Intro to Processor Architecture

PROJECT REPORT

TEAM 7

Team members:

Name : T.Sri Vishnuvarun

Roll no: 2022102031

Email : thorthi.varun@students.iiit.ac.in

Name : S.Rohit

Roll no: 2022102001

Email : rohit.sunkari@students.iiit.ac.in

Sequential Implementation of Y86 architecture:

Fetch:

We must read each instruction from the instruction memory one at a time during the fetch stage in order to determine the values of icode, ifun, rA, rB, and valC in accordance with the instruction.

For the convenience of simulation, I have specified the instruction memory as a register array instr_mem in my fetch stage:

reg [7:0] instr_mem[0:1023];

The fetch stage operates upon receiving an updated PC value triggered by the positive edge of the clock signal. It then proceeds to access the instruction memory to fetch the corresponding instruction. Should the PC value exceed the bounds of the instruction memory, indicating an "imem_error," the processor executes a "nop" operation, effectively idling across all stages for that clock cycle.

Assuming the PC value is within the valid range, the fetch stage retrieves 10 consecutive bytes from the instruction memory, starting from the byte pointed to by the PC. These bytes are combined into a single 10-byte register called "instr," equating to an 80-bit value. This consolidation is performed to accommodate the maximum size of a single instruction, which is 10 bytes.

In the fetch module of our sequential implementation, the process begins by fetching the instruction from memory based on the updated Program Counter (PC) value triggered by the clock signal's positive edge. The fetched instruction is stored in a register called "instr."

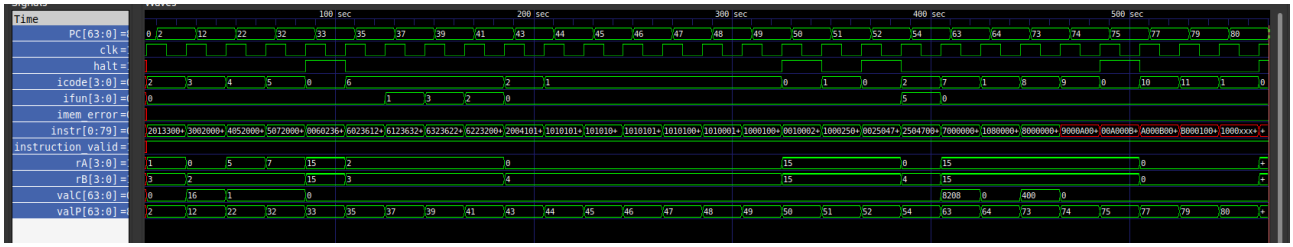
From this "instr" register, we extract the "icode" and "ifun" values by splitting the instruction into respective bit ranges. These values determine whether the instruction requires register IDs (need_regisids) and a value constant (valC).

Further, depending on the "icode" value, other output values are derived from "instr" as follows:

- "rA" and "rB" are obtained from specific bit ranges within "instr."
- "valC" is extracted from "instr" for certain instructions, with different bit ranges used based on the instruction type.
-

Additionally, the Program Counter (valP) is computed based on the "icode" value. Different increments are added to the current PC value depending on the type of instruction being executed.

If the "icode" is found to be invalid, indicating an invalid instruction, the "instr_valid" signal is set to 0, indicating that the fetched instruction is not valid. This ensures proper error handling and prevents the execution of erroneous instructions.



This is the output of fetch stage when tested individually

Decode and Writeback:

In the decode stage, we access the register memory of the processor, represented by a register array named "reg_mem[0:14]," within the decode and write-back module. This setup allows convenient simulation since only these two stages require access to the register memory.

