=                45, icode= 7, ifun= 4, r0=                55
176
177   processor.v:143: $finish called at 870 (1s)
178   clock=1, pc_counter=                45, icode= 0, ifun= 0, r0=                55
```

PIPELINING

1. FETCH

- **Function**: Fetches instructions from memory and prepares them for decoding.
- **Tasks**:
    - Retrieve instructions from memory based on the current Program Counter (PC).
    - Prepare fetched instructions for decoding by extracting opcode and operands.
    - Handle branch prediction and instruction alignment for the next cycle.
    - Maintain a continuous flow of instructions into the pipeline.
- **Key Components**:
    - Instruction Memory: Stores program instructions for fetching.
    - Program Counter (PC): Tracks the address of the next instruction to fetch.
    - Fetch Register: Holds the fetched instruction for decoding.
    - Branch Prediction Logic: Predicts the target address for branch instructions.

Operation:

- Fetch the next instruction from memory based on the current PC value.
- Extract opcode and operands from the fetched instruction for decoding.
- Predict branch targets and align instructions for the next cycle.
- Maintain a steady flow of instructions into the pipeline for continuous processing.

```
module decode(
    input clock,
    input [3:0] D_icode, D_ifun, D_ra, D_rb,
    input [63:0] D_valp,D_valc,
    input [1:0] D_status,
    input [63:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14,
    input [63:0] e_vale, m_valm, M_vale, W_valm, W_vale,
    input [3:0] e_dste, M_dstm, M_dste, W_dstm, W_dste,
    output reg [63:0] d_vala, d_valb,d_valc,
    output reg [1:0] d_status,
    output reg [3:0] d_dste, d_dstm, d_srca, d_srcb, d_icode, d_ifun
);

reg [63:0] register_file [0:14];
reg [63:0] vala,valb;

always @(*)
begin

        d_icode=D_icode;
        d_ifun=D_ifun;
        d_status=D_status;
        d_valc=D_valc;
    // if(clock)
    // begin
        register_file[0] = r0;
        register_file[1] = r1;
        register_file[2] = r2;
        register_file[3] = r3;
        register_file[4] = r4;
        register_file[5] = r5;
        register_file[6] = r6;
        register_file[7] = r7;
        register_file[8] = r8;
        register_file[9] = r9;
        register_file[10] = r10;
```

```
38              register_file[12] = r12;
39              register_file[13] = r13;
40              register_file[14] = r14;
41
42                  if(d_icode==4'd0 | d_icode==4'd1 | d_icode==4'd7) //halt,nop,jump
43                      begin
44                          d_srca=4'd15;d_srcb=4'd15;d_dste=4'd15;d_dstm=4'd15;
45                      end
46
47                  else if(d_icode==4'd2 | d_icode==4'd6) // cmovxx
48                      begin
49                          d_srca=D_ra;d_srcb=D_rb;d_dste=D_rb;d_dstm=4'd15;
50                      end
51
52                  else if(d_icode==4'd3)  //irmovq
53                      begin
54                          d_srca=4'd15;d_srcb=D_rb;d_dste=D_rb;d_dstm=4'd15;
55                      end
56
57                  else if(d_icode==4'd4) // rmmovq
58                      begin
59                          d_srca=D_ra;d_srcb=D_rb;d_dste=4'd15;d_dstm=4'd15;
60                      end
61
62                  else if(d_icode==4'd5) // mrmovq
63                      begin
64                          d_srca=4'd15;d_srcb=D_rb;d_dste=4'd15;d_dstm=D_ra;
65                      end
66
67                  else if(d_icode==4'd8) // call
68                      begin
69                          d_srca=4'd15;d_srcb=4;d_dste=4;d_dstm=4'd15;
70                      end
71
72                  else if(d_icode==4'd9) // return
73                      begin
```

```
72                  else if(d_icode==4'd9) // return
73                      begin
74                          d_srca = 4;d_srcb = 4;d_dste = 4;d_dstm = 4'd15;
75                      end
76
77                  else if(d_icode==4'd10) // pushq
78                      begin
79                          d_srca = D_ra;d_srcb = 4;d_dste = 4;d_dstm = 4'd15;
80                      end
81
82                  else if(d_icode==4'd11) // popq
83                      begin
84                          d_srca = 4;d_srcb = 4;d_dste = 4;d_dstm = D_ra;
85                      end
86
87              if(D_icode==4'd7 | D_icode==4'd8)
88                  d_vala=D_valp;
89              else if(d_srca!=15)
90                  d_vala=register_file[d_srca];
91              else
92                  d_vala=0;
93
94              if(d_srcb!=15)
95                  d_valb=register_file[d_srcb];
96              else
97                  d_valb=0;
98
99              // sel+fwd A
100
101             if(d_srca!=15)
102             begin
103                 if(d_srca==e_dste)
104                     d_vala=e_vale;
105                 else if(d_srca==M_dstm)
106                     d_vala=m_valm;
```

