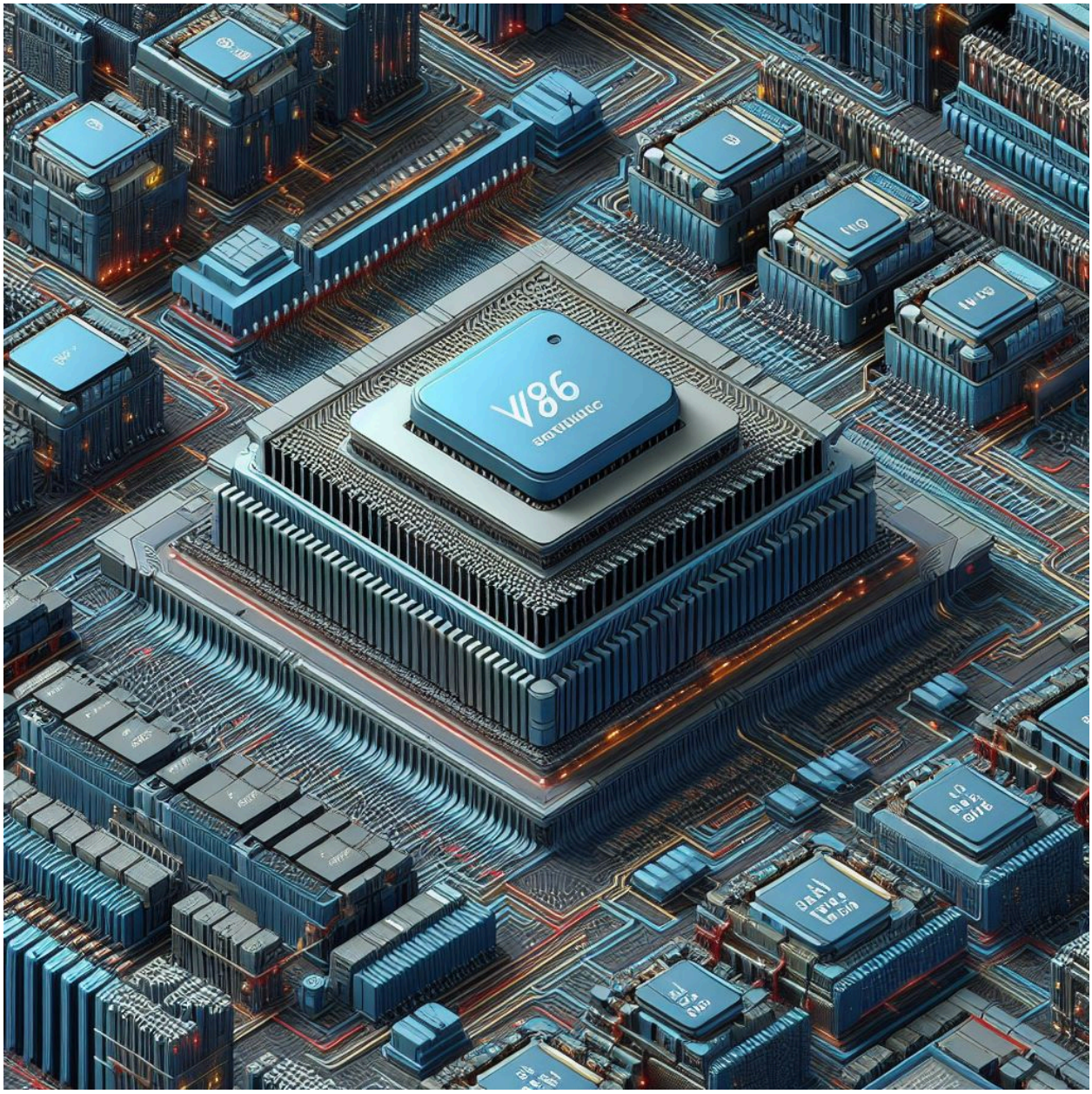
INTRODUCTION TO PROCESSOR ARCHITECTURE

Project Report (Spring 2024)

Verilog Implementation of Y86 Sequential and Pipelined Architectures



Introduction

In this project, we delve into the fascinating world of processor design using the VERILOG hardware description language. Our primary focus is on the Y86 64-bit processor architecture, and we explore both sequential and pipelined implementations.

- Sequential Implementation:

  We start by building the Y86 processor in a straightforward, step-by-step manner. This phase includes essential components like Fetch logic, Decode logic, Execute logic, Memory Logic, the Writeback stage and finally, the PC Update stage, which we'll later omit in the pipelined version.

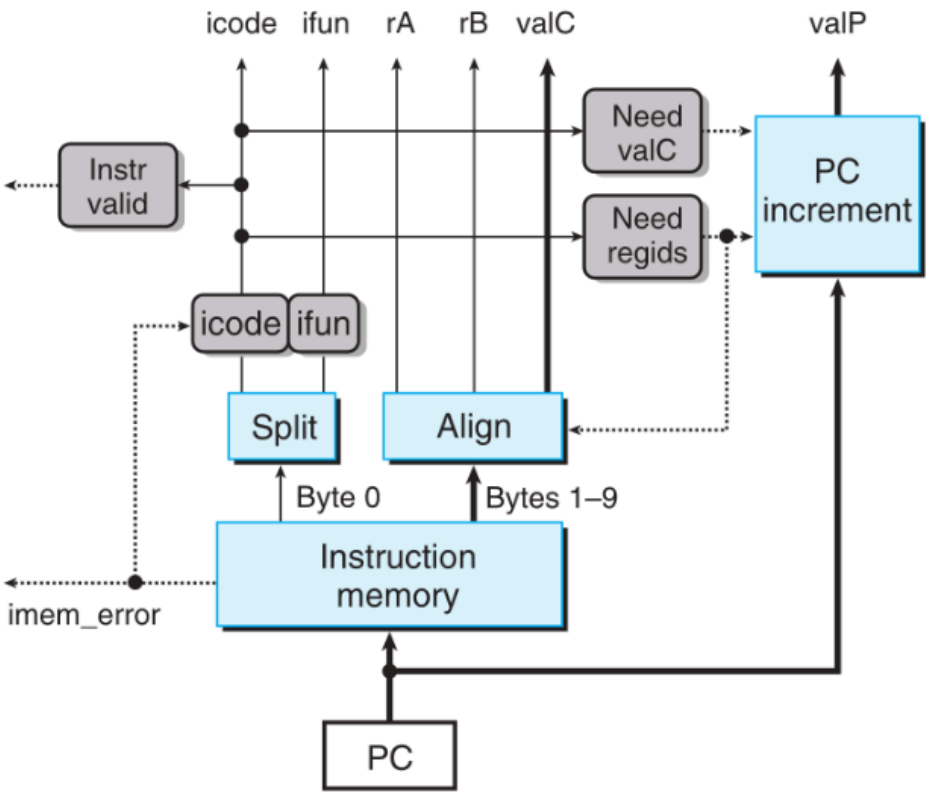
- Pipeline Implementation:
  - ❖ The real excitement begins when we transition to the pipelined architecture. Here, we break down the processor into five distinct stages: Fetch, Decode, Execute, Memory, and Writeback.
  - ❖ Throughout this phase, we tackle challenges related to data dependencies, forwarding, stalls, and halt instructions.
  - ❖ The pc-update stage, present in the sequential design, takes a backseat in the pipeline version.
- Test-Bench Implementation:
  - ❖ To validate our work, we create a comprehensive test-bench that exercises all stages of the processor.
  - ❖ By comparing the sequential and pipelined versions, we gain insights into their performance, trade-offs, and efficiency.