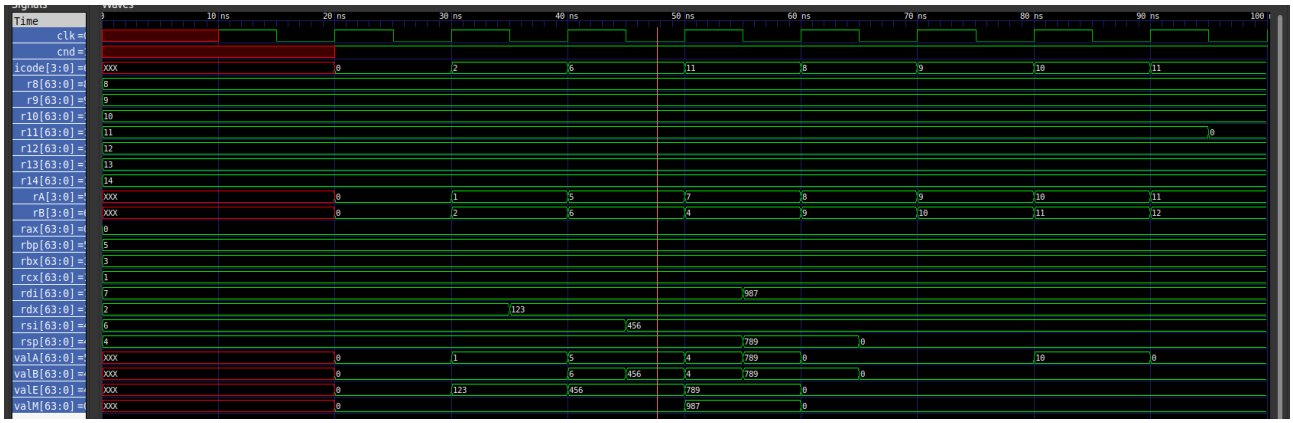
To determine the values of "valA" and "valB," we use the "rA" and "rB" indices to reference the "reg_mem" register array. Depending on the "icode" of the instructions, we assign values to "valA" and "valB" as follows:

- For the "cmovxx" instruction, "valA" is assigned the value stored in "reg_mem[rA]."
- For instructions like "rmmovq" and "OPq," "valA" is assigned the value stored in "reg_mem[rA]," and "valB" is assigned the value stored in "reg_mem[rB]."
- For the "call" instruction, "valB" is assigned the value stored in "reg_mem[4]," representing the "rsp" register.
- For instructions like "ret" and "popq," both "valA" and "valB" are assigned the value stored in "reg_mem[4]," representing the "rsp" register.

In the write-back stage, we need to write the values "valE" or "valM" into registers "rA," "rB," or the "rsp" register based on the "icode" of the instruction being executed. This operation is performed as follows:

- For the "cmovxx" instruction, if the condition "cnd" is true, "valE" is written into "reg_mem[rB]."
- For instructions like "irmovq" and "OPq," "valE" is written into "reg_mem[rB]."
- For the "mrmovq" instruction, "valM" is written into "reg_mem[rA]."
- For instructions like "call," "ret," and "pushq," "valE" is written into "reg_mem[4]," representing the "rsp" register.
- For the "popq" instruction, "valE" is written into "reg_mem[4]," and "valM" is written into "reg_mem[rA]." Here, "reg_mem[4]" represents the "rsp" register.

These operations are depicted by the corresponding blocks in the architecture diagram, namely "srcA," "srcB," "dstE," "dstM," and "register file."

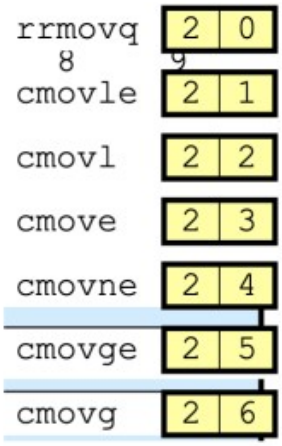


decode write back module output when tested individually

Execute:

The execute stage encompasses the ALU and operates based on the icode and ifun of an instruction.

- If an instruction necessitates an ifun, the value assigned to valE is determined accordingly.
- For the cmov instruction:
    - Various move conditions are checked based on the ifun.
    - Depending on the outcome of these conditions:
        - cnd is set to 1 if the conditions are met.
        - cnd remains zero if the conditions are not fulfilled.



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 <=)

In the execution stage, the value of valE is set to valC for the irmovq instruction.

- For the mrmovq and rmmovq instructions, valE is determined as the sum of valB and valC.
- Instructions labeled as OPx require further examination of ifun values.

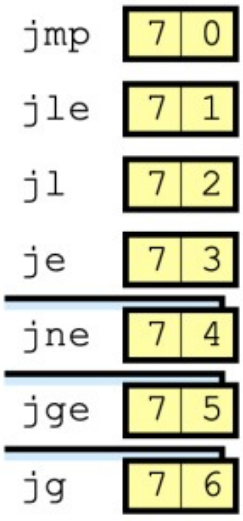
- The ALU is employed to execute the necessary operation specified by the ifun.