```
107                 else if(d_srca==M_dste)
108                     d_vala=M_vale;
109                 else if(d_srca==W_dste)
110                     d_vala=W_vale;
111                 else if(d_srca==W_dstm)
112                     d_vala=W_valm;
113                 else
114                     d_vala=vala;
115             end
116 
117             //fwd B
118             if(d_srcb!=15)
119             begin
120                 if(d_srcb==e_dste)
121                     d_valb=e_vale;
122                 else if(d_srcb==M_dstm)
123                     d_valb=m_valm;
124                 else if(d_srcb==M_dste)
125                     d_valb=M_vale;
126                 else if(d_srcb==W_dste)
127                     d_valb=W_vale;
128                 else if(d_srcb==W_dstm)
129                     d_valb=W_valm;
130                 else
131                     d_valb=valb;
132             end
133 
134         // end
135     end
136     endmodule
```

2. DECODE

Function: Decodes the fetched instruction and prepares it for execution.

Tasks:

- Identify the instruction type and decode its opcode and operands.
- Retrieve register values or immediate data for the instruction.
- Generate control signals based on the decoded instruction.
- Prepare the instruction for execution by providing necessary data.

Key Components:

- Instruction Decoder: Analyzes the opcode and operands to determine the instruction type.
- Register File: Provides access to register values for the instruction.
- Immediate Data Source: Supplies immediate values for instructions that require them.
- Control Signal Generator: Produces control signals based on the decoded instruction.

Operation:

- Receive the fetched instruction from the previous stage.
- Analyze the opcode and operands to identify the instruction type.
- Retrieve register values or immediate data required for the instruction.
- Generate control signals such as ALU operation type, register write enable, memory read/write signals, etc.
- Prepare the instruction for execution by organizing the necessary data and control signals for the next stage.

In the pipelined Decode Stage, the processor efficiently processes instructions by decoding them into their respective operations and preparing the necessary data for execution. By accurately identifying the instruction type, retrieving required data, and generating control signals, the Decode Stage sets the stage for the subsequent Execute Stage to perform the specified operation on the provided inputs.

```
module decode(
    input clock,
    input [3:0] D_icode, D_ifun, D_ra, D_rb,
    input [63:0] D_valp,D_valc,
    input [1:0] D_status,
    input [63:0] r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14,
    input [63:0] e_vale, m_valm, M_vale, W_valm, W_vale,
    input [3:0] e_dste, M_dstm, M_dste, W_dstm, W_dste,
    output reg [63:0] d_vala, d_valb,d_valc,
    output reg [1:0] d_status,
    output reg [3:0] d_dste, d_dstm, d_srca, d_srcb, d_icode, d_ifun
);

reg [63:0] register_file [0:14];
reg [63:0] vala,valb;

always @(*)
begin

        d_icode=D_icode;
        d_ifun=D_ifun;
        d_status=D_status;
        d_valc=D_valc;
    // if(clock)
    // begin
        register_file[0] = r0;
        register_file[1] = r1;
        register_file[2] = r2;
        register_file[3] = r3;
        register_file[4] = r4;
        register_file[5] = r5;
        register_file[6] = r6;
        register_file[7] = r7;
        register_file[8] = r8;
        register_file[9] = r9;
```

```
37              register_file[11] = r11;
38              register_file[12] = r12;
39              register_file[13] = r13;
40              register_file[14] = r14;
41
42                  if(d_icode==4'd0 | d_icode==4'd1 | d_icode==4'd7) //halt,nop,jump
43                      begin
44                          d_srca=4'd15;d_srcb=4'd15;d_dste=4'd15;d_dstm=4'd15;
45                      end
46
47                  else if(d_icode==4'd2 | d_icode==4'd6) // cmovxx
48                      begin
49                          d_srca=D_ra;d_srcb=D_rb;d_dste=D_rb;d_dstm=4'd15;
50                      end
51
52                  else if(d_icode==4'd3)  //irmovq
53                      begin
54                          d_srca=4'd15;d_srcb=D_rb;d_dste=D_rb;d_dstm=4'd15;
55                      end
56
57                  else if(d_icode==4'd4) // rmmovq
58                      begin
59                          d_srca=D_ra;d_srcb=D_rb;d_dste=4'd15;d_dstm=4'd15;
60                      end
61
62                  else if(d_icode==4'd5) // mrmovq
63                      begin
64                          d_srca=4'd15;d_srcb=D_rb;d_dste=4'd15;d_dstm=D_ra;
65                      end
66
67                  else if(d_icode==4'd8) // call
68                      begin
69                          d_srca=4'd15;d_srcb=4;d_dste=4;d_dstm=4'd15;
70                      end
71
72                  else if(d_icode==4'd9) // return
```

