Intro to Processor Architecture Project

Sannidhya Gupta

2021112012

Krishna

2021112005

Overview

The objective of this project is to develop a processor architecture design based on the Y86-64 ISA using Verilog. The processor should be able to execute all instructions in the Y86-64 ISA. Modular approach is to be followed. The goal is to produce a 5 stage pipelined Y86-64 implementation.

The program counter (PC) is a crucial component of a sequential processor architecture, as it keeps track of the next instruction to be executed. The PC is a register that stores the memory address of the instruction that the processor is currently executing or about to execute. Once an instruction is processed, the program counter is updated to point to the next instruction.

The clock cycle is a fundamental component of a sequential processor architecture, as it controls the timing of the fetch-execute cycle. The clock cycle is a fixed-length interval of time during which the processor performs a set of operations.

During each clock cycle, the processor performs a sequence of operations, including fetching the instruction from memory, decoding the instruction to determine the operation to be performed, executing the operation, and storing the result. The clock cycle also updates the program counter to point to the next instruction to be executed.

Sequential Processor Architecture

In a sequential processor architecture, there is a single processor that performs both arithmetic and logic operations, as well as the control functions needed to execute a program. The processor reads instructions from memory one at a time, executes them in sequence, and then moves on to the next instruction.

Modules :

Fetch

- The fetch stage retrieves the instruction from memory using the address stored in the program counter, which points to the memory location where the instruction is stored. Once the instruction is fetched, the program counter is incremented to point to the next instruction.
- The fetched instruction is then stored in an instruction register, which is a temporary storage location for the instruction while it is being executed. The instruction register holds the instruction until it is decoded and executed in the subsequent stages of the instruction cycle. The fetch stage is critical to the operation of a sequential processor, as it initiates the instruction cycle and sets the stage for the execution of the instruction.
- The instruction is to be read from instruction memory. We have made a separate module for `instruction memory`. For an instruction, the values of `icode` , `ifun`, `rA`, `rB` and `valC` are to be found. `valP` is to be calculated.
- Instruction Memory Module:

```
`timescale 1ns/10ps

module insMem(PC,instruction);

input [63:0] PC;
reg [7:0] insMem [0:127];
output reg [7:0] instruction;

initial begin
    insMem[  0] = 8'hA0;
    insMem[  1] = 8'h64;
    insMem[  2] = 8'h60;
    insMem[  3] = 8'h89;
    insMem[  4] = 8'h60;
    insMem[  5] = 8'h90;
    insMem[  6] = 8'h00;
    insMem[  7] = 8'h00;
    insMem[  8] = 8'h00;
    insMem[  9] = 8'h00;
    insMem[ 10] = 8'h00;
    insMem[ 11] = 8'h00;
    insMem[ 12] = 8'h00;
    insMem[ 13] = 8'h00;
    insMem[ 14] = 8'h00;
    insMem[ 15] = 8'h60;
    insMem[ 16] = 8'h40;
    insMem[ 17] = 8'h00;
    insMem[ 18] = 8'h00;
    insMem[ 19] = 8'h00;
    insMem[ 20] = 8'h00;
    insMem[ 21] = 8'h00;
    insMem[ 22] = 8'h00;
    insMem[ 23] = 8'h00;
    insMem[ 24] = 8'h00;
    insMem[ 25] = 8'h00;
    insMem[ 26] = 8'h00;
    insMem[ 27] = 8'h00;
    insMem[ 28] = 8'h00;
    insMem[ 29] = 8'h00;
end

always@(*)begin
    instruction = insMem[PC];
end

endmodule
```

- **icode and ifun**

The `icode` field is a 4-bit field that specifies the opcode of an instruction. Thus it specifies the type of operation to be performed. It can take on one of 16 possible values, each of which corresponds to a specific type of instruction.

The `ifun` field is a 4-bit field that specifies the function code of an instruction. It is used to further specify the operation to be performed by the instruction. For example, if the icode value is 2 (corresponding to an irmovl instruction), the ifun field can be used to specify whether the instruction is moving a value to a register or to memory.

- **rA and rB**

  `rA` and `rB` provide a way to specify the registers that are used as operands in an instruction. This allows the Y86 processor to perform operations on data stored in the registers, such as arithmetic or logic operations.

- **valC and valP**

  `ValC` (Value Constant) is a 4-byte field that contains a constant value used by an instruction. It is typically used to specify the immediate value that is used as an operand in an instruction. For example, in an irmovl (immediate move long) instruction, the ValC field contains the value that is being moved into a register.

  `ValP` (Value Program counter) is a 4-byte field that contains the address of the next instruction to be executed. It is used to update the value of the program counter (PC) during the execution of an instruction. For example, after executing a jump instruction, the ValP field would contain the address of the target of the jump, which would then be loaded into the PC to continue program execution.

Code:

```
`timescale 1ns/10ps

`include "./insMem.v"

module fetch (clock, PC,icode, ifun, rA, rB, valC, valP, halt);
// inputs
input clock;
input [63:0] PC;

// memory
reg [63:0] memPCval;
```

```
// outputs
output reg [ 3:0] icode;
output reg [ 3:0] ifun;
output reg [ 3:0] rA;
output reg [ 3:0] rB;
output reg [63:0] valP;
output reg [63:0] valC;
output reg halt;

// icodes
reg [3:0] codeHalt   = 4'b0000;// 0
reg [3:0] codeNop    = 4'b0001;// 1
reg [3:0] codeCmovXX = 4'b0010;// 2
reg [3:0] codeIRmovq = 4'b0011;// 3
reg [3:0] codeRMmovq = 4'b0100;// 4
reg [3:0] codeMRmovq = 4'b0101;// 5
reg [3:0] codeOPq    = 4'b0110;// 6
reg [3:0] codeJXX    = 4'b0111;// 7
reg [3:0] codeCall   = 4'b1000;// 8
reg [3:0] codeRet    = 4'b1001;// 9
reg [3:0] codePushq  = 4'b1010;// A
reg [3:0] codePopq   = 4'b1011;// B

wire [7:0] byte1;
wire [7:0] byte2;
wire [7:0] byte3;
wire [7:0] byte4;
wire [7:0] byte5;
wire [7:0] byte6;
wire [7:0] byte7;
wire [7:0] byte8;
wire [7:0] byte9;
wire [7:0] byte10;

insMem UUTmem1  (PC + 0, byte1);
insMem UUTmem2  (PC + 1, byte2);
insMem UUTmem3  (PC + 2, byte3);
insMem UUTmem4  (PC + 3, byte4);
insMem UUTmem5  (PC + 4, byte5);
insMem UUTmem6  (PC + 5, byte6);
insMem UUTmem7  (PC + 6, byte7);
insMem UUTmem8  (PC + 7, byte8);
insMem UUTmem9  (PC + 8, byte9);
insMem UUTmem10 (PC + 9, byte10);

always @(posedge clock)
begin
  // initial conditions
  rA = 4'b1111;
  rB = 4'b1111;
  valC = 64'd0;
  valP = 64'd0;
  halt = 0;
```

```
    if (PC >= 127) begin
      halt = 1;
    end
    else begin
      // calculating icode and ifun
      icode = byte1[7:4];
      ifun  = byte1[3:0];
      case(icode)
      codeHalt:begin
        halt = 1;
        valP = PC + 1;
      end
      codeNop:begin
        valP = PC + 1;
      end
      codeCmovXX:begin
        rA = byte2[7:4];
        rB = byte2[3:0];
        valP = PC+2;
      end
      codeIRmovq:begin
        rA = byte2[7:4];
        rB = byte2[3:0];
        valC = {byte10,byte9,byte8,byte7,byte6,byte5,byte4,byte3};
        valP = PC+10;
      end
      codeRMmovq:begin
        rA = byte2[7:4];
        rB = byte2[3:0];
        valC = {byte10,byte9,byte8,byte7,byte6,byte5,byte4,byte3};
        valP = PC+10;
      end
      codeMRmovq:begin
        rA = byte2[7:4];
        rB = byte2[3:0];
        valC = {byte10,byte9,byte8,byte7,byte6,byte5,byte4,byte3};
        valP = PC+10;
      end
      codeOPq:begin
        rA = byte2[7:4];
        rB = byte2[3:0];
        valP = PC+2;
      end
      codeJXX:begin
        valC = {byte9,byte8,byte7,byte6,byte5,byte4,byte3,byte2};
        valP = PC + 9;
      end
      codeCall:begin
        valC = {byte9,byte8,byte7,byte6,byte5,byte4,byte3,byte2};
        valP = PC + 9;
      end
      codeRet:begin
        valP = PC + 1;
      end
```

```
    codePushq:begin
      rA = byte2[7:4];
      rB = byte2[3:0];
      valP = PC+2;
    end
    codePopq:begin
      rA = byte2[7:4];
      rB = byte2[3:0];
      valP = PC+2;
    end

  endcase
  end
end

endmodule
```

- Instructions are fetched at the positive edge of the clock.
- Instruction memory bytes are stored in byte1, byte2,……., byte10 (As an instruction can have a maximum of 10 bytes). These are read from the instruction memory module.
- Different if and else-if loops are present to proceed according to the `icode`.
- `icode` and `ifun` are obtained by taking `icode = byte1[7:4]` and `ifun = byte1[3:0]` as they are stored in the first byte of instruction memory.
- `rA` and `rB` are obtained from the second byte of instruction memory by taking `rA = byte2[7:4]` and `rB = byte2[3:0]`, respectively.
- `valC` is obtained from the remaining 8 bytes of an instruction by taking `valC = {byte10,byte9,byte8,byte7,byte6,byte5,byte4,byte3};`
- `valP` is also calculated according to the type of instruction which is further determined by `icode`.