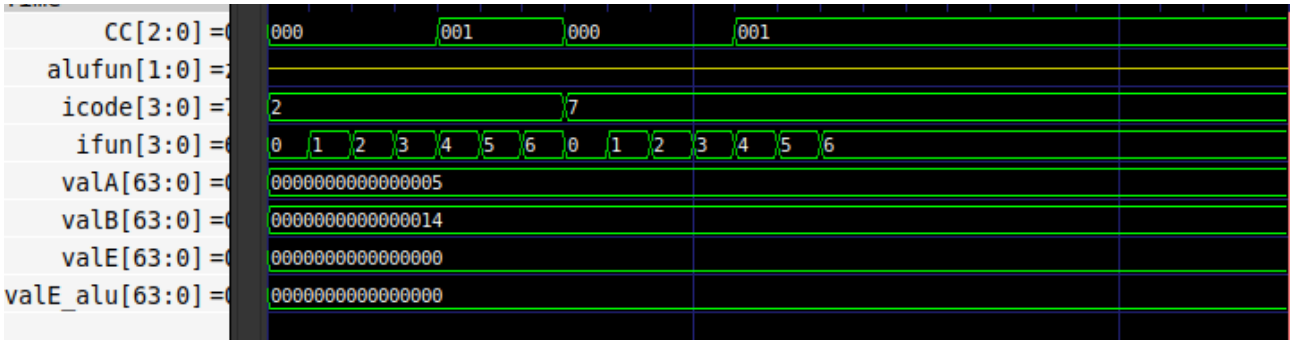
In the case of jXX instructions, the ifun value is examined to determine the specific jump condition.

- Similar to move conditions, the jump conditions are verified to see if they are met.
- If the jump conditions are satisfied, the program execution will proceed accordingly.
- Otherwise, the program will continue without branching.

For the call and pushq instructions, valE is computed as -64'd8 added to the value of valB.

- Conversely, for the ret and popq instructions, valE is determined as 64'd8 added to the value of valB.





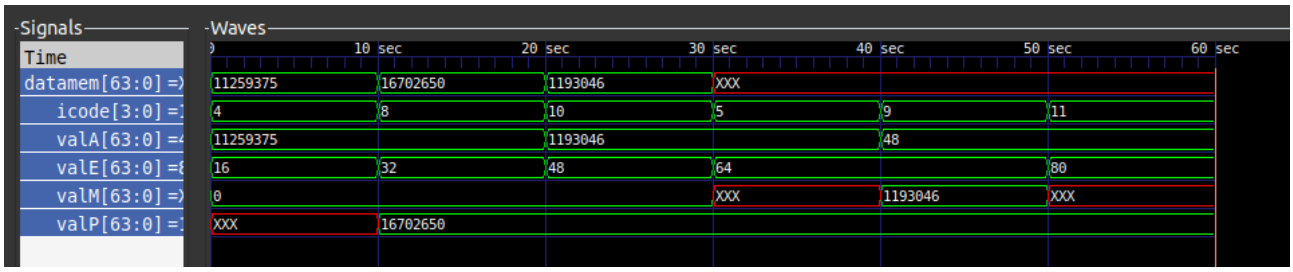
Execute stage output when tested individually

Memory:

The memory stage handles read and write operations to memory.

- Data memory is declared as a register array, data_mem[0:255], solely within the memory stage.
- Based on the instruction code (icode), the stage determines whether to perform memory reads or writes.
- Memory address calculation involves icode, valE, and valA, while data input calculation involves icode, valA, and valP.
- For instructions such as rmmovq, call, and pushq, the stage writes to memory:
    - For rmmovq and pushq, data_mem[valE] is assigned valA.
    - For call, data_mem[valE] is assigned valP.
- For instructions like mrmovq, ret, and popq, the stage reads from memory:
    - For mrmovq and popq, valM is assigned data_mem[valE].
    - For ret, valM is assigned data_mem[valA].

- These operations are represented by memory read, memory write, memory address, and memory data blocks within the architecture.