```
                begin
                    d_srca = 4;d_srcb = 4;d_dste = 4;d_dstm = 4'd15;
                end

            else if(d_icode==4'd10) // pushq
                begin
                    d_srca = D_ra;d_srcb = 4;d_dste = 4;d_dstm = 4'd15;
                end

            else if(d_icode==4'd11) // popq
                begin
                    d_srca = 4;d_srcb = 4;d_dste = 4;d_dstm = D_ra;
                end

        if(D_icode==4'd7 | D_icode==4'd8)
            d_vala=D_valp;
        else if(d_srca!=15)
            d_vala=register_file[d_srca];
        else
            d_vala=0;

        if(d_srcb!=15)
            d_valb=register_file[d_srcb];
        else
            d_valb=0;

        // sel+fwd A

        if(d_srca!=15)
        begin
            if(d_srca==e_dste)
                d_vala=e_vale;
            else if(d_srca==M_dstm)
```

```
105                 else if(d_srca==M_dstm)
106                     d_vala=m_valm;
107                 else if(d_srca==M_dste)
108                     d_vala=M_vale;
109                 else if(d_srca==W_dste)
110                     d_vala=W_vale;
111                 else if(d_srca==W_dstm)
112                     d_vala=W_valm;
113                 else
114                     d_vala=vala;
115             end
116 
117             //fwd B
118             if(d_srcb!=15)
119             begin
120                 if(d_srcb==e_dste)
121                     d_valb=e_vale;
122                 else if(d_srcb==M_dstm)
123                     d_valb=m_valm;
124                 else if(d_srcb==M_dste)
125                     d_valb=M_vale;
126                 else if(d_srcb==W_dste)
127                     d_valb=W_vale;
128                 else if(d_srcb==W_dstm)
129                     d_valb=W_valm;
130                 else
131                     d_valb=valb;
132             end
133 
134         // end
135     end
136     endmodule
```

3. EXECUTE

Function: The Execute stage in a pipelined processor is responsible for executing the operation specified by the decoded instruction. The primary function of the Execute stage includes performing arithmetic and logical operations, data manipulation, control flow operations, and generating results for further processing.

Tasks:

- **Execute Operation**: Perform the operation specified by the instruction, such as arithmetic calculations, logical operations, data manipulation, and control flow operations.
- **Process Data**: Manipulate data operands based on the operation to generate the desired result.
- **Evaluate Control Flow**: Determine the next instruction address for control flow operations like branching or jumping.
- **Update Flags**: Modify status flags based on the operation result for conditional branching or other control purposes.

Key Components:

- **ALU (Arithmetic Logic Unit)**: Executes arithmetic and logical operations on data operands.
- **Control Unit**: Manages the execution of different instruction types and controls the flow of data within the stage.
- **Register File Access**: Retrieves operand values from registers and updates registers with computation results.

Operation:

- Receive the decoded instruction from the Decode stage.
- Execute the specified operation using the ALU and process data operands accordingly.
- Evaluate control flow conditions and determine the next instruction address for branching or jumping instructions.
- Update status flags based on the operation result for conditional branching or other control purposes.
- Generate the result of the operation to be passed to the next stage for further processing.

```v
`include "./ALU/ALU.v"
`include "./ALU/decoder.v"
`include "./ALU/fulladder.v"
`include "./ALU/subt.v"
`include "./ALU/XOR.v"
`include "./ALU/adder.v"
`include "./ALU/AND.v"

module execute(
    input clock,set_cc,
    input [3:0] E_icode,E_ifun,E_dste,E_dstm,
    input [1:0] E_status,
    input signed [63:0] E_vala, E_valb, E_valc,
    output reg [3:0] e_icode,
    output reg [1:0] e_status,
    output reg [63:0] e_vale,e_vala,
    output reg [3:0] e_dste,e_dstm,
    output reg e_cnd,
    // input clock,
    // input signed [63:0] vala, valb, valc,
    // input [3:0] icode, ifun,
    // output reg [63:0] vale,
    // output reg condition_cnd,
    output reg overflow_flag,
    output reg sign_flag,
    output reg zero_flag
);

reg signed [63:0] input_A, input_B;
reg c0, c1;
wire overflow;
wire signed [63:0] output_alu;

ALU alu0(
    .c0(c0),
    .c1(c1),
```

```
43      initial begin
44          overflow_flag = 1'b0;
45          sign_flag = 1'b0;
46          zero_flag = 1'b0;
47          e_cnd = 1'b0;
48      end
49
50      always @(*) begin
51
52          e_cnd=0;
53
54          // if(clock==1)
55          // begin
56              //test
57              // e_cnd=0;
58
59              case (E_icode)
60
61                  4'd2:
62                      begin
63                          {c1, c0} = 2'b00;
64                          input_A = E_vala;
65                          input_B = 64'd0;
66                          e_vale=output_alu;
67                          if(E_ifun==4'd0)
68                          begin
69                              e_cnd=1'b1;
70                          end
71                          else if(E_ifun==4'd1)
72                          begin
73                              e_cnd=((sign_flag^overflow_flag) | zero_flag);
74                          end
75                          else if(E_ifun==4'd2)
76                          begin
77                              e_cnd=(sign_flag^overflow_flag);
78                          end
```