Decode and Write Back

As decode uses register files to read the values whereas write back uses register files to update the registers. We cannot make a separate register module because we can read registers from it but cannot update them in write back. Hence, we combined decode and write back stages in a single Verilog file and put the registers in the same file too.

Decode

- In the decode stage, `valA` and `valB` are updated based on `icode` and the registers `rA` and `rB`.
- The values of `valA` and `valB` depend on `icode` as follows:
  - For Cmovxx,

    `valA = registers[rA]`
  - For Rmmovq and OPq,

    `valA = registers[rA]` and `valB = registers[rB]`
  - For MRmovq,

    `valB = registers[rB]`
  - For Call,

    `valB = registers[rsp]`
  - For ret and Popq,

    `valA = registers[rsp]` and `valB = registers[rsp]`
  - For Pushq,

    `valA = registers[rA]` and `valB = registers[rsp]`

  (Note: `rsp` is the stack pointer register)

Write Back

- The values of `valE` or `valM` are to be updated(written back) into registers `rA`, `rB` or `rsp`, depending on the icode of the given instruction.
- The write back depends on `icode` as follows:
  - For Cmovxx, if `cnd` value is 1,

    `registers[rB] = valE`

    if cnd value is 0,

    `registers[4'hf] = valE` (updating garbage register)
  - For IRmovq and OPq,

    `registers[rB] = valE`

- For MRmovq,

  `registers[rA] = valM`

- For Call, ret and Pushq,

  `registers[rsp] = valE`

- For Popq,

  `registers[rsp] = valE` and `registers[rA] = valM`

Code:

```
`timescale 1ns/10ps

module decode_writeBack(clock, icode, rA, rB, valA, valB, valE, valM, Cnd);

//// decode
// inputs
input clock;
input [3:0] icode;
input [3:0] rA;
input [3:0] rB;
// outputs
output reg [63:0] valA;
output reg [63:0] valB;

//// writeBack
input [63:0] valE;
input [63:0] valM;
input Cnd;

// icodes
reg [3:0] codeHalt   = 4'b0000;// 0
reg [3:0] codeNop    = 4'b0001;// 1
reg [3:0] codeCmovXX = 4'b0010;// 2
reg [3:0] codeIRmovq = 4'b0011;// 3
reg [3:0] codeRMmovq = 4'b0100;// 4
reg [3:0] codeMRmovq = 4'b0101;// 5
reg [3:0] codeOPq    = 4'b0110;// 6
reg [3:0] codeJXX    = 4'b0111;// 7
reg [3:0] codeCall   = 4'b1000;// 8
reg [3:0] codeRet    = 4'b1001;// 9
reg [3:0] codePushq  = 4'b1010;// A
reg [3:0] codePopq   = 4'b1011;// B

// registers
parameter rax = 4'h0;
parameter rcx = 4'h1;
parameter rdx = 4'h2;
```

```
parameter rbx = 4'h3;
parameter rsp = 4'h12;
parameter rbp = 4'h5;
parameter rsi = 4'h6;
parameter rdi = 4'h7;
parameter r8  = 4'h8;
parameter r9  = 4'h9;
parameter r10 = 4'hA;
parameter r11 = 4'hB;
parameter r12 = 4'hC;
parameter r13 = 4'hD;
parameter r14 = 4'hE;
parameter r15 = 4'hF;
reg [63:0] registers [0:15];

initial begin
  valA = 64'd0;
  valB = 64'd0;

  // registers
  registers[rax]=64'h0;
  registers[rcx]=64'h1;
  registers[rdx]=64'h2;
  registers[rbx]=64'h3;
  registers[rsp]=64'h5;
  registers[rbp]=64'h5;
  registers[rsi]=64'h6;
  registers[rdi]=64'h7;
  registers[r8 ]=64'h8;
  registers[r9 ]=64'h9;
  registers[r10]=64'ha;
  registers[r11]=64'hb;
  registers[r12]=64'hc;
  registers[r13]=64'hd;
  registers[r14]=64'he;
  registers[r15]=64'hf;
end

always @(*) begin
  // checking command
  case(icode)
  codeHalt:begin
      // Nothing
  end
  codeNop:begin
    // Nothing
  end
  codeCmovXX:begin
    valA = registers[rA];
  end
  codeIRmovq:begin
    // nothing
  end
  codeRMmovq:begin
```

```
    valA = registers[rA];
    valB = registers[rB];
  end
  codeMRmovq:begin
    valB = registers[rB];
  end
  codeOPq:begin
    valA = registers[rA];
    valB = registers[rB];
  end
  codeCall:begin
    valB = registers[rsp];
  end
  codeRet:begin
    valA = registers[rsp];
    valB = registers[rsp];
  end
  codePushq:begin
    valA = registers[rA];
    valB = registers[rsp];
  end
  codePopq:begin
    valA = registers[rsp];
    valB = registers[rsp];
  end
  endcase
end

////WRITEBACK

// for accessing registers
reg [3:0] regNo;
reg sel_reg_IO;

always @(posedge clock) begin
  sel_reg_IO = 1; // not inputting
end


always @(negedge clock) begin
case(icode)
  codeHalt:begin
    // Nothing
  end
  codeNop:begin
    // Nothing
  end
  codeCmovXX:begin
    if (Cnd == 1) begin
      registers[rB] = valE;
    end
    else begin
      registers[4'hf] = valE;
    end
```

```
    end
    codeIRmovq:begin
      registers[rB] = valE;
    end
    codeRMmovq:begin
      // Nothing
    end
    codeMRmovq:begin
      registers[rA] = valM;
    end
    codeOPq:begin
      registers[rB] = valE;
    end
    codeJXX:begin
      // Nothing
    end
    codeCall:begin
      registers[rsp] = valE;
    end
    codeRet:begin
      registers[rsp] = valE;
    end
    codePushq:begin
      registers[rsp] = valE;
    end
    codePopq:begin
      registers[rsp] = valE;
      registers[rA] = valM;
    end
  endcase
  end

  endmodule
```

Execute

This is the stage where all the computation happens for an instruction. It has ALU. The execute stage typically involves activating one or more functional units, such as an arithmetic logic unit or a memory unit, to perform the necessary calculations or memory accesses.

It has two modules namely `execute` and `conditionalCodes`. The `execute` part does the computation of the instruction and `conditionalCodes` part sets up the conditional codes and conditional flag for the next instruction.

ALU - Arithmetic Logic Unit, deals with signed numbers with the following functions:

ADD - Adds two signed 64 bit numbers and returns a 64 bit output and an overflow bit.
SUB - Subtracts two signed 64 bit numbers and returns a 64 bit output and an overflow bit.
AND - ANDs two 64 bit signed numbers and returns a 64 bit output and 0 as overflow bit.
XOR - XORs two 64 bit signed numbers and returns a 64 bit output and 0 as overflow bit.

We made the ALU by keeping a modular design approach and creating different modules that perform the various operations. The interface between the ALU and the modules remains the same, but the implementation can be changed inside the modules and even the ALU itself, still keeping the functioning intact. The design methodology used in the various modules is mentioned below:

1) AND:

The AND operation was the simplest function to create a module out of. We take the two 64 bit numbers and use a for loop with genvar variables in order to simulate functioning of 64 AND gates which AND each individual bits of the two input registers `a` and `b` and push the output of each operation into corresponding bit of the `out` register.

2) XOR:

A very similar approach is used here as the AND operation. The two 64 bit numbers are taken and a for loop is used with genvar variables in order to simulate functioning of 64 XOR gates which perform the XOR operation of each individual bits of the two input registers `a` and `b` and push the output of each operation into corresponding bit of the `out` register.

3) ADD:

The basic idea about performing a 64 bit addition is cascading 64 Full Adders and using plugging the carryout bit of one full adder into the third input pin of the next full adder and thus we can perform the signed addition. On reaching the last and second last full adder, we can calculate if the output overflows based on the carry bits and the signed bits of the input lines.

4) SUB:

64 bit subtraction can be simulated by the adder module itself by taking 2’s complement of one of the input lines and then using the same implementation used in the adder module in order to add it to the other input line. Also, 2’s complement can be performed by taking NOT of every bit by using 64 not gates using genvar and then passing 1 as an input instead of zero in the first full adder while performing the addition. Again, overflow can be determined by looking at the signed bits of the two numbers and also the carry of the last full adder.