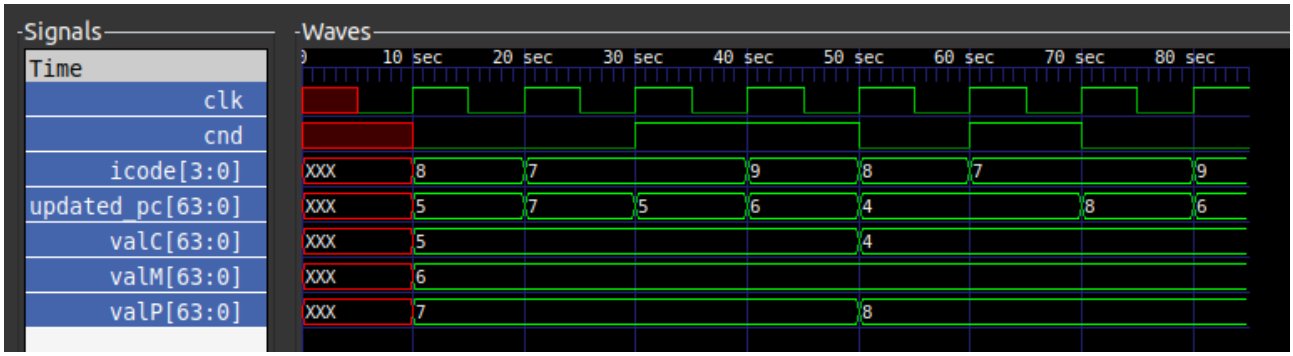


Output of memory block when tested individually

PC update:

The PC update module determines the next value of the program counter (PC) after an instruction completes execution.

- For most instructions, the PC is updated by adding the instruction's length in bytes.
- Specifically:
  - For jXX instructions, if the condition (cnd) is met, the updated PC is set to valC; otherwise, it's set to valP.
  - For call instructions, the updated PC is assigned valC.
  - For ret instructions, the updated PC is assigned valM.
  - For all other instructions, the updated PC is assigned valP.

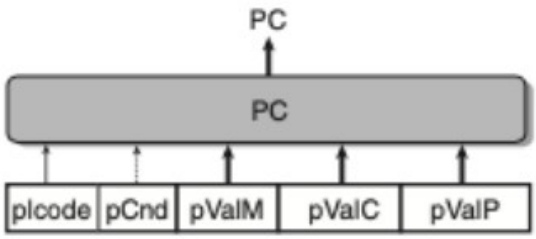


Pipelining Implementation of Y86 architecture:

The first step to pipelining is the rearrangement of computation stages.

Rearranging stages:

- In the pipelined implementation, the PC update stage is relocated to the beginning of the cycle.
- This adjustment enables the continuous fetching of the next instruction without waiting for the previous instruction's PC update stage to finish.
- Known as circuit retiming, this change alters the overall structure of the circuit while preserving its local behavior.
- With this modification, updated PC values from various stages of instructions that have progressed beyond those stages are supplied to the fetch stage.
- Balancing delays between stages in the pipelined system becomes feasible due to this repositioning of the PC update stage at the cycle's outset.



Inserting pipeline registers:

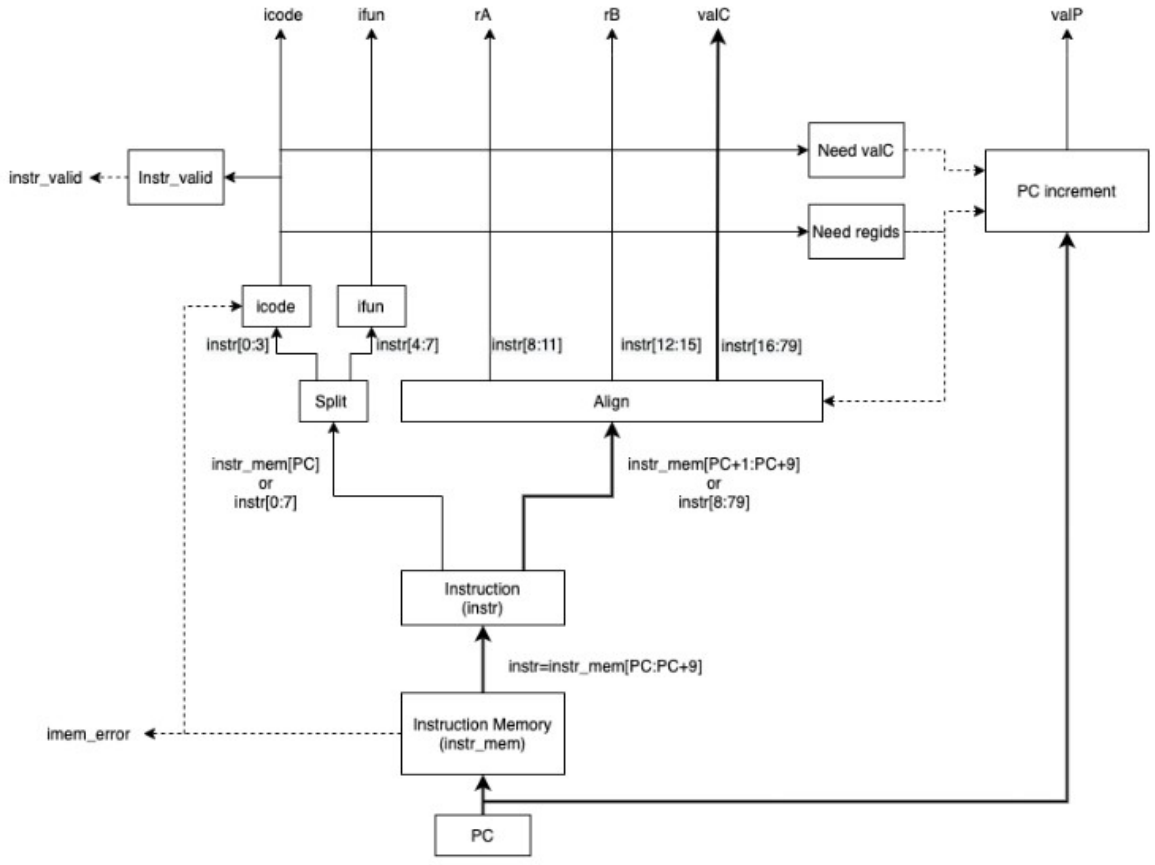
- In the pipeline insertion process, pipeline registers are introduced between each stage of the SEQ implementation.
- These registers serve to halt the flow of signals from one stage to the next, preventing interference with ongoing processing.
- Specifically:
    - The register before the fetch stage holds a predicted value of the program counter.
    - The D register is positioned between the fetch and decode stages, containing information on the most recently fetched instruction for decode stage processing.
    - Between the decode and execute stages lies the E register, which stores details of the recently decoded instruction and values retrieved from the register file for execute stage processing.
    - The M register is located between the execute and memory stages, housing the results of the most recently executed instruction for memory stage processing. Additionally, it retains information related to branch conditions and targets for conditional jumps.
    - Lastly, the W register resides between the memory stage and the feedback paths, holding computed results for writing back to the register file and supplying the return address to the PC selection logic during the completion of a ret instruction.