Sequential Implementation

In our endeavor to design a processor architecture that optimally utilizes hardware resources, we have structured the instruction processing into distinct stages, ensuring a uniform sequence for all instructions despite their varied functionalities. Each stage encapsulates specific operations tailored to efficiently execute instructions. Here's a breakdown of our approach:

- **Fetch Stage**

  At the fetch stage, we read instruction bytes from memory using the Program Counter (PC) as the memory address. Extracting the instruction specifier byte, denoted as `icode` and `ifun`, is our primary task. Additionally, we fetch register specifier bytes, potentially acquiring register operand specifiers `rA` and `rB`. Moreover, we compute the address of the next sequential instruction, `valP`, by adding the length of the fetched instruction to the current PC value. We also might need `valC` as the input. This might represent an immediate value or a constant operand required for computation. For example, in arithmetic or logical operations, `valC` is used to compute the effective address or the target address for the next instruction. This stage also outputs two errors: `instr_valid`, indicating the validity of the instruction, and `imem_error`, which flags when the instruction address is out of bounds, thereby contributing to the generation of the status code in the memory stage. A block diagram representation of this stage is as follows:



FETCH.v

```v
module fetch(
  input clk,
  input [63:0] PC,
  output reg [3:0] icode,
  output reg [3:0] ifun,
  output reg [3:0] rA,
  output reg [3:0] rB,
  output  reg [63:0] valC,
  output  reg [63:0] valP,
  output reg instr_valid,
  output reg imem_error,
  output reg need_regids,
  output reg need_valC,
  output reg hlt
);

//the fetch stage includes the instruction memory hardware unit.
reg [7:0] instruction_memory[0:1023];
```

- **Combined Decode and Writeback Stages**

  In this stage, we perform both decoding of instructions and writing back of results to the register file.

  Decoding:

  During decoding, we extract up to two operands from the register file, resulting in values `valA` and `valB`.

  The selection of source registers (`srcA` and `srcB`) is determined based on the instruction type:

  For instructions such as `RRMOVQ`, `RMMOVQ`, `IOPQ`, and stack-related instructions (`IPUSHQ, IPOPQ, ICALL, IRET`), `srcA` and `srcB` are determined based on the instruction code and register specifiers `rA` and `rB.`

  In the case of conditional move instructions, srcA and srcB are determined based on the condition computed in the Execute stage.

  If no register reading is necessary, `NOP` is selected as the source register.

  Write Back:

  Once the instruction is decoded and executed, the results are written back to the register file.

  The destination registers for write ports `E` (execute) and `M` (memory) are determined based on the instruction type:

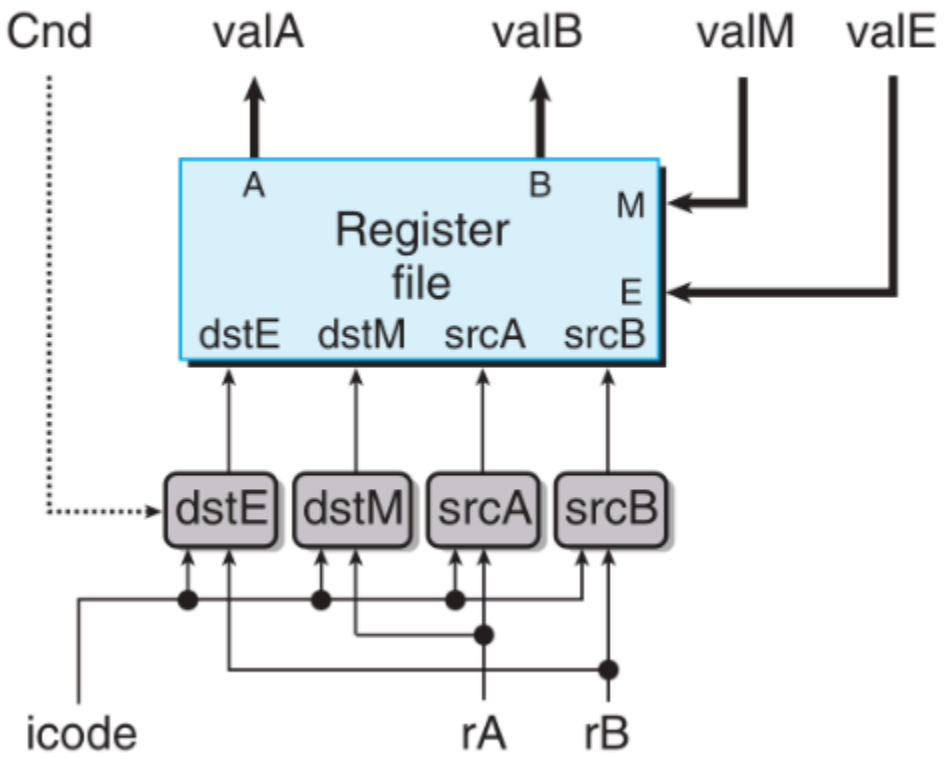
For instructions such as `RRMOVQ, IRMOVQ`, and arithmetic/logical operations (`OPQ`), the destination register for write port E is specified by the rB field of the instruction.

For stack-related instructions (`IPUSHQ, IPOPQ, ICALL, IRET`), the destination register for write port `E` is `%rsp`.

In cases where no register write is needed, `NOP` is selected as the destination register.