Code:

```
`timescale 1ns/10ps
`include "./ALU/ADD/Add64.v"
`include "./ALU/SUB/Sub64.v"
`include "./ALU/AND/AND64.v"
`include "./ALU/XOR/XOR64.v"
`include "./ALU/FullAdder.v"

// ================================

module ALU(sel,a,b,out,CondCod); //zf sf of
input [1:0] sel;
input signed [63:0] a, b;
output reg signed [63:0] out;
output reg [2:0] CondCod;
wire signed [63:0] outAdd,outSub,outAND,outXOR;
wire ovAdd,ovSub,ovADD,ovXOR;

Add64 ADDER      (a,b,outAdd,ovAdd);
Sub64 SUBTRACTER (a,b,outSub,ovSub);
AND64 ANDER      (a,b,outAND,ovAND);
XOR64 XORER      (a,b,outXOR,ovXOR);

initial begin
  CondCod[0] = 0;
  CondCod[1] = 0;
  CondCod[2] = 0;
end

always@(*)
begin
  case(sel)
    2'b00:begin
        out=outAdd;
        CondCod[0]=ovAdd;
      end
    2'b01:begin
        out=outSub;
        CondCod[0]=ovSub;
      end
```

```
    2'b10:begin
        out=outAND;
        CondCod[0]=ovAND;
      end
    2'b11:begin
        out=outXOR;
        CondCod[0]=ovXOR;
      end
  endcase

  if (out[63]==1'd1) begin
    CondCod[1] = 1;
  end
  else begin
    CondCod[1] = 0;
  end
  if (out == 64'd0) begin
    CondCod[2] = 1;
  end
  else begin
    CondCod[2] = 0;
  end
end

endmodule
```

---

back to Execute……

- `valE` is computed in this stage, depending on `icode` and `ifun` which define the type of instruction.
- Condition Codes(if any) are also set in this stage for the next instruction.
- It thus defines the input A `(ALU_a)` and input B `(ALU_b)` of the ALU, select line of the ALU `(sel)`

  and condition code `(CC)`.
- The execute stage is implemented as follows:
  - For Cmovxx,

    `ALU_a = valA`

    `ALU_b = 0`

    `Select line = 00 (add)`

    Checking of condition codes based on `ifun`

- For IRmovq,

  `ALU_a = valC`

  `ALU_b = 0`

  `Select line = 00 (add)`

- For RMmovq and MRmovq,

  `ALU_a = valC`

  `ALU_b = valB`

  `Select line = 00 (add)`

- For OPq,

  `ALU_a = valA`

  `ALU_b = valB`

  `Select line` will be selected based on `ifun`

- For Jxx,

  `ALU_a = valC`

  `ALU_b = 0`

  Checking of condition codes based on `ifun`

- For Call, Pushq,

  `ALU_a = valB`

  `ALU_b = 8`

  `Select line = 01 (sub)`

- For Ret, Popq,

  `ALU_a = valB`

  `ALU_b = 8`

  `Select line = 00 (sub)`

- ALU output is assigned to `valE` for each of the above cases.

Instruction		Synonym	Move condition	Description
`cmove`	S, R	`cmovz`	`ZF`	Equal / zero
`cmovne`	S, R	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	S, R		`SF`	Negative
`cmovns`	S, R		`~SF`	Nonnegative
`cmovg`	S, R	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	S, R	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	S, R	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	S, R	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	S, R	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	S, R	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	S, R	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	S, R	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

Condition Codes are set according to different logics according to the `ifun` of the instruction.(as shown in the above example of CmovXX)

Code:

```
`timescale 1ns/10ps

module conditionalCodes (icode,ifun, CC, Cnd);
  input [3:0] icode;
  input [3:0] ifun;
  input [2:0] CC;
  output Cnd;

  parameter BRANCH_UNC = 4'h0; // Unconditional jump
  parameter BRANCH_LE  = 4'h1; // Jump if less than or equal
  parameter BRANCH_L   = 4'h2; // Jump if less than
  parameter BRANCH_E   = 4'h3; // Jump if equal
  parameter BRANCH_NE  = 4'h4; // Jump if not equal
  parameter BRANCH_GE  = 4'h5; // Jump if greater than or equal
  parameter BRANCH_G   = 4'h6; // Jump if greater than

  wire ZF = CC[2];
  wire SF = CC[1];
  wire OF = CC[0];

  assign Cnd =
  (ifun == BRANCH_UNC) |
  (ifun == BRANCH_LE & ((SF ^ OF) | ZF)) |
```

```
  (ifun == BRANCH_L  & (SF ^ OF)) |
  (ifun == BRANCH_E  & ZF) |
  (ifun == BRANCH_NE & ~ZF) |
  (ifun == BRANCH_GE & ~(SF ^ OF)) |
  (ifun == BRANCH_G  & ~(SF ^ OF) & ~ZF) |
  (icode == 4'd6);
endmodule

module execute (clock, icode, ifun, valA, valB, valC, CCold, valE, CC, Cnd);

// inputs
input clock;
input [3:0 ] icode;
input [3:0 ] ifun;
input [63:0] valA;
input [63:0] valB;
input [63:0] valC;
// input [2:0] CC;

// for ALU
reg [1:0] sel;
reg signed [63:0] ALU_a,ALU_b;
wire [63:0] ALU_out;

// for CondCod
output Cnd;

// outputs
wire [2:0] CCtemp;
input [2:0] CCold;
output reg [2:0] CC;
output wire [63:0] valE;

// icodes
reg [3:0] codeHalt   = 4'b0000;// 0
reg [3:0] codeNop    = 4'b0001;// 1
reg [3:0] codeCmovXX = 4'b0010;// 2
reg [3:0] codeIRmovq = 4'b0011;// 3
reg [3:0] codeRMmovq = 4'b0100;// 4
reg [3:0] codeMRmovq = 4'b0101;// 5
reg [3:0] codeOPq    = 4'b0110;// 6
reg [3:0] codeJXX    = 4'b0111;// 7
reg [3:0] codeCall   = 4'b1000;// 8
reg [3:0] codeRet    = 4'b1001;// 9
reg [3:0] codePushq  = 4'b1010;// A
reg [3:0] codePopq   = 4'b1011;// B

initial begin
  // valE = 64'd0;
  CC = 3'b000;
end

always @(*) begin
case(icode)
```

```
codeHalt:begin
  // Nothing
end
codeNop:begin
  // Nothing
end
codeCmovXX:begin
  ALU_a = valA;
  ALU_b = 64'd0;
  sel = 2'b00; // add
  CC = CCold;
end
codeIRmovq:begin
  ALU_a = valC;
  ALU_b = 64'd0;
  sel = 2'b00; // add
  CC = CCtemp;
end
codeRMmovq:begin
  ALU_a = valC;
  ALU_b = valB;
  sel = 2'b00; // add
  CC = CCtemp;
end
codeMRmovq:begin
  ALU_a = valC;
  ALU_b = valB;
  sel = 2'b00; // add
  CC = CCtemp;
end
codeOPq:begin
  ALU_a = valA;
  ALU_b = valB;
  case (ifun)
  4'd0:begin // add
    sel = 2'd0;
  end
  4'd1:begin // sub
    sel = 2'd1;
  end
  4'd2:begin // and
    sel = 2'd2;
  end
  4'd3:begin // xor
    sel = 2'd3;
  end
  endcase
  CC = CCtemp;
end
codeJXX:begin
  // It will set Cnd
  CC = CCold;
end
codeCall:begin
```

```
    ALU_a = valB;
    ALU_b = 64'd8;
    sel = 2'b01; // SUB
    CC = CCtemp;
  end
  codeRet:begin
    ALU_a = valB;
    ALU_b = 64'd8;
    sel = 2'b00; // ADD
    CC = CCtemp;
  end
  codePushq:begin
    ALU_a = valB;
    ALU_b = 64'd8;
    sel = 2'b01; // SUB
    CC = CCtemp;
  end
  codePopq:begin
    ALU_a = valB;
    ALU_b = 64'd8;
    sel = 2'b00; // ADD
    CC = CCtemp;
  end
endcase
end

ALU UUT_alu (sel,ALU_a,ALU_b,ALU_out,CCtemp);
conditionalCodes UUT_cond2 (icode,ifun, CCold, Cnd);

assign valE = ALU_out;

endmodule
```

Memory

During the memory stage, the Y86 processor will first calculate the memory address to be accessed based on the effective address computed in the previous stage. It will then issue a read or write request to memory, depending on the operation specified by the instruction.

- If a write operation is performed, the data to be written will be taken from another register called the register file and written to the memory address.