Rearranging and relabelling signals:

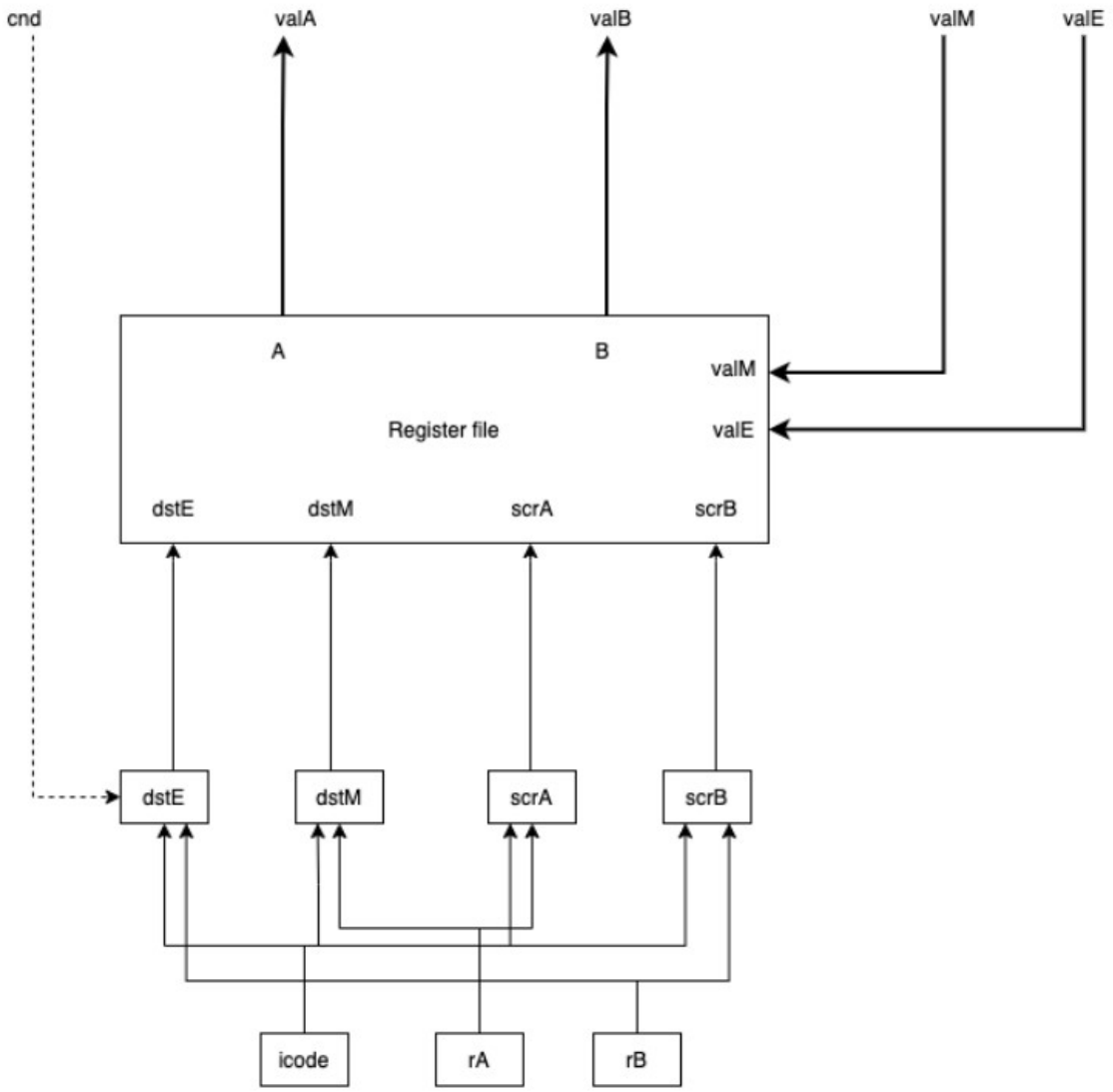
In the pipelined implementation, signals for each instruction pass through every stage sequentially. To manage these signals effectively, they are named according to the stage they are currently in. This naming convention ensures clarity and avoids confusion when multiple instructions are being processed simultaneously. For instance, signals are labeled as f_icode, d_icode, w_icode, etc., corresponding to the fetch, decode, and write-back stages, respectively.