This combined stage optimizes the utilization of the register file by performing both decoding and write back operations efficiently, contributing to the seamless execution of instructions in the processor pipeline. A diagram of the Decode and Writeback code is given below.

```
module decode_wb(
    input clk,
    input Cnd,
    input[3:0] icode,
    input[3:0] rA,
    input[3:0] rB,
    input signed [63:0] valE,
    input signed [63:0] valM,
    output reg signed [63:0] valA,
    output reg signed [63:0] valB,
    output reg signed [63:0] register_value0,
    output reg signed [63:0] register_value1,
    output reg signed [63:0] register_value2,
    output reg signed [63:0] register_value3,
    output reg signed [63:0] register_value4,
    output reg signed [63:0] register_value5,
    output reg signed [63:0] register_value6,
    output reg signed [63:0] register_value7,
    output reg signed [63:0] register_value8,
    output reg signed [63:0] register_value9,
    output reg signed [63:0] register_value10,
    output reg signed [63:0] register_value11,
    output reg signed [63:0] register_value12,
    output reg signed [63:0] register_value13,
    output reg signed [63:0] register_value14
  );

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

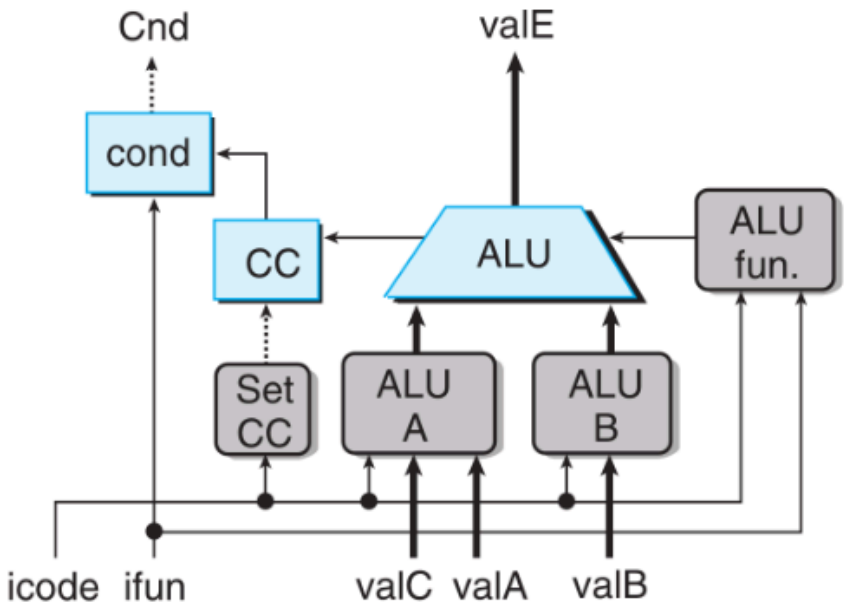


- **Execute Stage**

The execute stage encompasses the Arithmetic/Logic Unit (ALU), which performs operations such as addition, subtraction, logical `AND`, logical `OR`, and exclusive `OR` on inputs `aluA` and `aluB`, determined by the `icode:ifun` signal. The ALU output, `valE`, is derived from these computations. Additionally, the ALU generates control signals for condition code updates—`zeroflag`, `signgflag`, and `overflowflag`—whenever it operates. However, the condition code register is only updated when executing `OPq` instructions, controlled by the signal `set_cc`.

Moreover, the execute stage incorporates a hardware unit labeled "cond" responsible for evaluating conditions for conditional branches or data transfers. This unit utilizes a combination of condition codes and function codes to determine the Cnd signal, which is crucial for conditional moves and next PC logic in conditional branches. While Cnd may be set to either 1 or 0 for certain instructions, it is disregarded by the control logic for others. Further elaboration on the design of this unit is not provided.

In summary, the execute stage orchestrates ALU operations, manages condition code updates, and evaluates conditions for conditional branches, contributing significantly to the execution flow and control logic within the processor architecture. A diagram representing how this stage executes is given below.



```
`include "./alu.v"

module execute(
    input clk,
    input [3:0] icode, ifun,
    input  signed [63:0] valA, valB, valC,
    // input [2:0] CC,
    output reg [63:0] valE,
    output reg cnd,
    output reg [2:0] outCC
);

//defining the output of intermediate ALU instances and overflow values
wire [63:0] result_BC, result_add, result_and, result_sub, result_xor, result_IN, result_DE;
wire overflow1, overflow2, overflow3, overflow_add, overflow_and, overflow_sub, overflow_xor;       //temporary storage values
wire overflow;


//instantiating the ALU blocks
alu_block alu_int1(.A(valB), .B(valC), .S(2'b00), .result(result_BC), .overflow(overflow1));        //used in the calculation
alu_block alu_int3(.A(valB), .B(64'd8), .S(2'b00), .result(result_IN), .overflow(overflow2));       //used in the calculation
alu_block alu_int4(.A(valB), .B(64'd8), .S(2'b01), .result(result_DE), .overflow(overflow3));       //used in the calculation

alu_block alu_add(.A(valA), .B(valB), .S(2'b00), .result(result_add), .overflow(overflow_add));
alu_block alu_sub(.A(valB), .B(valA), .S(2'b01), .result(result_sub), .overflow(overflow_sub));
alu_block alu_and(.A(valA), .B(valB), .S(2'b10), .result(result_and), .overflow(overflow_and));
alu_block alu_xor(.A(valA), .B(valB), .S(2'b11), .result(result_xor), .overflow(overflow_xor));
```

- ## Memory Stage

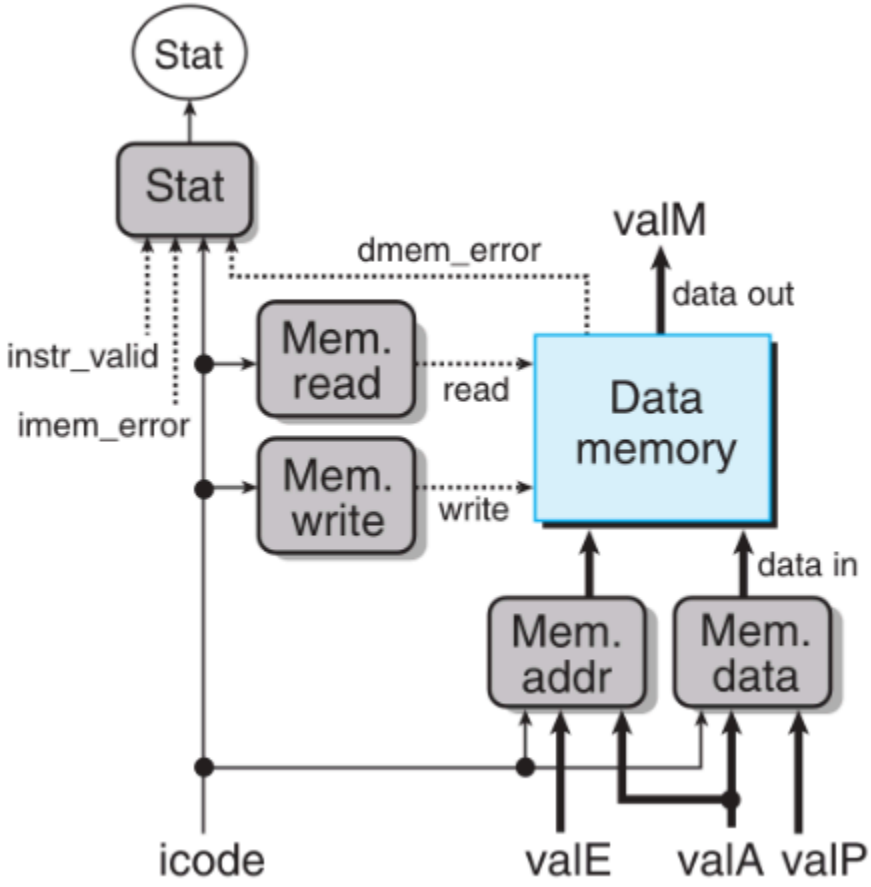
  The memory stage is responsible for handling program data read or write operations. Two control blocks are tasked with generating the memory address and memory input data (for write operations), while two additional blocks generate control signals to indicate whether a read or write operation should be performed. When a read operation is executed, the data memory produces the value `valM`.

  In the memory stage, the desired memory operation for each instruction type is determined. Notably, the memory address for both read and write operations is consistently derived from either `valE` or `valA`.

  Moreover, a crucial function of the memory stage is to compute the status code, Stat, which reflects the outcome of the instruction execution. This computation is based on the values of `icode`,

`imem_error`, and `instr_valid` generated during the fetch stage, as well as the signal `dmem_error` generated by the data memory. The status code encapsulates important information regarding the success or failure of the instruction execution process, aiding in the overall management of program execution flow within the processor architecture. A diagrammatic representation of this stage is given below.