- If a read operation is performed, the data from the designated operation is read into `valM`.
- Implementation of memory stage is as follows:
  - For MRmovq, Popq, we need to read from memory

    `valM = mem[valE]`
  - For ret, we need to read from memory

    `valM = mem[valA]`
  - For RMmovq, Pushq, we need to write to memory

    `mem[valE] = valA`
  - For Call, we need to write to memory

    `mem[valE] = valP`

Code:

```
`timescale 1ns/10ps

module memory(clock, icode, valA, valE, valP, valM);


// inputs
input clock;
input [3:0 ] icode;
input [63:0] valA;
input [63:0] valE;
input [63:0] valP;

// outputs
output reg [63:0] valM;

// icodes
reg [3:0] codeHalt   = 4'b0000;// 0
reg [3:0] codeNop    = 4'b0001;// 1
reg [3:0] codeCmovXX = 4'b0010;// 2
reg [3:0] codeIRmovq = 4'b0011;// 3
reg [3:0] codeRMmovq = 4'b0100;// 4
reg [3:0] codeMRmovq = 4'b0101;// 5
reg [3:0] codeOPq    = 4'b0110;// 6
reg [3:0] codeJXX    = 4'b0111;// 7
reg [3:0] codeCall   = 4'b1000;// 8
reg [3:0] codeRet    = 4'b1001;// 9
reg [3:0] codePushq  = 4'b1010;// A
reg [3:0] codePopq   = 4'b1011;// B
```

```
reg [63:0] mem [0:127];

initial begin
    mem[  0] = 64'h00;
    mem[  1] = 64'h00;
    mem[  2] = 64'h00;
    mem[  3] = 64'h00;
    mem[  4] = 64'h00;
    mem[  5] = 64'h0f;
    mem[  6] = 64'h00;
    mem[  7] = 64'h00;
    mem[  8] = 64'h00;
    mem[  9] = 64'h00;
    mem[ 10] = 64'h00;
    mem[ 11] = 64'h00;
    mem[ 12] = 64'h00;
    mem[ 13] = 64'h00;
    mem[ 14] = 64'h00;
    mem[ 15] = 64'h00;
    mem[ 16] = 64'h00;
    mem[ 17] = 64'h00;
    mem[ 18] = 64'h00;
    mem[ 19] = 64'h00;
end

always@(*)begin
  case(icode)
  codeHalt:begin
    // Nothing
  end
  codeNop:begin
    // Nothing
  end
  codeCmovXX:begin
    // Nothing
  end
  codeIRmovq:begin
    // Nothing
  end
  codeRMmovq:begin
    mem[valE] = valA;
  end
  codeMRmovq:begin
    valM = mem[valE];
  end
  codeOPq:begin
    // Nothing
  end
  codeJXX:begin
    // idk
  end
  codeCall:begin
    mem[valE] = valP;
  end
```

```
  codeRet:begin
    valM = mem[valA];
  end
  codePushq:begin
    mem[valE] = valA;
  end
  codePopq:begin
    valM = mem[valA];
  end
endcase
end

endmodule
```

Here is an example instruction memory, implemented with this sequential processor:

Instruction Memory:

```
initial begin
    insMem[  0] = 8'ha0;
    insMem[  1] = 8'haf;
    insMem[  2] = 8'h60;
    insMem[  3] = 8'h04;
    insMem[  4] = 8'h61;
    insMem[  5] = 8'hee;
    insMem[  6] = 8'h23;
    insMem[  7] = 8'h89;
    insMem[  8] = 8'h60;
    insMem[  9] = 8'h09;
    insMem[ 10] = 8'h20;
    insMem[ 11] = 8'h01;
    insMem[ 12] = 8'h60;
    insMem[ 13] = 8'h21;
    insMem[ 14] = 8'h80;
    insMem[ 15] = 8'h23;
    insMem[ 16] = 8'h00;
    insMem[ 17] = 8'h00;
    insMem[ 18] = 8'h00;
    insMem[ 19] = 8'h00;
    insMem[ 20] = 8'h00;
    insMem[ 21] = 8'h00;
    insMem[ 22] = 8'h00;
    insMem[ 23] = 8'h00;
    insMem[ 24] = 8'h00;
    insMem[ 25] = 8'h00;
    insMem[ 26] = 8'h00;
```

```
    insMem[ 27] = 8'h00;
    insMem[ 28] = 8'h00;
    insMem[ 29] = 8'h00;
    insMem[ 30] = 8'h00;
    insMem[ 31] = 8'h00;
    insMem[ 32] = 8'h00;
    insMem[ 33] = 8'h00;
    insMem[ 34] = 8'h00;
    insMem[ 35] = 8'h60;
    insMem[ 36] = 8'h04;
    insMem[ 37] = 8'h90;
    insMem[ 38] = 8'h00;
    insMem[ 39] = 8'h00;
end
```

Registers:

```
    registers[rax]=64'h0;
    registers[rcx]=64'h1;
    registers[rdx]=64'h2;
    registers[rbx]=64'h3;
    registers[rsp]=64'h15;
    registers[rbp]=64'h5;
    registers[rsi]=64'h6;
    registers[rdi]=64'h7;
    registers[r8 ]=64'h8;
    registers[r9 ]=64'h9;
    registers[r10]=64'ha;
    registers[r11]=64'hb;
    registers[r12]=64'hc;
    registers[r13]=64'hd;
    registers[r14]=64'he;
    registers[r15]=64'hf;
```

Result:

```
                  30
PC =                    0
icode = a
ifun = 0
rA = a
rB = f
valA =                   10
valB =                   21
valC = 0000000000000000
valE =                   13
 ZF = 0, SF = 0, OF = 0, Cnd=1

                  40
PC =                    2
icode = a
ifun = 0
rA = a
rB = f
valA =                   10
valB =                   13
valC = 0000000000000000
valE =                    5
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Explanation

In the positive edge of

```
                50
PC =                    2
icode = 6
ifun = 0
rA = 0
rB = 4
valA =                   0
valB =                  13
valC = 0000000000000000
valE =                  13
 ZF = 0, SF = 0, OF = 0, Cnd=1

                60
PC =                    4
icode = 6
ifun = 0
rA = 0
rB = 4
valA =                   0
valB =                  13
valC = 0000000000000000
valE =                  13
 ZF = 0, SF = 0, OF = 0, Cnd=1

                70
PC =                    4
icode = 6
ifun = 1
rA = e
rB = e
valA =                  14
valB =                  14
valC = 0000000000000000
valE =                   0
 ZF = 1, SF = 0, OF = 0, Cnd=1

                80
PC =                    6
icode = 6
ifun = 1
rA = e
rB = e
valA =                   0
valB =                   0
valC = 0000000000000000
valE =                   0
 ZF = 1, SF = 0, OF = 0, Cnd=1
```

Explanation

```
                  90
PC =                    6
icode = 2
ifun = 3
rA = 8
rB = 9
valA =                    8
valB =                    0
valC = 0000000000000000
valE =                    8
 ZF = 1, SF = 0, OF = 0, Cnd=1

                 100
PC =                    8
icode = 2
ifun = 3
rA = 8
rB = 9
valA =                    8
valB =                    0
valC = 0000000000000000
valE =                    8
 ZF = 1, SF = 0, OF = 0, Cnd=1

                 110
PC =                    8
icode = 6
ifun = 0
rA = 0
rB = 9
valA =                    0
valB =                    8
valC = 0000000000000000
valE =                    8
 ZF = 0, SF = 0, OF = 0, Cnd=1

                 120
PC =                   10
icode = 6
ifun = 0
rA = 0
rB = 9
valA =                    0
valB =                    8
valC = 0000000000000000
valE =                    8
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Explanation

```
                130
PC =                    10
icode = 2
ifun = 0
rA = 0
rB = 1
valA =                   0
valB =                   8
valC = 0000000000000000
valE =                   0
 ZF = 0, SF = 0, OF = 0, Cnd=1

                140
PC =                    12
icode = 2
ifun = 0
rA = 0
rB = 1
valA =                   0
valB =                   8
valC = 0000000000000000
valE =                   0
 ZF = 0, SF = 0, OF = 0, Cnd=1

                150
PC =                    12
icode = 6
ifun = 0
rA = 2
rB = 1
valA =                   2
valB =                   0
valC = 0000000000000000
valE =                   2
 ZF = 0, SF = 0, OF = 0, Cnd=1

                160
PC =                    14
icode = 6
ifun = 0
rA = 2
rB = 1
valA =                   2
valB =                   2
valC = 0000000000000000
valE =                   4
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Explanation

```
                170
PC =                  14
icode = 8
ifun = 0
rA = f
rB = f
valA =                 2
valB =                13
valC = 0000000000000023
valE =                 5
 ZF = 0, SF = 0, OF = 0, Cnd=1

                180
PC =                  35
icode = 8
ifun = 0
rA = f
rB = f
valA =                 2
valB =                 5
valC = 0000000000000023
valE =                -3
 ZF = 0, SF = 1, OF = 0, Cnd=1

                190
PC =                  35
icode = 6
ifun = 0
rA = 0
rB = 4
valA =                 0
valB =                 5
valC = 0000000000000000
valE =                 5
 ZF = 0, SF = 0, OF = 0, Cnd=1