```
78                      end
79                      else if(E_ifun==4'd3)
80                      begin
81                          e_cnd=zero_flag;
82                      end
83                      else if(E_ifun==4'd4)
84                      begin
85                          e_cnd=~zero_flag;
86                      end
87                      else if(E_ifun==4'd5)
88                      begin
89                          e_cnd=~(sign_flag^overflow_flag);
90                      end
91                      else if(E_ifun==4'd6)
92                      begin
93                          e_cnd=~((sign_flag ^ overflow_flag) | zero_flag);;
94                      end
95
96                      if(e_cnd==0)
97                          e_dste=4'd15;
98                      else
99                          e_dste=E_dste;
100                 end
101
102             4'd3:
103                 begin
104                     {c1, c0} = 2'b00;
105                     input_A = E_valc;
106                     input_B = 64'd0;
107                     e_vale=output_alu;
108                     e_dste=E_dste;
109                 end
110
111             4'd4:
112                 begin
113                     {c1, c0} = 2'b00;
```

```
                    input_A = E_valc;
                    input_B = E_valb;
                    e_vale=output_alu;
                    e_dste=E_dste;
                end

            4'd5:
                begin
                    {c1, c0} = 2'b00;
                    input_A = E_valc;
                    input_B = E_valb;
                    e_vale=output_alu;
                    e_dste=E_dste;
                end

            4'd6:
                begin
                    {c1, c0} = E_ifun[1:0];
                    input_A = E_valb;
                    input_B = E_vala;
                    e_vale=output_alu;
                    e_dste=E_dste;

                    if(set_cc==1)
                    begin

                        sign_flag = (output_alu[63]);
                        overflow_flag = overflow;

                        if(output_alu==64'b0)
                            zero_flag=1;
                        else
                            zero_flag=0;
                        // zero_flag = (output_alu==64'b0);
                    end
                end
```

```
150
151                 4'd7:
152                     begin
153                         if(E_ifun==4'd0)
154                             e_cnd=1'b1;
155                         else if(E_ifun==4'd1)
156                             e_cnd=((sign_flag^overflow_flag) | zero_flag);
157                         else if(E_ifun==4'd2)
158                             e_cnd=(sign_flag^overflow_flag);
159                         else if(E_ifun==4'd3)
160                             e_cnd=zero_flag;
161                         else if(E_ifun==4'd4)
162                             e_cnd=~zero_flag;
163                         else if(E_ifun==4'd5)
164                             e_cnd=~(sign_flag ^ overflow_flag);
165                         else if(E_ifun==4'd6)
166                             e_cnd=~((sign_flag ^ overflow_flag) | zero_flag);
167
168                         e_dste=E_dste;
169                     end
170
171                 4'd8:
172                     begin
173                         {c1, c0} = 2'b01;
174                         input_A = E_valb;
175                         input_B = 64'd8;
176                         e_vale=output_alu;
177                         e_dste=E_dste;
178                     end
179
180                 4'd9:
181                     begin
182                         {c1, c0} = 2'b00;
183                         input_A = 64'd8;
184                         input_B = E_valb;
185                         e_vale=output_alu;
```

```
184                         input_B = E_valb;
185                         e_vale=output_alu;
186                         e_dste=E_dste;
187                     end
188 
189                 4'd10:
190                     begin
191                         {c1, c0} = 2'b01;
192                         input_A = E_valb;
193                         input_B = 64'd8;
194                         e_vale=output_alu;
195                         e_dste=E_dste;
196                     end
197 
198                 4'd11:
199                     begin
200                         {c1, c0} = 2'b00;
201                         input_A = 64'd8;
202                         input_B = E_valb;
203                         e_vale=output_alu;
204                         e_dste=E_dste;
205                     end
206             endcase
207 
208             e_icode=E_icode;
209             e_dstm=E_dstm;
210             e_status=E_status;
211             e_vala=E_vala;
212 
213             // if(E_icode!=4'd2)
214             //     e_dste=E_dste;
215         end
216     // end
217 
218 
219     endmodule
```

4. MEMORY

Function: The Memory stage in a pipelined processor is responsible for reading from and writing to memory. It handles data transfers between the processor and memory, including loading data into memory, retrieving data from memory, and managing memory-related operations.

Tasks:

- Memory Access: Read data from or write data to the memory based on the instruction requirements.
- Data Transfer: Transfer data between the processor and memory for instructions like load/store operations.
- Address Calculation: Calculate memory addresses based on the instruction and operand values.
- Data Handling: Manage data input/output operations and data manipulation during memory access.

Key Components:

- Data Memory: Represents the memory array where data is stored and retrieved during memory operations.
- Memory Address Calculation Unit: Computes memory addresses based on the instruction and operand values.
- Data Transfer Control: Manages the data transfer process between the processor and memory.

Operation:

- Receive the necessary data and control signals from the Execute stage.
- Determine whether the instruction requires a memory read or write operation.
- Calculate the memory address based on the instruction and operand values.
- Perform the memory read/write operation as specified by the instruction.
- Update data memory with new values or retrieve data from memory based on the instruction type.
- Prepare the data for further processing in the pipeline stages or for writing back to registers.

In the pipelined Memory Stage, the processor efficiently handles memory-related operations by reading from and writing to memory, managing data transfers, calculating memory addresses, and ensuring proper data handling during memory access.