```
module memory(clk, icode, valA, valE, valM, valP, dmem_err);
    input clk;
    input [3:0] icode;
    input signed [63:0] valA, valP;
    input  signed [63:0] valE;
    output reg [63:0] valM;
    output reg dmem_err;

    reg [63:0] memory[0:1023];//1kb

    always @(*)
    begin
        dmem_err = 1'b0;

        if (valE > 1024 )
        begin
            $display("value of valE:%d",valE);
            dmem_err = 1'b1;
        end
        else
        begin
            if (icode == 4'b0101)   //mrmovq
                valM = memory[valE];
            else if (icode == 4'b1001 || icode == 4'b1011)   //ret or popq
                valM = memory[valA];
            else if (icode == 4'b0100 || icode == 4'b1010 || icode == 4'b1000)   //rmmovq, pushq, call
                memory[valE] = (icode == 4'b1000) ? valP : valA;


        end
    end

endmodule
```



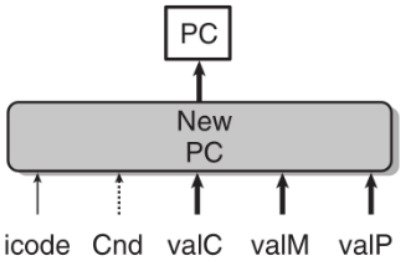
- **PC Update**

  The final stage in the SEQ processor architecture generates the new value of the program counter (PC). As depicted in the figure below, the new PC value is determined based on the instruction type and whether a branch should be taken. Specifically, the new PC can be one of the following:

  - ☐ `valC`: If the instruction type warrants a direct jump or call operation.
  - ☐ valM: If a memory access instruction results in a jump to an address stored in memory.

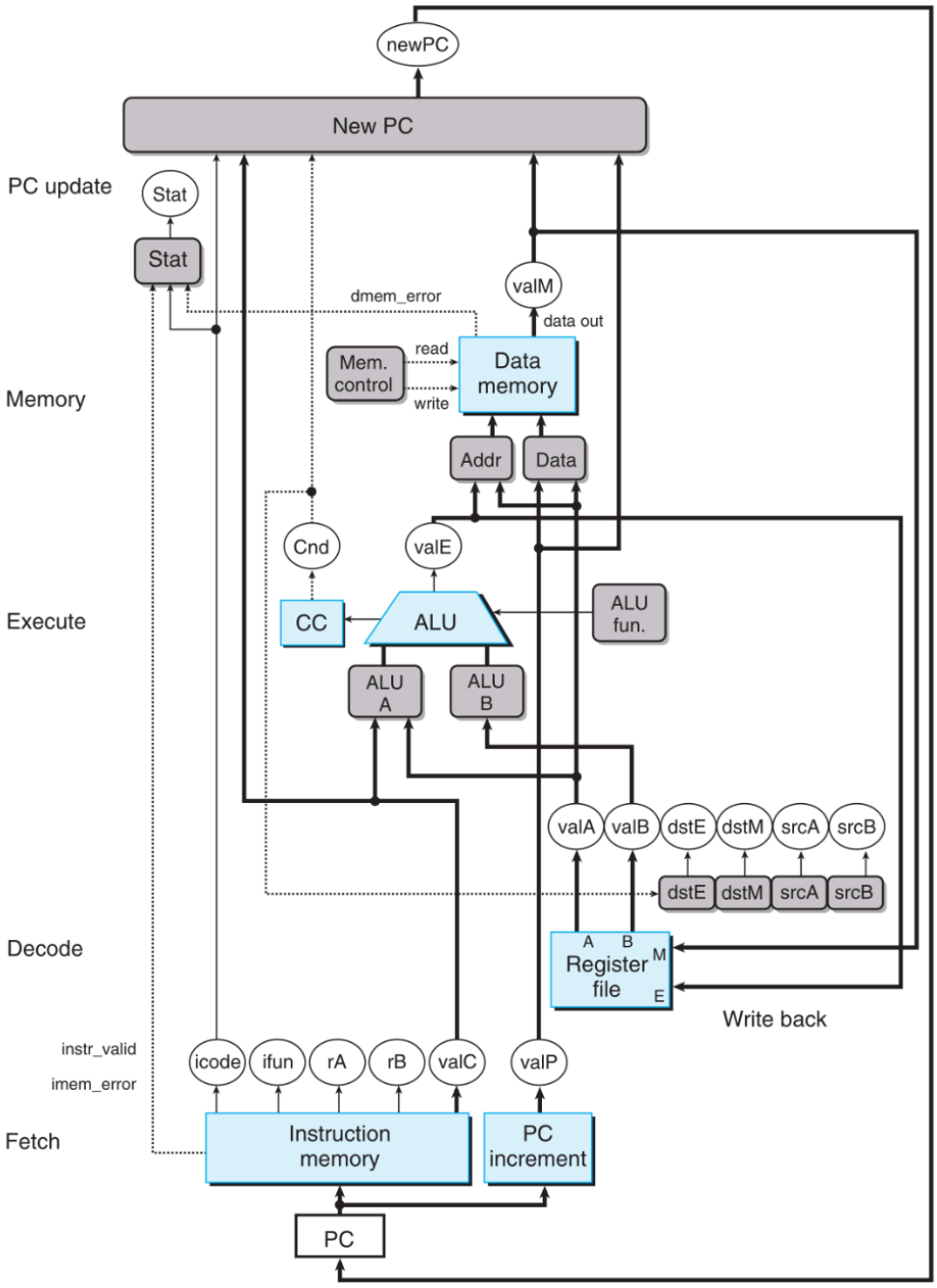
- ☐ `valP`: If neither a jump nor a call is executed, the PC is incremented to the address of the next sequential instruction.

This stage orchestrates the management of program flow by updating the PC to the appropriate address, ensuring the correct execution sequence within the processor architecture.