                200
PC =                  37
icode = 6
ifun = 0
rA = 0
rB = 4
valA =                 0
valB =                 5
valC = 0000000000000000
valE =                 5
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Explanation

```
                210
PC =                    37
icode = 9
ifun = 0
rA = f
rB = f
valA =                   5
valB =                   5
valC = 0000000000000000
valE =                  13
 ZF = 0, SF = 0, OF = 0, Cnd=1

                220
PC =                    23
icode = 9
ifun = 0
rA = f
rB = f
valA =                  13
valB =                  13
valC = 0000000000000000
valE =                  21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                230
PC =                    23
icode = 0
ifun = 0
rA = f
rB = f
valA =                  13
valB =                  13
valC = 0000000000000000
valE =                  21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                240
PC =                    24
icode = 0
ifun = 0
rA = f
rB = f
valA =                  13
valB =                  13
valC = 0000000000000000
valE =                  21
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

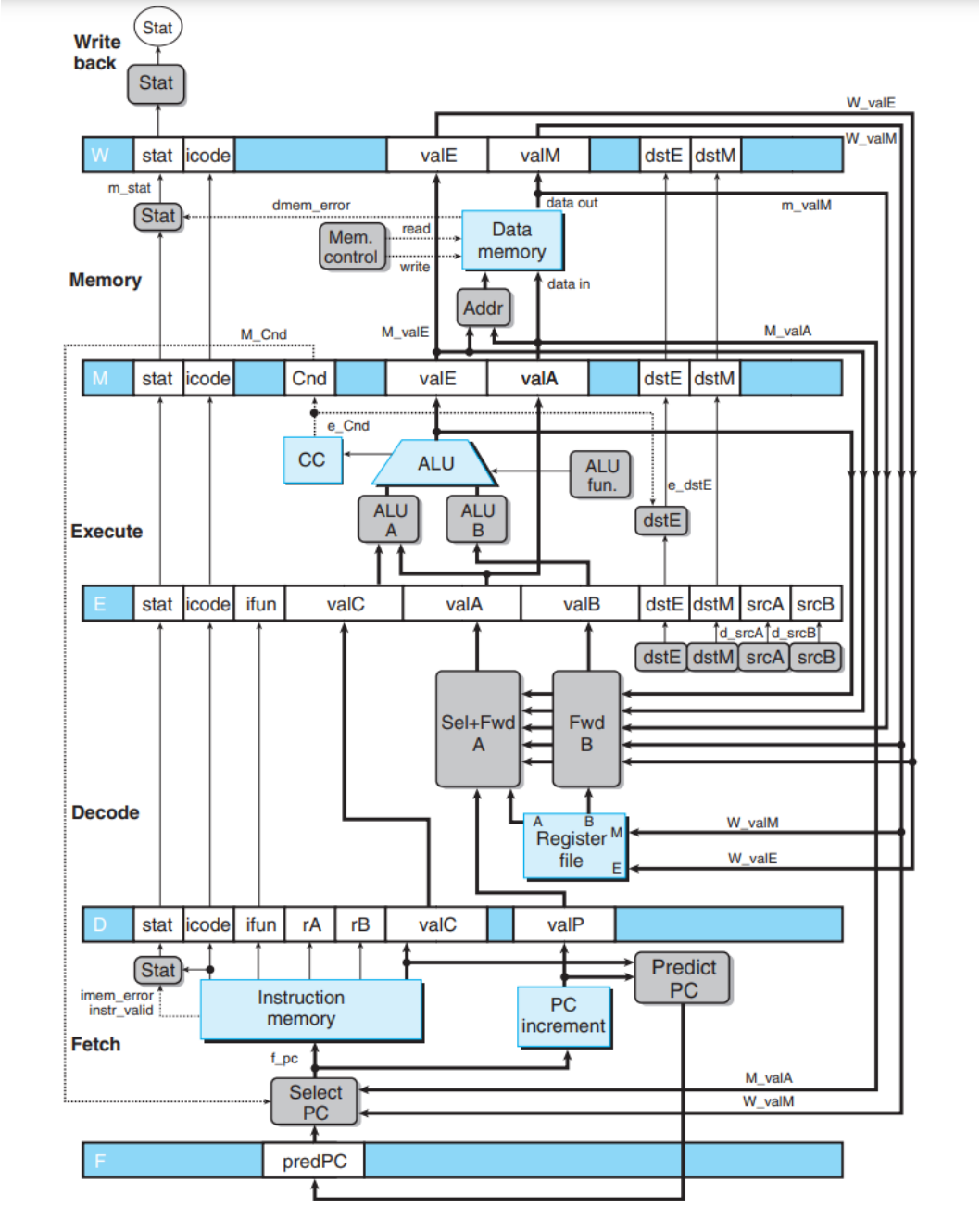
Explanation

```
                250
PC =                    24
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                260
PC =                    25
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                270
PC =                    25
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                280
PC =                    26
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Explanation

```
                290
PC =                    26
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1

                300
PC =                    27
icode = 0
ifun = 0
rA = f
rB = f
valA =                    13
valB =                    13
valC = 0000000000000000
valE =                    21
 ZF = 0, SF = 0, OF = 0, Cnd=1
```

Pipelined Processor Architecture



In a pipelined y86 processor, the processing of instructions is broken down into a series of stages that are executed concurrently. Each stage performs a different operation on the instruction being processed, such as fetching the instruction from memory, decoding it, executing it, and writing the result back to memory.

By dividing the processing of instructions into multiple stages that can be executed in parallel, a pipelined y86 processor can achieve a higher level of performance than a

non-pipelined processor. This is because the processor can begin executing the next instruction before it has completed processing the previous one, which reduces the amount of idle time and increases the overall throughput of the processor.

However, pipelining also introduces additional complexities, such as the need to handle data hazards and control hazards, which can affect the performance of the processor if not handled properly. Therefore, designing an efficient pipelined processor requires careful consideration of the trade-offs between performance and complexity.

PC Update to be changed

The final stage in the cycle of an instruction is the PC update stage in the SEQ implementation.

As we wish to be able to retrieve the next instruction continuously without having to wait for the PC update stage of the previous instruction, we should move the PC update stage to the beginning of the cycle for the pipelined implementation. Circuit retiming is the term for this. This modifies the circuit's overall state while having no impact on its local behavior. Additionally, it enables us to balance the delays in the pipelined system between stages.

Now, at the beginning of the cycle, the PC Update stage can keep providing updated PC to the fetch stage using required values from different stages.

PC Predict

PC prediction is required in pipelined y86 processors to improve performance by reducing the number of pipeline stalls caused by control hazards.

Control hazards occur when an instruction that changes the control flow of the program, such as a conditional branch or a jump, is in the pipeline. When this happens, the next instruction to be executed is not the next sequential instruction in memory, but rather the target of the branch or jump instruction. As a result, the pipeline needs to be flushed and restarted, causing a significant delay in the execution of the program.

PC prediction attempts to predict the target of the branch or jump instruction before it is actually executed, allowing the pipeline to continue executing instructions without stalling. If the prediction is correct, the pipeline continues to execute instructions without any delay. If the prediction is incorrect, the pipeline needs to be flushed and restarted,

but the delay is minimized because the flushing only affects the instructions after the branch or jump instruction.

Code:

```
`timescale 1ns/10ps
module PCPredict (clock, F_control, f_icode, f_valC, f_valP, PCpred);
  // inputs
  input clock;
  input [ 3:0] f_icode;
  input [63:0] f_valC;
  input [63:0] f_valP;
  input [63:0] F_control;
  // outputs
  output reg [63:0] PCpred;

  // icodes
  reg [3:0] codeHalt   = 4'b0000;// 0
  reg [3:0] codeNop    = 4'b0001;// 1
  reg [3:0] codeCmovXX = 4'b0010;// 2
  reg [3:0] codeIRmovq = 4'b0011;// 3
  reg [3:0] codeRMmovq = 4'b0100;// 4
  reg [3:0] codeMRmovq = 4'b0101;// 5
  reg [3:0] codeOPq    = 4'b0110;// 6
  reg [3:0] codeJXX    = 4'b0111;// 7
  reg [3:0] codeCall   = 4'b1000;// 8
  reg [3:0] codeRet    = 4'b1001;// 9
  reg [3:0] codePushq  = 4'b1010;// A
  reg [3:0] codePopq   = 4'b1011;// B

  always @(*)begin
    if (clock == 0) begin
      PCpred = 0;end
  end

  always@(negedge clock)begin
    if (F_control == 0) begin
      if (f_icode == codeCall) begin
        PCpred = f_valC; end
      else if (f_icode == codeJXX) begin
        PCpred = f_valC; end
      else begin
        PCpred = f_valP; end
    end
    else begin // F is stalled
      case(f_icode)
      codeHalt:begin
        PCpred = f_valP - 1;
      end
      codeNop:begin
        PCpred = f_valP - 1;
```

```
      end
      codeCmovXX:begin
        PCpred = f_valP - 2;
      end
      codeIRmovq:begin
        PCpred = f_valP - 10;
      end
      codeRMmovq:begin
        PCpred = f_valP - 10;
      end
      codeMRmovq:begin
        PCpred = f_valP - 10;
      end
      codeOPq:begin
        PCpred = f_valP - 2;
      end
      codeJXX:begin
        PCpred = f_valP - 9;
      end
      codeCall:begin
        PCpred = f_valP - 9;
      end
      codeRet:begin
        PCpred = f_valP - 1;
      end
      codePushq:begin
        PCpred = f_valP - 2;
      end
      codePopq:begin
        PCpred = f_valP - 2;
      end

    endcase
    end
  end
endmodule
```

- PCpred takes the value of `f_valC` when `f_icode` is `codeCall` or `codeJXX`. Else it takes the value of `f_valP`.
- Here, the thing to note is that if F is stalled, we have to reassign the PCpred so that when fetch runs again it grabs the previous PC value.