```
module memory(
    input clock,
    input [3:0] M_icode,M_dste,M_dstm,
    input [63:0] M_vala,M_vale,
    input [1:0] M_status,
    output reg [1:0] m_status,
    output reg [3:0] m_icode,m_dste,m_dstm,
    output reg memory_block_error,
    output reg [63:0] m_vale,m_valm,written_mem
);


reg [63:0] memory_register [0:1023];

initial begin
    memory_block_error=0;
    //modif
   // m_valm=0;
end

always@(*)
begin
    if(M_icode==4'd5) //mrmovq
    begin
        if(M_vale>1023)
            memory_block_error=1;
        else
            m_valm=memory_register[M_vale];
    end

    if(M_icode==4'd9) //return
    begin
        if(M_vala>1023)
            memory_block_error=1;
        else
            m_valm=memory_register[M_vala];
    end
```

```
39          if(M_icode==4'd11) //popq
40          begin
41              if(M_vala>1023)
42                  memory_block_error=1;
43              else
44                  m_valm=memory_register[M_vala];
45          end
46
47      end
48
49      // always@(*)
50      // begin
51      // end
52
53      // always@(negedge clock)
54      always@(*)
55      begin
56          if(M_vale>1023)
57              memory_block_error=1;
58          else
59          begin
60              if(M_icode==4'd4) //rmmovq
61              begin
62                  memory_register[M_vale]=M_vala;
63                  written_mem=memory_register[M_vale];
64              end
65
66              if(M_icode==4'd8) //call
67              begin
68                  memory_register[M_vale]=M_vala;
69                  written_mem=memory_register[M_vale];
70              end
71
72              if(M_icode==4'd10) //pushq
73              begin
```

```
78      end
79
80      always@(*)begin
81          m_vale=M_vale;
82          m_icode=M_icode;
83          m_dste=M_dste;
84          m_dstm=M_dstm;
85      end
86
87      always@(*)
88      begin
89          if(memory_block_error==1)
90              m_status=3;
91          else
92              m_status=M_status;
93      end
94
95      endmodule
```

5. WRITEBACK

Function: The Write Back stage in a pipelined processor is responsible for writing the results of the executed instruction back to the register file. The primary function of the Write Back stage includes updating register values with the computation results and preparing the return address for the Program Counter (PC) selection logic.

Tasks:

- Write Result to Register: Update the register file with the computation result generated in the Execute stage.
- Prepare Return Address: Determine the return address for the PC selection logic, especially for instructions like 'ret'.
- Handle Register Write Enable: Control the write enable signal to ensure the correct register is updated with the result.

Key Components:

- Register File: Provides access to registers for writing back the computation results.
- Return Address Logic: Determines the return address for the PC selection logic.
- Write Enable Control: Manages the write enable signal to update the appropriate register.

Operation:

- Receive the computation result from the Execute stage.
- Write back the result to the designated register in the register file.
- Prepare the return address for the PC selection logic, especially for 'ret' instructions.
- Control the write enable signal to ensure the correct register is updated with the result.
- Complete the instruction execution cycle by finalizing the register updates and return address preparation for subsequent instructions.

```
module writeback(
    input clock,
    input [3:0] W_dste,W_dstm,W_icode,
    input [63:0] W_vale,W_valm,
    input [1:0] W_status,
    output reg [63:0] r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,r10,r11,r12,r13,r14
);

reg[63:0] register_file[0:14];

// always@(negedge clock)
always@(posedge clock)
begin
    if(W_status==0)
    begin
        if(W_dste != 15) register_file[W_dste] = W_vale;
        if(W_dstm != 15) register_file[W_dstm] = W_valm;
    end