```
module pc_update(clk, icode, Cnd, PC_new, valC, valM, valP);

input clk;
input [3:0] icode;
input Cnd;
input [63:0] valC, valM, valP;
output reg  [63:0]  PC_new;

always@(negedge clk)
begin
    //if the instruction is a CALL (icode == 4'b1000) PC_new should be the target address of the call (valC)
    if (icode == 4'b1000)
        PC_new <= valC;

    //if the instruction is a conditional jump (JXX, icode == 4'b0111) and the condition is met (Cnd == true) PC_new should be
    else if (icode == 4'b0111)
    begin
        if (Cnd)
        begin
        PC_new = valC;
        end
        else
        begin
        PC_new = valP;
        end
    end

    //if the instruction is a RET (icode == 4'b1001) PC_new should be the return address from the stack (valM)
    else if (icode == 4'b1001)
        PC_new = valM;

    //for all other instructions PC_new should be the address of the next instruction (valP)
    else
        PC_new = valP;
end

endmodule
```

The final diagrammatic representation of the Sequential implementation is given below:



Pipeline Implementation

Pipelining implementation of the Y86 architecture represents a significant advancement in processor design, offering enhanced performance and efficiency compared to the Sequential (SEQ) implementation. In pipelining, the execution of instructions is divided into discrete stages, with each stage concurrently processing multiple instructions. This parallelism allows for higher throughput and better utilization of hardware resources, ultimately resulting in improved overall system performance

Unlike the Sequential implementation, where each instruction progresses through the processor stages one after the other, pipelining enables simultaneous execution of multiple instructions at different stages of processing. This parallel processing reduces the overall execution time of instruction sequences, as instructions overlap in their execution, effectively increasing the throughput of the processor

However, implementing pipelining introduces challenges such as pipeline hazards, where dependencies between instructions may cause stalls or incorrect results. Techniques such as forwarding and branch prediction are employed to mitigate these hazards and ensure smooth execution of pipelined instructions.

In essence, while both Sequential and Pipelining implementations aim to execute instructions in a controlled sequence, pipelining introduces parallelism to improve performance, making it a more sophisticated and efficient architecture compared to its Sequential counterpart. Let us now have a look at how we implemented the pipelining in Verilog.

The main differences between Pipelining and Sequential implementations is that the PC update stage comes before the Fetch stage. In fact it is a part of the Fetch stage. So, overall there are only five differentiable stages in Pipelining. The other difference is that there are registers after each stage that store the values of the previous stages and they are called the pipeline registers. Each pipeline register serves a specific purpose:

F holds a predicted value of the program counter.