Inserting pipeline registers

A pipelined computer usually has "pipeline registers" after each stage. These store information from the instruction and calculations so that the logic gates of the next stage can do the next step

They are used for several reasons:

1. To break up the pipeline stages: A pipeline register is used to separate each pipeline stage, which allows the pipeline to break up the work into smaller, more manageable pieces. This makes it easier to design and implement a pipeline, as each stage can be optimized individually.
2. To hold intermediate results: Each pipeline stage performs a specific task on the data, and the results of that task are stored in the pipeline register until they are needed by the next stage. This helps to prevent data dependencies and allows each stage to work independently of the others.
3. To synchronize the pipeline: The pipeline registers provide a way to synchronize the pipeline stages. Each stage can only work on the data in the pipeline register when it is available, and it can only pass the results to the next stage when the next pipeline register is ready to receive it. This ensures that the pipeline stages work together in a coordinated way.
4. To enable forwarding: Pipeline registers can also be used to enable forwarding, which allows the processor to bypass the pipeline registers and send data directly from one stage to another. This helps to improve performance by reducing the number of cycles needed to complete an instruction.

Overall, pipeline registers play a critical role in the functioning of a pipelined y86 processor by enabling efficient and synchronized processing of instructions.

Data Forwarding

In pipelined y86, data-forwarding refers to the technique used to resolve data hazards by forwarding the necessary data to an instruction that needs it, rather than stalling the pipeline.

Data hazards occur in pipelined y86 when an instruction in the pipeline requires data that is not yet available because it has not been produced by a previous instruction in the pipeline. In a non-pipelined processor, the solution would be to stall the pipeline until the data becomes available. However, this reduces the performance of the processor.

To avoid stalling, pipelined y86 uses data-forwarding to supply the required data to an instruction that needs it. Data-forwarding is possible because the y86 processor has

multiple data paths that allow data to be routed around the pipeline without affecting the instructions currently in the pipeline.

There are two types of data-forwarding used in y86 pipelining:

1. Forwarding data from the execute stage to the memory stage: This occurs when an instruction in the memory stage requires data that is being calculated in the execute stage of the pipeline. The data is forwarded directly from the execute stage to the memory stage, allowing the memory stage instruction to continue executing without waiting for the data.
2. Forwarding data from the memory stage to the execute stage: This occurs when an instruction in the execute stage requires data that was written to memory by a previous instruction in the pipeline. The data is forwarded directly from the memory stage to the execute stage, allowing the execute stage instruction to continue executing without waiting for the data.

By using data-forwarding, pipelined y86 can continue executing instructions without stalling, which results in a faster and more efficient processor.

Code to handle data forwarding: (present in decode_writeback)

```
always @(*) begin
    // for rA
    if (D_icode == codeCall | D_icode == codeJXX)begin
      d_valA = D_valP;end // Use incremented PC for fallback
    else if (D_rA == E_dstE)begin
      d_valA = e_valE;end // forward valE from execute
    else if (D_rA == M_dstM)begin
      d_valA = m_valM;end // forward valM from memory
    else if (D_rA == M_dstE)begin
      d_valA = M_valE;end // forward valE from memory
    else if (D_rA == W_dstM)begin
      d_valA = W_valM;end // forward valM from writeback
    else if (D_rA == W_dstE)begin
      d_valA = W_valE;end // forward valE from writeback
    else begin
      d_valA = registers[D_rA];end
    // for rB
    if (D_rB == E_dstE)begin
      d_valB = e_valE;end
    else if (D_rB == M_dstM)begin
      d_valB = m_valM;end
    else if (D_rB == M_dstE)begin
```

```
      d_valB = M_valE;end
    else if (D_rB == W_dstM)begin
      d_valB = W_valM;end
    else if (D_rB == W_dstE)begin
      d_valB = W_valE;end
    else begin
      d_valB = registers[D_rB];end
  end
```

Handling Exceptions:

Load-Use Hazard:

```
initial begin
    insMem[  0] = 8'h50;
    insMem[  1] = 8'h10;
    insMem[  2] = 8'h0e;
    insMem[  3] = 8'h00;
    insMem[  4] = 8'h00;
    insMem[  5] = 8'h00;
    insMem[  6] = 8'h00;
    insMem[  7] = 8'h00;
    insMem[  8] = 8'h00;
    insMem[  9] = 8'h00;
    insMem[ 10] = 8'h60;
    insMem[ 11] = 8'h11;
    insMem[ 12] = 8'h60;
    insMem[ 13] = 8'h22;
    insMem[ 14] = 8'h60;
    insMem[ 15] = 8'h11;
    insMem[ 16] = 8'h00;
    insMem[ 17] = 8'h00;
    insMem[ 18] = 8'h00;
    insMem[ 19] = 8'h00;
    insMem[ 20] = 8'h00;
    insMem[ 21] = 8'h00;
    insMem[ 22] = 8'h00;
    insMem[ 23] = 8'h00;
    insMem[ 24] = 8'h00;
    insMem[ 25] = 8'h00;
    insMem[ 26] = 8'h00;
    insMem[ 27] = 8'h00;
    insMem[ 28] = 8'h00;
    insMem[ 29] = 8'h00;
end
```

Used instruction memory that produces load use hazard. Following is the output obtained for the following. As soon as the MRmovq operation reaches execute stage, it bubbles the next execute and stalls fetch and decode so that the decode of the next operation reads the updated value of the register from m_valM as now memory stage will lie above the decode of next operation.
Output:

```
10
FETCH: PCpred=x
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0
DECODE:
D_icode= x, D_ifun= x, D_rA= x, D_rB= x, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  20
FETCH: PCpred=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, d_valA=15, d_valB=15
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  30
FETCH: PCpred=0
f_icode= 5, f_ifun= 0, f_rA= 1, f_rB= 0, f_valC=14
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, d_valA=15, d_valB=15
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  40
FETCH: PCpred=10
f_icode= 5, f_ifun= 0, f_rA= 1, f_rB= 0, f_valC=14
DECODE:
D_icode= 5, D_ifun= 0, D_rA= 1, D_rB= 0, d_valA=1, d_valB=0
EXECUTE:
```

```
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                50
FETCH: PCpred=10
f_icode= 6, f_ifun= 0, f_rA= 1, f_rB= 1, f_valC=0
DECODE:
D_icode= 5, D_ifun= 0, D_rA= 1, D_rB= 0, d_valA=1, d_valB=0
EXECUTE:
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                60
FETCH: PCpred=12
f_icode= 6, f_ifun= 0, f_rA= 1, f_rB= 1, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=1, d_valB=1
EXECUTE:
E_icode= 5, E_ifun= 0, E_valA=1, E_valB=0, E_valC=14, E_dstE=15, E_dstM= 1, e_valE=x
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                70
FETCH: PCpred=12
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=1, d_valB=1
EXECUTE:
E_icode= 5, E_ifun= 0, E_valA=1, E_valB=0, E_valC=14, E_dstE=15, E_dstM= 1, e_valE=14
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                80
FETCH: PCpred=12
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=x, d_valB=x
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=15, E_dstM=15, e_valE=14
MEMORY:
M_icode= 5, M_valA=1, M_valE=14, M_dstE=15, M_dstM= 1, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x
```

```
                 90
FETCH: PCpred=12
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=18, d_valB=18
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=15, E_dstM=15, e_valE=14
MEMORY:
M_icode= 5, M_valA=1, M_valE=14, M_dstE=15, M_dstM= 1, M_Cnd=x, m_valM=18
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                100
FETCH: PCpred=14
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=18, E_valB=18, E_valC=0, E_dstE= 1, E_dstM=15, e_valE=14
MEMORY:
M_icode= 1, M_valA=0, M_valE=14, M_dstE=15, M_dstM=15, M_Cnd=x, m_valM=18
WRITEBACK:
W_icode= 5, W_dstE=15, W_dstM= 1, W_valE=14, W_valM=18

                110
FETCH: PCpred=14
f_icode= 6, f_ifun= 0, f_rA= 1, f_rB= 1, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=18, E_valB=18, E_valC=0, E_dstE= 1, E_dstM=15, e_valE=36
MEMORY:
M_icode= 1, M_valA=0, M_valE=14, M_dstE=15, M_dstM=15, M_Cnd=x, m_valM=18
WRITEBACK:
W_icode= 5, W_dstE=15, W_dstM= 1, W_valE=14, W_valM=18

                120
FETCH: PCpred=16
f_icode= 6, f_ifun= 0, f_rA= 1, f_rB= 1, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=36, d_valB=36
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=36
MEMORY:
M_icode= 6, M_valA=18, M_valE=36, M_dstE= 1, M_dstM=15, M_Cnd=x, m_valM=18
WRITEBACK:
W_icode= 1, W_dstE=15, W_dstM=15, W_valE=14, W_valM=18

                130
FETCH: PCpred=16
f_icode= 0, f_ifun= 0, f_rA=15, f_rB=15, f_valC=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 1, D_rB= 1, d_valA=36, d_valB=36
EXECUTE:
```

```
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4
MEMORY:
M_icode= 6, M_valA=18, M_valE=36, M_dstE= 1, M_dstM=15, M_Cnd=x, m_valM=18
WRITEBACK:
W_icode= 1, W_dstE=15, W_dstM=15, W_valE=14, W_valM=18
```

As we can see, when we encounter the MRmovq in execute stage at time #70, the execute stage is bubbled, and fetch as well as decode stages are put into stall mode. As it can be seen at time #90, which is the next posedge of the clock, the PC for fetch remains the same as fetch module is in stall mode.