    r0=register_file[0];
    r1=register_file[1];
    r2=register_file[2];
    r3=register_file[3];
    r4=register_file[4];
    r5=register_file[5];
    r6=register_file[6];
    r7=register_file[7];
    r8=register_file[8];
    r9=register_file[9];
    r10=register_file[10];
    r11=register_file[11];
    r12=register_file[12];
    r13=register_file[13];
    r14=register_file[14];
end

```

CONTROL LOGIC,DATA FORWARDING AND HAZARDS

Pipeline Control Logic: Pipeline control logic in a processor manages the flow of instructions through the pipeline stages to ensure correct execution and maintain pipeline efficiency. The primary functions of pipeline control logic include instruction fetching, decoding, executing, and writing back results. It also handles control hazards, data hazards, and structural hazards to prevent conflicts and ensure smooth instruction execution.

Tasks:

- ***Instruction Fetching***: Controls the fetching of instructions from memory and their entry into the pipeline.
- ***Instruction Decoding***: Manages the decoding of instructions and their preparation for execution.
- ***Execution Control***: Coordinates the execution of instructions in the pipeline stages.
- ***Write Back Control****:* Handles the writing back of results to registers or memory.
- ***Branch Prediction*****:** Predicts branch outcomes to minimize pipeline stalls due to branch instructions.

Key Components:

- ***Control Signals***: Signals generated at each pipeline stage to control the flow of instructions.
- ***Pipeline Registers***: Store intermediate results between pipeline stages.
- ***Branch Prediction Unit***: Predicts branch outcomes to reduce pipeline stalls.
- ***Hazard Detection Unit***: Identifies hazards and stalls in the pipeline.

Operation:

- ***Instruction Flow Control***: Ensures instructions enter the pipeline in the correct order and progress through stages smoothly.
- ***Hazard Detection***: Detects hazards such as data hazards, control hazards, and structural hazards that can impact pipeline performance.
- ***Stall Management*****:** Inserts pipeline stalls or bubbles to resolve hazards and maintain data dependencies.
- ***Control Signal Generation***: Generates control signals to coordinate the execution of instructions in different pipeline stages.

- ***Branch Prediction:*** Predicts branch outcomes to minimize pipeline flushes and stalls caused by branch instructions.

Data Forwarding: Data forwarding, also known as data bypassing, is a technique used in pipelined processors to resolve data hazards by forwarding data directly from the output of one pipeline stage to the input of another stage without waiting for it to be written back to the register file. This technique helps in maintaining pipeline efficiency by reducing stalls caused by data dependencies between instructions.

Tasks:

- ***Detect Data Hazards***: Identify situations where an instruction requires data that is produced by a previous instruction still in the pipeline.
- ***Forward Data:*** Directly transfer data from the producing instruction to the consuming instruction to resolve data hazards.
- ***Control Data Paths:*** Manage the data forwarding paths and ensure correct data is forwarded to the dependent instructions.

Key Components:

- ***Data Forwarding Paths:*** Routes data from the producing instruction to the consuming instruction.
- ***Data Hazard Detection Unit:*** Identifies data hazards and triggers data forwarding when needed.
- ***Control Logic:*** Manages the data forwarding process and ensures correct data is forwarded.

Operation:

- ***Data Dependency Detection:*** Detect dependencies between instructions that require data produced by previous instructions.
- ***Data Forwarding Activation:*** Activate data forwarding paths to directly transfer data to dependent instructions.
- ***Data Path Control:*** Ensure that the correct data is forwarded to the dependent instruction to resolve data hazards and prevent stalls.
- ***Pipeline Efficiency:*** Improve pipeline efficiency by reducing stalls caused by data dependencies and allowing instructions to proceed without waiting for data to be written back to registers.

Hazards: Hazards in a pipelined processor refer to situations that can potentially disrupt the smooth execution of instructions and impact pipeline performance. There are three main types of hazards:

- ***Data Hazards:*** Arise when an instruction depends on the result of a previous instruction that has not yet produced the result.
- ***Control Hazards:*** Occur when the pipeline must stall or flush instructions due to branch mispredictions or other control flow changes.
- ***Structural Hazards:*** Arise from resource conflicts when multiple instructions require the same hardware resource simultaneously.

Handling Hazards:

- ***Stall Insertion:*** Introduce pipeline stalls to resolve hazards and ensure correct data dependencies.
- ***Data Forwarding:*** Use data forwarding to directly transfer data between pipeline stages and resolve data hazards.
- ***Branch Prediction:*** Predict branch outcomes to minimize stalls caused by control hazards.
- ***Instruction Reordering:*** Reorder instructions to reduce hazards and improve pipeline efficiency.

By effectively managing pipeline control logic, implementing data forwarding techniques, and addressing hazards, a pipelined processor can enhance performance, reduce stalls, and optimize instruction execution in a pipeline architecture.

WRAPPER MODULE INPUTS AND OUTPUTS

```
1     irmovq $10, %rsp
2     irmovq $5 , %rax
3     irmovq $4 , %rbx
4     call NF
5     xorq %rax, %rbx
6     halt
7
8     NF:
9         addq %rax , %rbx
10        ret
11    addq %rax, %rbx
```