D holds information about the most recently fetched instruction for decoding.

E holds information about the most recently decoded instruction and register file values for execution.

M holds the results of the most recently executed instruction, along with branch conditions and targets for conditional jumps.

W holds computed results for writing back to the register file and return addresses for PC selection logic when completing a ret instruction.

With our naming system, the uppercase prefixes 'D', 'E', 'M', and 'W' refer to pipeline registers, and so `M_stat` refers to the status code field of pipeline register M. The lowercase prefixes 'f', 'd', 'e', 'm', and 'w' refer to the pipeline stages, and so `m_stat` refers to the status signal generated in the memory stage by a control logic block.

PC Prediction

In designing PIPE, we have implemented measures to effectively handle control dependencies and ensure the issuance of a new instruction on every clock cycle, aiming for a throughput of one instruction per cycle. However, challenges arise when dealing with conditional branches and `ret` instructions, where the determination of the next instruction's location is deferred until several cycles later.

To address this, we utilize branch prediction techniques, wherein we predict the next value of the program counter based on information available during the fetch stage. For most instruction types, this prediction is reliable, with the next instruction's address being either `valC` or `valP`. In the case of conditional jumps, we predict that the branch will always be taken, setting the new PC value to `valC`. This strategy ensures a consistent flow of instructions, albeit with potential inaccuracies in prediction.

Regarding ret instructions, predicting the return address proves challenging due to the unbounded set of possible results. As a result, we opt not to predict the return address and instead wait until the `ret` instruction passes through the write-back stage to determine the new PC value.

As mentioned before, the Fetch stage of PIPE is responsible for both predicting the next PC value and selecting the actual PC for instruction fetch. The "Predict PC" block in this stage chooses between `valP` and `valC`, storing the predicted value in pipeline register F. Meanwhile, the "Select PC" block, determines the address for instruction memory based on various factors, ensuring the seamless progression of instruction fetch in the pipeline.

- **PC Selection and Fetch Stage**

  This stage is responsible for reading instructions from memory and extracting different instruction fields, utilizing hardware units similar to those used in the SEQ fetch stage.

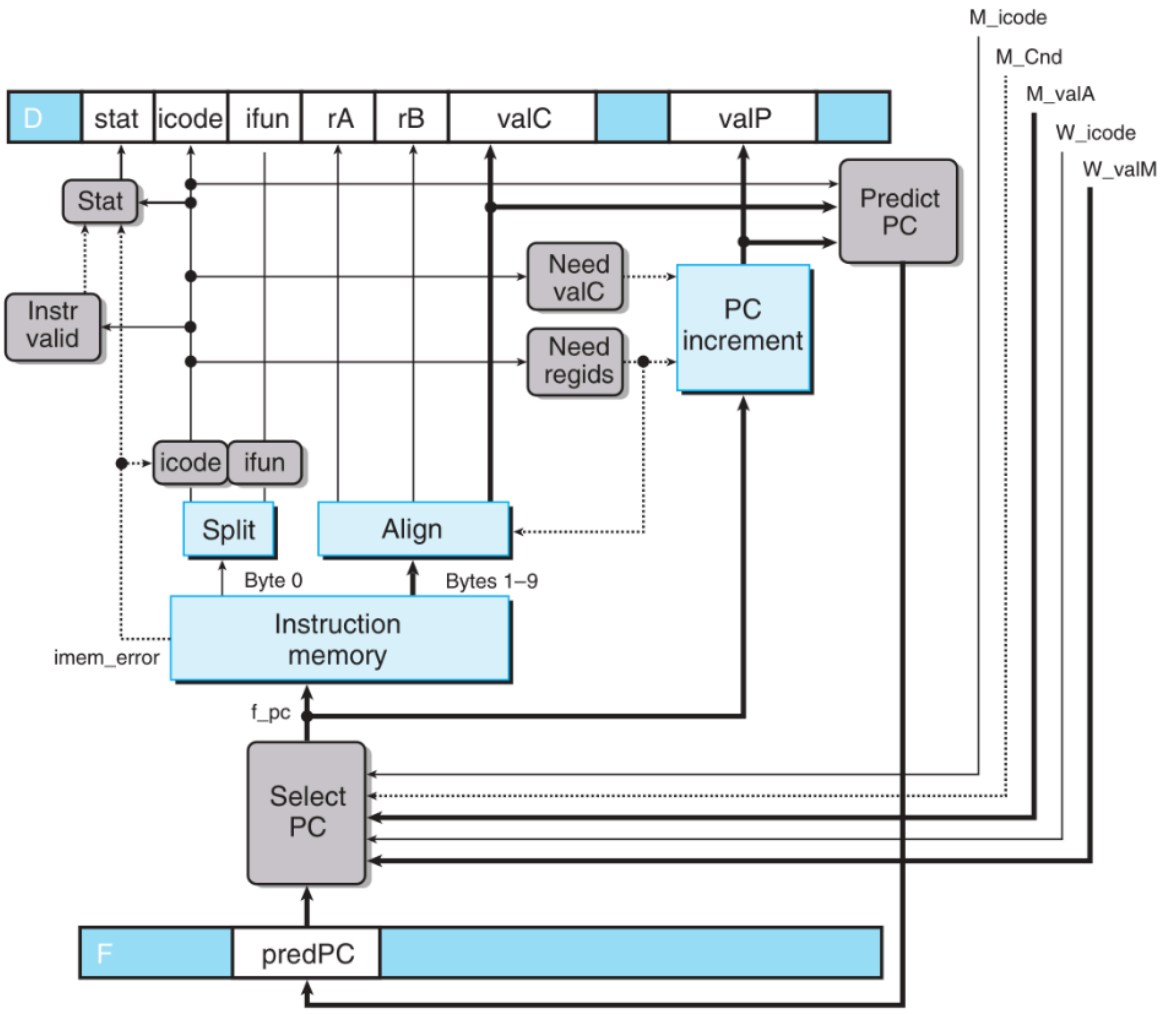
  The PC selection logic operates by choosing between three potential sources for the program counter. If a mispredicted branch enters the memory stage, indicating that the branch was not taken, the value of `valP` for this instruction (representing the address of the following instruction) is read from pipeline register `M`. On the other hand, when a `ret` instruction reaches the write-back stage, the return address is read from pipeline register `W`. For all other cases, the predicted value of the PC stored in pipeline register `F` (`F_predPC`) is used.