Handling mispredicted branch:

```
initial begin
    insMem[  0] = 8'h60;
    insMem[  1] = 8'h22;
    insMem[  2] = 8'h71;
    insMem[  3] = 8'h10;
    insMem[  4] = 8'h00;
    insMem[  5] = 8'h00;
    insMem[  6] = 8'h00;
    insMem[  7] = 8'h00;
    insMem[  8] = 8'h00;
    insMem[  9] = 8'h00;
    insMem[ 10] = 8'h00;
    insMem[ 11] = 8'h61;
    insMem[ 12] = 8'h32;
    insMem[ 13] = 8'h60;
    insMem[ 14] = 8'h99;
    insMem[ 15] = 8'h00;
    insMem[ 16] = 8'h60;
    insMem[ 17] = 8'h88;
    insMem[ 18] = 8'h60;
    insMem[ 19] = 8'haa;
    insMem[ 20] = 8'h00;
    insMem[ 21] = 8'h00;
    insMem[ 22] = 8'h00;
    insMem[ 23] = 8'h00;
    insMem[ 24] = 8'h00;
    insMem[ 25] = 8'h00;
    insMem[ 26] = 8'h00;
    insMem[ 27] = 8'h00;
    insMem[ 28] = 8'h00;
    insMem[ 29] = 8'h00;
end
```

Output:

```
0
FETCH: PCpred=x, PC=x
f_icode= x, f_ifun= x, f_rA= x, f_rB= x, f_valC=x, f_valP=x
DECODE:
D_icode= x, D_ifun= x, D_rA= x, D_rB= x, D_valP=x, d_valA=x, d_valB=0
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  10
FETCH: PCpred=x, PC=x
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA= x, D_rB= x, D_valP=x, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  20
FETCH: PCpred=0, PC=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  30
FETCH: PCpred=0, PC=0
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=15, d_valB=15
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
```

```
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  40
FETCH: PCpred=2, PC=2
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=15, d_valB=15
EXECUTE:
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x, e_C
nd=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  50
FETCH: PCpred=2, PC=2
f_icode= 7, f_ifun= 1, f_rA=15, f_rB=15, f_valC=16, f_valP=11
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=2, d_valB=2
EXECUTE:
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x, e_C
nd=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  60
FETCH: PCpred=16, PC=16
f_icode= 7, f_ifun= 1, f_rA=15, f_rB=15, f_valC=16, f_valP=11
DECODE:
D_icode= 7, D_ifun= 1, D_rA=15, D_rB=15, D_valP=11, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=x, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  70
FETCH: PCpred=16, PC=16
f_icode= 6, f_ifun= 0, f_rA= 8, f_rB= 8, f_valC=0, f_valP=18
DECODE:
D_icode= 7, D_ifun= 1, D_rA=15, D_rB=15, D_valP=11, d_valA=11, d_valB=15
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x
```

```
                  80
FETCH: PCpred=18, PC=18
f_icode= 6, f_ifun= 0, f_rA= 8, f_rB= 8, f_valC=0, f_valP=18
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 8, D_rB= 8, D_valP=18, d_valA=11, d_valB=15
EXECUTE:
E_icode= 7, E_ifun= 1, E_valA=11, E_valB=15, E_valC=16, E_dstE= 2, E_dstM=15, e_valE=4, e_
Cnd=0
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  90
FETCH: PCpred=18, PC=18
f_icode= 6, f_ifun= 0, f_rA=10, f_rB=10, f_valC=0, f_valP=20
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 8, D_rB= 8, D_valP=18, d_valA=8, d_valB=8
EXECUTE:
E_icode= 7, E_ifun= 1, E_valA=11, E_valB=15, E_valC=16, E_dstE= 2, E_dstM=15, e_valE=4, e_
Cnd=0
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                 100
FETCH: PCpred=20, PC=11
f_icode= 6, f_ifun= 0, f_rA=10, f_rB=10, f_valC=0, f_valP=20
DECODE:
D_icode= 1, D_ifun= 0, D_rA=15, D_rB=15, D_valP=0, d_valA=8, d_valB=8
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=15, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= 7, M_valA=11, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                 110
FETCH: PCpred=20, PC=11
f_icode= 6, f_ifun= 1, f_rA= 3, f_rB= 2, f_valC=0, f_valP=13
DECODE:
D_icode= 1, D_ifun= 0, D_rA=15, D_rB=15, D_valP=0, d_valA=4, d_valB=4
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=15, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= 7, M_valA=11, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                 120
FETCH: PCpred=13, PC=13
```

```
f_icode= 6, f_ifun= 1, f_rA= 3, f_rB= 2, f_valC=0, f_valP=13
DECODE:
D_icode= 6, D_ifun= 1, D_rA= 3, D_rB= 2, D_valP=13, d_valA=4, d_valB=4
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=4, E_valB=4, E_valC=0, E_dstE= 8, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= 1, M_valA=0, M_valE=4, M_dstE=15, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 7, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                130
FETCH: PCpred=13, PC=13
f_icode= 6, f_ifun= 0, f_rA= 9, f_rB= 9, f_valC=0, f_valP=15
DECODE:
D_icode= 6, D_ifun= 1, D_rA= 3, D_rB= 2, D_valP=13, d_valA=3, d_valB=2
EXECUTE:
E_icode= 1, E_ifun= 0, E_valA=4, E_valB=4, E_valC=0, E_dstE= 8, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= 1, M_valA=0, M_valE=4, M_dstE=15, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 7, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                140
FETCH: PCpred=15, PC=15
f_icode= 6, f_ifun= 0, f_rA= 9, f_rB= 9, f_valC=0, f_valP=15
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 9, D_rB= 9, D_valP=15, d_valA=3, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=3, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= 1, M_valA=4, M_valE=4, M_dstE= 8, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 1, W_dstE=15, W_dstM=15, W_valE=4, W_valM=x

                150
FETCH: PCpred=15, PC=15
f_icode= 0, f_ifun= 0, f_rA=15, f_rB=15, f_valC=0, f_valP=16
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 9, D_rB= 9, D_valP=15, d_valA=9, d_valB=9
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=3, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=1, e_Cnd
=1
MEMORY:
M_icode= 1, M_valA=4, M_valE=4, M_dstE= 8, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 1, W_dstE=15, W_dstM=15, W_valE=4, W_valM=x

                160
FETCH: PCpred=16, PC=16
f_icode= 0, f_ifun= 0, f_rA=15, f_rB=15, f_valC=0, f_valP=16
DECODE:
```

```
D_icode= 0, D_ifun= 0, D_rA=15, D_rB=15, D_valP=16, d_valA=9, d_valB=9
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=9, E_valB=9, E_valC=0, E_dstE= 9, E_dstM=15, e_valE=1, e_Cnd
=1
MEMORY:
M_icode= 6, M_valA=3, M_valE=1, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 1, W_dstE= 8, W_dstM=15, W_valE=4, W_valM=x
```

On time step 90 the processor executes the jump instruction, and on evaluating Cnd, it realises the the prediction of jump was wrong, and now has to revert back to the correct PC, which it obtains from M_valA in next step (110). Also, decode is bubbled so that the mispredicted instructions do not cause problems and unnecessary updates in memory and writeback.

A Sample Instruction Memory

```
initial begin
    insMem[  0] = 8'h60; //add
    insMem[  1] = 8'h22;
    insMem[  2] = 8'h80; //call
    insMem[  3] = 8'h0e; //dest = insMem[14]
    insMem[  4] = 8'h00;
    insMem[  5] = 8'h00;
    insMem[  6] = 8'h00;
    insMem[  7] = 8'h00;
    insMem[  8] = 8'h00;
    insMem[  9] = 8'h00;
    insMem[ 10] = 8'h00;
    insMem[ 11] = 8'h61;
    insMem[ 12] = 8'h32;
    insMem[ 13] = 8'h00;
    insMem[ 14] = 8'h30; // IRMOVQ
    insMem[ 15] = 8'hf9;
    insMem[ 16] = 8'h45;
    insMem[ 17] = 8'h00;
    insMem[ 18] = 8'h00;
    insMem[ 19] = 8'h00;
    insMem[ 20] = 8'h00;
    insMem[ 21] = 8'h00;
    insMem[ 22] = 8'h00;
    insMem[ 23] = 8'h00;
    insMem[ 24] = 8'h61; // Sub
    insMem[ 25] = 8'hee;
    insMem[ 26] = 8'h21; // Cmove
    insMem[ 27] = 8'hbd;
    insMem[ 28] = 8'h60; //Add to check Cmove
```

```
    insMem[ 29] = 8'hdd;
end
```