```
clock=0, f_pc=                  0, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  4, f_status=0, D_icode =  x, D_ifun =  x, D_rA =  x, D_rB =  x, D_valp=
clock=1, f_pc=                 10, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  0, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  4, D_valp=
clock=0, f_pc=                 10, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  0, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  4, D_valp=
clock=1, f_pc=                 20, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  3, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  0, D_valp=
clock=0, f_pc=                 20, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  3, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  0, D_valp=
clock=1, f_pc=                 30, f_icode =  8, f_ifun =  0, f_rA = 15, f_rB =  3, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  3, D_valp=
clock=0, f_pc=                 30, f_icode =  8, f_ifun =  0, f_rA = 15, f_rB =  3, f_status=0, D_icode =  3, D_ifun =  0, D_rA = 15, D_rB =  3, D_valp=
clock=1, f_pc=                 42, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  8, D_ifun =  0, D_rA = 15, D_rB =  3, D_valp=
clock=0, f_pc=                 42, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  8, D_ifun =  0, D_rA = 15, D_rB =  3, D_valp=
clock=1, f_pc=                 44, f_icode =  9, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  6, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 44, f_icode =  9, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  6, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  9, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  9, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 39, f_icode =  6, f_ifun =  3, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 39, f_icode =  6, f_ifun =  3, f_rA =  0, f_rB =  3, f_status=0, D_icode =  1, D_ifun =  0, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 41, f_icode =  0, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  6, D_ifun =  3, D_rA =  0, D_rB =  3, D_valp=
clock=0, f_pc=                 41, f_icode =  0, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  6, D_ifun =  3, D_rA =  0, D_rB =  3, D_valp=
clock=1, f_pc=                 42, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
clock=0, f_pc=                 42, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
clock=1, f_pc=                 44, f_icode =  9, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
clock=0, f_pc=                 44, f_icode =  9, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
clock=1, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
clock=0, f_pc=                 45, f_icode =  6, f_ifun =  0, f_rA =  0, f_rB =  3, f_status=1, D_icode =  0, D_ifun =  0, D_rA = 15, D_rB = 15, D_valp=
z_processor.v:310: $finish called at 270 (1s)
```

```
D_valp=   x, D_valc=   x, D_status=x, E_icode= x, E_ifun= x, E_srca= x, E_srcb= x, E_dste= x, E_dstm= x,
D_valp=  10, D_valc=  39, D_status=0, E_icode= x, E_ifun= x, E_srca= x, E_srcb= x, E_dste= x, E_dstm= x,
D_valp=  10, D_valc=  39, D_status=0, E_icode= x, E_ifun= x, E_srca= x, E_srcb= x, E_dste= x, E_dstm= x,
D_valp=  20, D_valc=   5, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  20, D_valc=   5, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  30, D_valc=   4, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 0, E_dste= 0, E_dstm=15,
D_valp=  30, D_valc=   4, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 0, E_dste= 0, E_dstm=15,
D_valp=  39, D_valc=  42, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=  39, D_valc=  42, D_status=0, E_icode= 3, E_ifun= 0, E_srca=15, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=  44, D_valc=  42, D_status=0, E_icode= 8, E_ifun= 0, E_srca=15, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  44, D_valc=  42, D_status=0, E_icode= 8, E_ifun= 0, E_srca=15, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 6, E_ifun= 0, E_srca= 0, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 6, E_ifun= 0, E_srca= 0, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 9, E_ifun= 0, E_srca= 4, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 9, E_ifun= 0, E_srca= 4, E_srcb= 4, E_dste= 4, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=  45, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=  41, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=  41, D_valc=  42, D_status=0, E_icode= 1, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 6, E_ifun= 3, E_srca= 0, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 6, E_ifun= 3, E_srca= 0, E_srcb= 3, E_dste= 3, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 0, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 0, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 0, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
D_valp=   0, D_valc=   0, D_status=1, E_icode= 0, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,

D_valp=   0, D_valc=   0, D_status=1, E_icode= 0, E_ifun= 0, E_srca=15, E_srcb=15, E_dste=15, E_dstm=15,
```

```
E_vala=   x, E_valb=   x, E_valc=   x, E_status=x, e_cnd=0, M_icode= x, M_status=x, M_cnd=x,
E_vala=   x, E_valb=   x, E_valc=   x, E_status=x, e_cnd=0, M_icode= x, M_status=x, M_cnd=0,
E_vala=   x, E_valb=   x, E_valc=   x, E_status=x, e_cnd=0, M_icode= x, M_status=x, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=  39, E_status=0, e_cnd=0, M_icode= x, M_status=x, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=  39, E_status=0, e_cnd=0, M_icode= x, M_status=x, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=   5, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=   5, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=   4, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   x, E_valc=   4, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=  39, E_valb=  39, E_valc=  42, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=  39, E_valb=  39, E_valc=  42, E_status=0, e_cnd=0, M_icode= 3, M_status=0, M_cnd=0,
E_vala=   5, E_valb=   4, E_valc=  42, E_status=0, e_cnd=0, M_icode= 8, M_status=0, M_cnd=0,
E_vala=   5, E_valb=   4, E_valc=  42, E_status=0, e_cnd=0, M_icode= 8, M_status=0, M_cnd=0,
E_vala=  31, E_valb=  31, E_valc=  42, E_status=0, e_cnd=0, M_icode= 6, M_status=0, M_cnd=0,
E_vala=  31, E_valb=  31, E_valc=  42, E_status=0, e_cnd=0, M_icode= 6, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 9, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 9, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   x, E_valb=   x, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   x, E_valb=   x, E_valc=  42, E_status=0, e_cnd=0, M_icode= 1, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=   0, E_status=1, e_cnd=0, M_icode= 6, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=   0, E_status=1, e_cnd=0, M_icode= 6, M_status=0, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=   0, E_status=1, e_cnd=0, M_icode= 0, M_status=1, M_cnd=0,
E_vala=   0, E_valb=   0, E_valc=   0, E_status=1, e_cnd=0, M_icode= 0, M_status=1, M_cnd=0,