  The PC prediction logic determines the next PC value based on the fetched instruction type. If the instruction is a call or a jump (`JXX` or `CALL`), the next PC value (`f_predPC`) is set to `valC`. Otherwise, it defaults to `valP`, indicating the address of the next instruction in sequence.

  Additionally, the fetch stage includes logic blocks such as "`Instr_valid`" and "`Need_regids`" which are similar to those used in the SEQ design, with appropriately named source signals.

  Unlike in the SEQ implementation, the computation of instruction status is split into two parts in the fetch stage of PIPE. Here, it's possible to test for a memory error due to an out-of-range instruction address or to detect an illegal instruction or a halt instruction.

However, detecting an invalid data address must be deferred to the memory stage for further processing. A diagrammatic representation of the code is given below.



- **Decode and Writeback Stages**

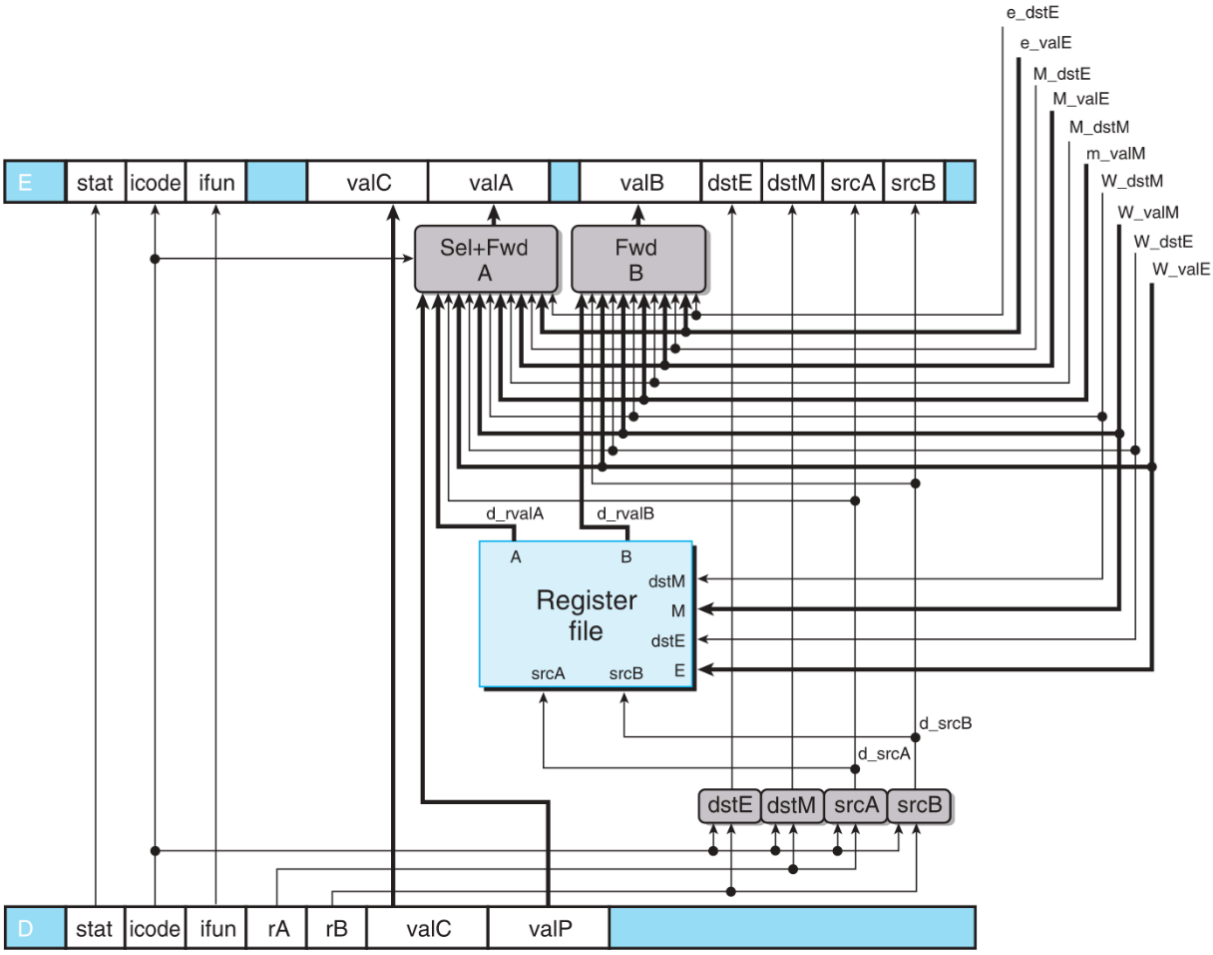
In the PIPE processor, the blocks responsible for determining the destination registers (`dstE` and `dstM`) and fetching source operands (`srcA` and `srcB`) closely resemble their counterparts in the SEQ implementation. However, there's a difference in where the register IDs for the write ports originate. Instead of coming from the decode

stage, they are sourced from the write-back stage (signals `W_dstE` and `W_dstM`). This ensures that writes occur to the destination registers specified by the instruction during the write-back stage.

The complexity of this stage primarily lies in the forwarding logic. The block labeled "Sel+Fwd A" not only combines the valP signal with `valA` to reduce pipeline register state but also handles forwarding logic for source operand `valA`.

For merging signals `valA` and `valP`, we exploit the fact that only call and jump instructions require the valP value in later stages, while these instructions do not require the value read from the A port of the register file. Selection is controlled by the instruction code (`icode`) signal for this stage. When `D_icode` matches the instruction code for either `call` or `JXX`, the block selects `D_valP` as its output.

In the write-back stage, the overall processor status (`Stat`) is computed based on the status value in pipeline register `W`. This status indicates normal operation (`AOK`) or one of three exception conditions. Pipeline register `W` holds the state of the most recently completed instruction, making it suitable for indicating the overall processor status. The only special consideration is when there's a bubble in the write-back stage. The diagrammatic representation for the same stage has been displayed below.

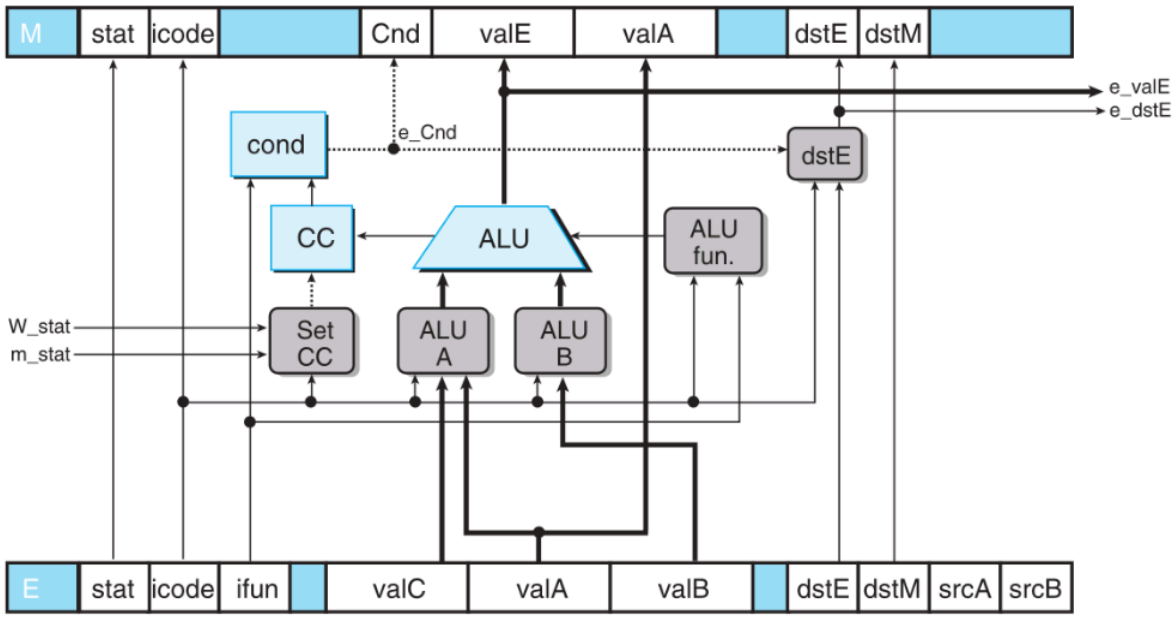


- **Execute Stage**

  In the PIPE processor, the below figure illustrates the execute stage logic, which closely mirrors that of the SEQ implementation. The hardware units and logic blocks are identical, albeit with appropriate signal renaming. Notably, signals like `e_valE` and `e_dstE` are forwarded to the decode stage as one of the forwarding sources.