Output:

```
0
FETCH: PCpred=x, PC=x
f_icode= x, f_ifun= x, f_rA= x, f_rB= x, f_valC=x, f_valP=x
DECODE:
D_icode= x, D_ifun= x, D_rA= x, D_rB= x, D_valP=x, d_valA=x, d_valB=0
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  10
FETCH: PCpred=x, PC=x
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA= x, D_rB= x, D_valP=x, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  20
FETCH: PCpred=0, PC=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  30
FETCH: PCpred=0, PC=0
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=15, d_valB=15
EXECUTE:
```

```
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=x, E_dstE= x, E_dstM= x, e_valE=x, e_Cnd
=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  40
FETCH: PCpred=2, PC=2
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=15, d_valB=15
EXECUTE:
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x, e_C
nd=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  50
FETCH: PCpred=2, PC=2
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=2, d_valB=2
EXECUTE:
E_icode= x, E_ifun= x, E_valA=15, E_valB=15, E_valC=0, E_dstE= x, E_dstM= x, e_valE=x, e_C
nd=x
MEMORY:
M_icode= x, M_valA=x, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  60
FETCH: PCpred=14, PC=14
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=x, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  70
FETCH: PCpred=14, PC=14
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=11, d_valB=15
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4, e_Cnd
=1
```

```
MEMORY:
M_icode= x, M_valA=15, M_valE=x, M_dstE= x, M_dstM= x, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  80
FETCH: PCpred=24, PC=24
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=11, d_valB=15
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=15, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=4, e_
Cnd=1
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                  90
FETCH: PCpred=24, PC=24
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=x, d_valB=9
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=15, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=7, e_
Cnd=1
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE= x, W_dstM= x, W_valE=x, W_valM=x

                 100
FETCH: PCpred=26, PC=26
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=x, d_valB=9
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=7, e_Cn
d=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=7, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                 110
FETCH: PCpred=26, PC=26
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=14, d_valB=14
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=69, e_C
nd=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=7, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
```

```
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                120
FETCH: PCpred=28, PC=28
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=14, d_valB=14
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=69, e_
Cnd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=7, W_valM=x

                130
FETCH: PCpred=28, PC=28
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=11, d_valB=13
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=0, e_C
nd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=7, W_valM=x

                140
FETCH: PCpred=30, PC=30
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=11, d_valB=13
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_C
nd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x

                150
FETCH: PCpred=30, PC=30
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=0, d_valB=0
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_
Cnd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x
```

```
                160
FETCH: PCpred=0, PC=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=0, d_valB=0
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_Cn
d=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x

                170
FETCH: PCpred=0, PC=0
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=x, d_valB=x
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x

                180
FETCH: PCpred=2, PC=2
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=x
MEMORY:
M_icode= 6, M_valA=0, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 2, W_dstE=13, W_dstM=15, W_valE=11, W_valM=x

                190
FETCH: PCpred=2, PC=2
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=2, d_valB=2
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=x
MEMORY:
M_icode= 6, M_valA=0, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 2, W_dstE=13, W_dstM=15, W_valE=11, W_valM=x

                200
```

```
FETCH: PCpred=14, PC=14
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=0, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=x, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                210
FETCH: PCpred=14, PC=14
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=11, d_valB=x
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=x, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                220
FETCH: PCpred=24, PC=24
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=11, d_valB=x
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=x, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=4, e_C
nd=1
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                230
FETCH: PCpred=24, PC=24
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=x, d_valB=9
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=x, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=x, e_C
nd=1
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                240
FETCH: PCpred=26, PC=26
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
```

```
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=x, d_valB=9
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=x, e_Cn
d=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=x, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                250
FETCH: PCpred=26, PC=26
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=14, d_valB=14
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=69, e_C
nd=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=x, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                260
FETCH: PCpred=28, PC=28
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=14, d_valB=14
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=69, e_
Cnd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=x, W_valM=x

                270
FETCH: PCpred=28, PC=28
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=11, d_valB=13
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=0, e_C
nd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=x, W_valM=x

                280
FETCH: PCpred=30, PC=30
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=11, d_valB=13
```

```
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_C
nd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x

                290
FETCH: PCpred=30, PC=30
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=0, d_valB=0
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_
Cnd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x

                300
FETCH: PCpred=0, PC=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=0, d_valB=0
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_Cn
d=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x

                310
FETCH: PCpred=0, PC=0
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=x, d_valB=x
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x

                320
FETCH: PCpred=2, PC=2
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
```

```
=x
MEMORY:
M_icode= 6, M_valA=0, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 2, W_dstE=13, W_dstM=15, W_valE=11, W_valM=x

                330
FETCH: PCpred=2, PC=2
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=2, d_valB=2
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=x
MEMORY:
M_icode= 6, M_valA=0, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 2, W_dstE=13, W_dstM=15, W_valE=11, W_valM=x

                340
FETCH: PCpred=14, PC=14
f_icode= 8, f_ifun= 0, f_rA=15, f_rB=15, f_valC=14, f_valP=11
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=2, d_valB=2
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=0, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=x, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                350
FETCH: PCpred=14, PC=14
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 8, D_ifun= 0, D_rA=15, D_rB=15, D_valP=11, d_valA=11, d_valB=x
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=2, E_valB=2, E_valC=0, E_dstE= 2, E_dstM=15, e_valE=4, e_Cnd
=1
MEMORY:
M_icode= x, M_valA=x, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=x, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                360
FETCH: PCpred=24, PC=24
f_icode= 3, f_ifun= 0, f_rA=15, f_rB= 9, f_valC=69, f_valP=24
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=11, d_valB=x
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=x, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=4, e_C
nd=1
MEMORY:
```

```
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                370
FETCH: PCpred=24, PC=24
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
DECODE:
D_icode= 3, D_ifun= 0, D_rA=15, D_rB= 9, D_valP=24, d_valA=x, d_valB=9
EXECUTE:
E_icode= 8, E_ifun= 0, E_valA=11, E_valB=x, E_valC=14, E_dstE= 4, E_dstM=15, e_valE=x, e_C
nd=1
MEMORY:
M_icode= 6, M_valA=2, M_valE=4, M_dstE= 2, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= x, W_dstE=13, W_dstM=15, W_valE=0, W_valM=x

                380
FETCH: PCpred=26, PC=26
f_icode= 6, f_ifun= 1, f_rA=14, f_rB=14, f_valC=0, f_valP=26
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=x, d_valB=9
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=x, e_Cn
d=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=x, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                390
FETCH: PCpred=26, PC=26
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 6, D_ifun= 1, D_rA=14, D_rB=14, D_valP=26, d_valA=14, d_valB=14
EXECUTE:
E_icode= 3, E_ifun= 0, E_valA=x, E_valB=9, E_valC=69, E_dstE= 9, E_dstM=15, e_valE=69, e_C
nd=1
MEMORY:
M_icode= 8, M_valA=11, M_valE=x, M_dstE= 4, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE= 2, W_dstM=15, W_valE=4, W_valM=x

                400
FETCH: PCpred=28, PC=28
f_icode= 2, f_ifun= 1, f_rA=11, f_rB=13, f_valC=0, f_valP=28
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=14, d_valB=14
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=69, e_
Cnd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
```

```
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=x, W_valM=x

                410
FETCH: PCpred=28, PC=28
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 2, D_ifun= 1, D_rA=11, D_rB=13, D_valP=28, d_valA=11, d_valB=13
EXECUTE:
E_icode= 6, E_ifun= 1, E_valA=14, E_valB=14, E_valC=0, E_dstE=14, E_dstM=15, e_valE=0, e_C
nd=1
MEMORY:
M_icode= 3, M_valA=x, M_valE=69, M_dstE= 9, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 8, W_dstE= 4, W_dstM=15, W_valE=x, W_valM=x

                420
FETCH: PCpred=30, PC=30
f_icode= 6, f_ifun= 0, f_rA=13, f_rB=13, f_valC=0, f_valP=30
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=11, d_valB=13
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_C
nd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x

                430
FETCH: PCpred=30, PC=30
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= 6, D_ifun= 0, D_rA=13, D_rB=13, D_valP=30, d_valA=0, d_valB=0
EXECUTE:
E_icode= 2, E_ifun= 1, E_valA=11, E_valB=13, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_
Cnd=0
MEMORY:
M_icode= 6, M_valA=14, M_valE=0, M_dstE=14, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 3, W_dstE= 9, W_dstM=15, W_valE=69, W_valM=x

                440
FETCH: PCpred=0, PC=0
f_icode= x, f_ifun= x, f_rA=15, f_rB=15, f_valC=0, f_valP=0
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=0, d_valB=0
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=11, e_Cn
d=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x
```

```
                450
FETCH: PCpred=0, PC=0
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= x, D_ifun= x, D_rA=15, D_rB=15, D_valP=0, d_valA=x, d_valB=x
EXECUTE:
E_icode= 6, E_ifun= 0, E_valA=0, E_valB=0, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=1
MEMORY:
M_icode= 2, M_valA=11, M_valE=11, M_dstE=13, M_dstM=15, M_Cnd=0, m_valM=x
WRITEBACK:
W_icode= 6, W_dstE=14, W_dstM=15, W_valE=0, W_valM=x

                460
FETCH: PCpred=2, PC=2
f_icode= 6, f_ifun= 0, f_rA= 2, f_rB= 2, f_valC=0, f_valP=2
DECODE:
D_icode= 6, D_ifun= 0, D_rA= 2, D_rB= 2, D_valP=2, d_valA=x, d_valB=x
EXECUTE:
E_icode= x, E_ifun= x, E_valA=x, E_valB=x, E_valC=0, E_dstE=13, E_dstM=15, e_valE=0, e_Cnd
=x
MEMORY:
M_icode= 6, M_valA=0, M_valE=0, M_dstE=13, M_dstM=15, M_Cnd=1, m_valM=x
WRITEBACK:
W_icode= 2, W_dstE=13, W_dstM=15, W_valE=11, W_valM=x
```