E_vala=   0, E_valb=   0, E_valc=   0, E_status=1, e_cnd=0, M_icode= 1, M_status=1, M_cnd=0,
```

```
M_vale=            x, M_vala=            x, M_dste= x, M_dstm= x, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=            x, M_vala=            x, M_dste= x, M_dstm= x, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=            x, M_vala=            x, M_dste= x, M_dstm= x, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=            x, M_vala=            x, M_dste= x, M_dstm= x, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=            x, M_vala=            x, M_dste= x, M_dstm= x, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= x, W_dste= x, W_dstm= x, W_vale=            x,
M_vale=            5, M_vala=            0, M_dste= 0, M_dstm=15, W_icode= 3, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=            5, M_vala=            0, M_dste= 0, M_dstm=15, W_icode= 3, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=            4, M_vala=            0, M_dste= 3, M_dstm=15, W_icode= 3, W_dste= 0, W_dstm=15, W_vale=            5,
M_vale=            4, M_vala=            0, M_dste= 3, M_dstm=15, W_icode= 3, W_dste= 0, W_dstm=15, W_vale=            5,
M_vale=           31, M_vala=           39, M_dste= 4, M_dstm=15, W_icode= 3, W_dste= 3, W_dstm=15, W_vale=            4,
M_vale=           31, M_vala=           39, M_dste= 4, M_dstm=15, W_icode= 3, W_dste= 3, W_dstm=15, W_vale=            4,
M_vale=            9, M_vala=            5, M_dste= 3, M_dstm=15, W_icode= 8, W_dste= 4, W_dstm=15, W_vale=           31,
M_vale=            9, M_vala=            5, M_dste= 3, M_dstm=15, W_icode= 8, W_dste= 4, W_dstm=15, W_vale=           31,
M_vale=           39, M_vala=           31, M_dste= 4, M_dstm=15, W_icode= 6, W_dste= 3, W_dstm=15, W_vale=            9,
M_vale=           39, M_vala=           31, M_dste= 4, M_dstm=15, W_icode= 6, W_dste= 3, W_dstm=15, W_vale=            9,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 9, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 9, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=           39, M_vala=            0, M_dste= 4, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=            x, M_vala=            x, M_dste= 3, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=            x, M_vala=            x, M_dste= 3, M_dstm=15, W_icode= 1, W_dste= 4, W_dstm=15, W_vale=           39,
M_vale=            x, M_vala=            0, M_dste= 3, M_dstm=15, W_icode= 6, W_dste= 3, W_dstm=15, W_vale=            x,
M_vale=            x, M_vala=            0, M_dste= 3, M_dstm=15, W_icode= 6, W_dste= 3, W_dstm=15, W_vale=            x,

M_vale=            x, M_vala=            0, M_dste= 3, M_dstm=15, W_icode= 0, W_dste= 3, W_dstm=15, W_vale=            x,
```

```
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=            x,
r0=            x, r1=            x, r2=            x, r3=            x, r4=           39,
r0=            x, r1=            x, r2=            x, r3=            x, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            x, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            x, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            4, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            4, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            4, r4=           31,
r0=            5, r1=            x, r2=            x, r3=            4, r4=           31,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           31,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           31,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,
r0=            5, r1=            x, r2=            x, r3=            9, r4=           39,

r0=            5, r1=            x, r2=            x, r3=            x, r4=           39,
```

```
irmovq $0, %rax
irmovq $10, %rdx
irmovq $1, %rsi

loop:
    addq %rdx, %rax
    subq %rsi, %rdx
    andq %rdx, %rdx
    jne loop

halt
```

```
clock=0, f_pc=                0, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  3,
clock=1, f_pc=               10, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  2,
clock=0, f_pc=               10, f_icode =  3, f_ifun =  0, f_rA = 15, f_rB =  2,
clock=1, f_pc=               20, f_icode =  6, f_ifun =  0, f_rA =  2, f_rB =  3,
clock=0, f_pc=               20, f_icode =  6, f_ifun =  0, f_rA =  2, f_rB =  3,
clock=1, f_pc=               22, f_icode =  0, f_ifun =  0, f_rA =  2, f_rB =  3,
clock=0, f_pc=               22, f_icode =  0, f_ifun =  0, f_rA =  2, f_rB =  3,
clock=1, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
clock=0, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
clock=1, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
clock=0, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
clock=1, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
clock=0, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
z_processor.v:310: $finish called at 130 (1s)
clock=1, f_pc=               23, f_icode =  x, f_ifun =  x, f_rA =  2, f_rB =  3,
```

```
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                  x,
r1=                  x, r2=                  x, r3=                256,
r1=                  x, r2=                  x, r3=                256,
r1=                  x, r2=                512, r3=                256,
r1=                  x, r2=                512, r3=                256,

r1=                  x, r2=                512, r3=                768,
```