  However, a difference lies in the logic labeled "`set_CC`," responsible for determining whether to update the condition codes. Here, signals `m_stat` and `W_stat` serve as inputs. These signals are crucial for

detecting cases where an instruction causing an exception is progressing through later pipeline stages, necessitating the suppression of any condition code updates.

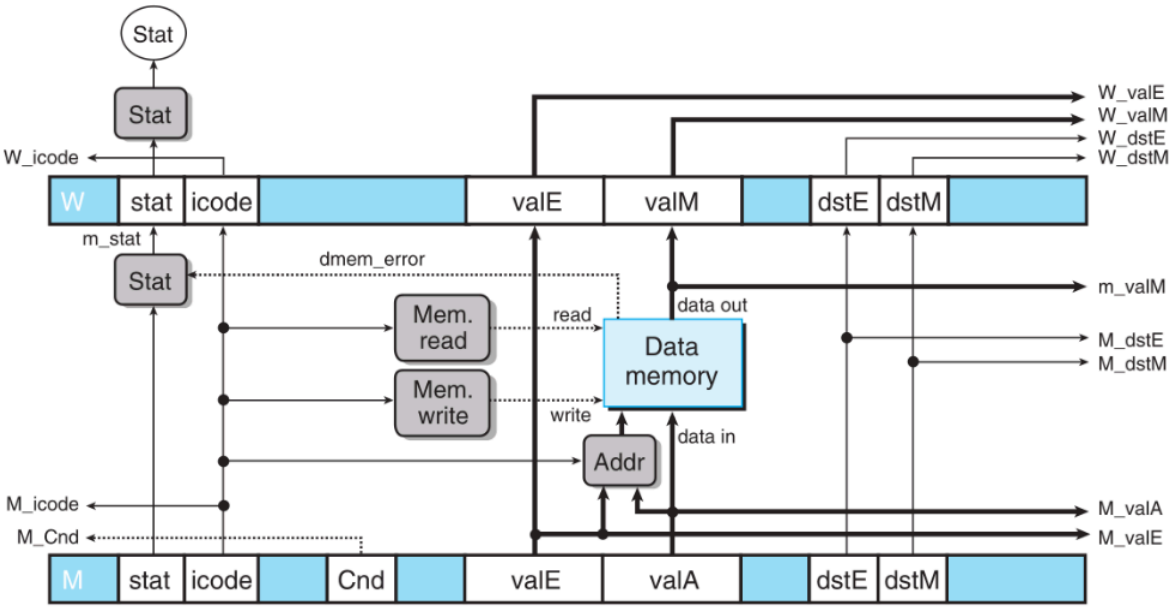


- **Memory Stage**

  In the PIPE processor, the figure outlines the memory stage logic, which differs slightly from the memory stage in SEQ. Notably, the block labeled "Mem. data" in SEQ, responsible for selecting between data sources valP (for call instructions) and `valA`, is absent in PIPE. Instead, this selection is handled by the "Sel+Fwd A" block in the decode stage.

  However, most other blocks in this stage remain identical to their SEQ counterparts, albeit with appropriate signal renaming. Furthermore, it's evident that many values in pipeline registers `M` and `W` are supplied to other parts of the circuit as part of the forwarding and pipeline control

logic. This ensures smooth data flow and efficient operation within the pipeline architecture of the PIPE processor.



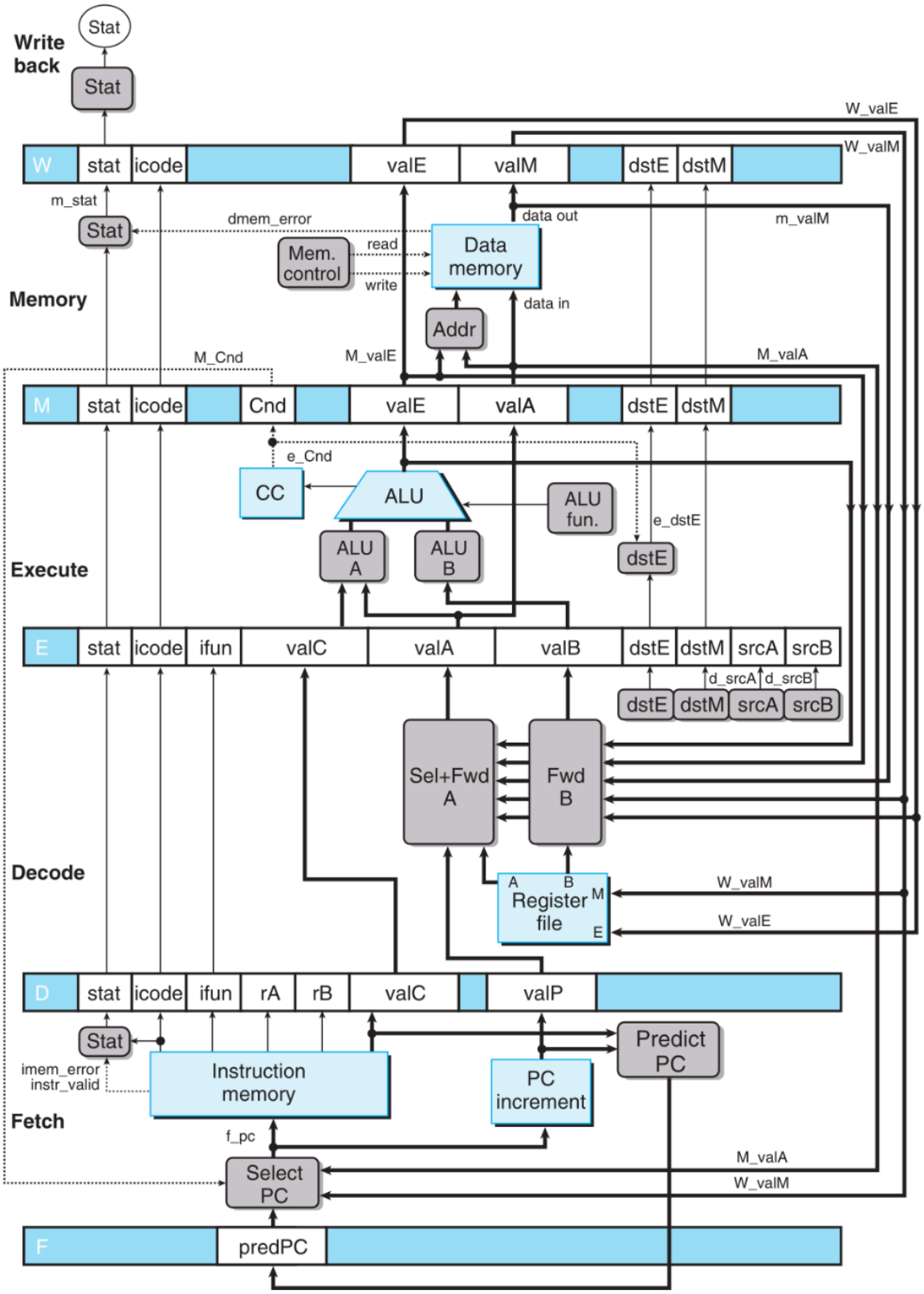
- **Pipeline Control Logic**

Now, we proceed to finalize the PIPE design by implementing the pipeline control logic. This component is crucial for managing four specific control cases where other mechanisms such as data forwarding and branch prediction are insufficient:

- ☐ Load/use hazards: To address situations where an instruction reads a value from memory and another instruction uses this value immediately after, we need to introduce a stall cycle in the pipeline.
- ☐ Processing ret: The pipeline must pause until the ret instruction reaches the write-back stage to ensure correct handling of return address updates.

- ☐ Mispredicted branches: In cases where a branch instruction is mispredicted, resulting in instructions being fetched incorrectly, we need to cancel these instructions and resume fetching from the correct instruction address.
- ☐ Exceptions: When an instruction triggers an exception, it's essential to prevent subsequent instructions from updating the programmer-visible state and halt execution once the excepting instruction reaches the write-back stage.

The final diagrammatic representation of the Pipeline implementation of Y86 is as follows:


Write back
Stat
Stat
W_valE
W_valM
W
stat
icode
valE
valM
dstE
dstM
m_stat
Stat
dmem_error
data out
m_valM
Mem. control
read
write
Data memory
Memory
data in
Addr
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_dstE
ALU A
ALU B
dstE
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
A
B
Register file
M
E
W_valM
W_valE
Decode
D
stat
icode
ifun
rA
rB
valC
valP
Stat
imem_error
instr_valid
Instruction memory
PC increment
Predict PC
Fetch
f_pc
M_valA
W_valM
Select PC
F
predPC

Thank You!

Done by : Team 14

Member-1 : Rishiveer Yadav Angirekula

Roll No : 2023122004

Member-2 : Sahil

Roll No: : 2023122006