Architecture diagram

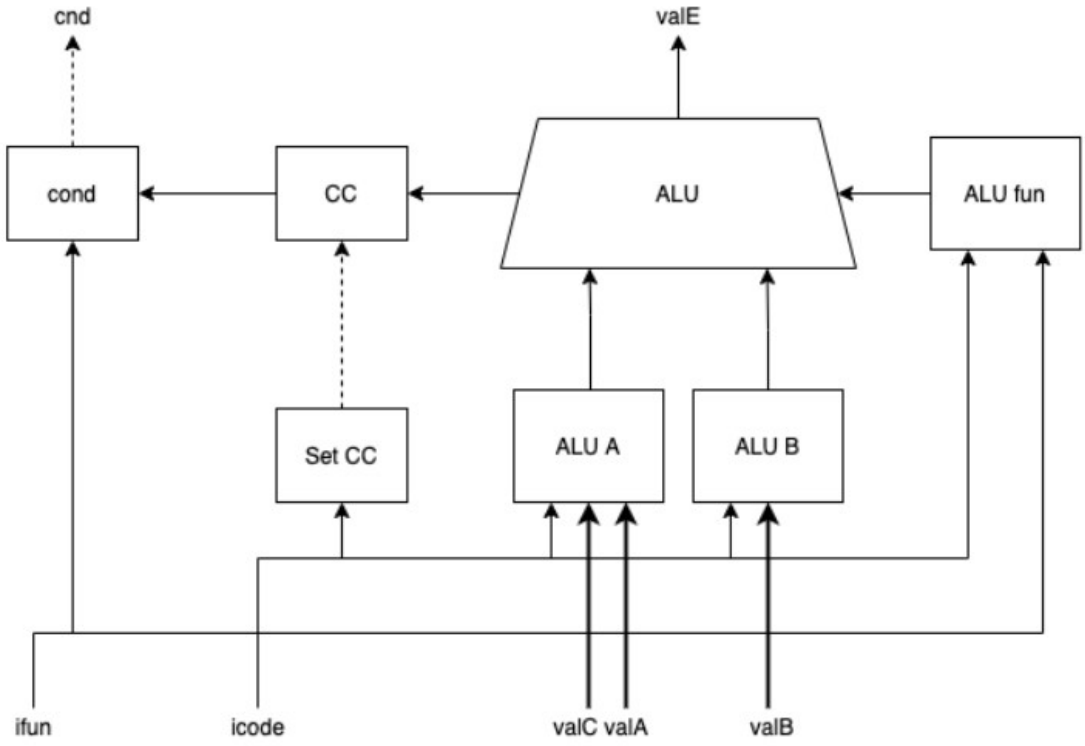
FETCH STAGE:



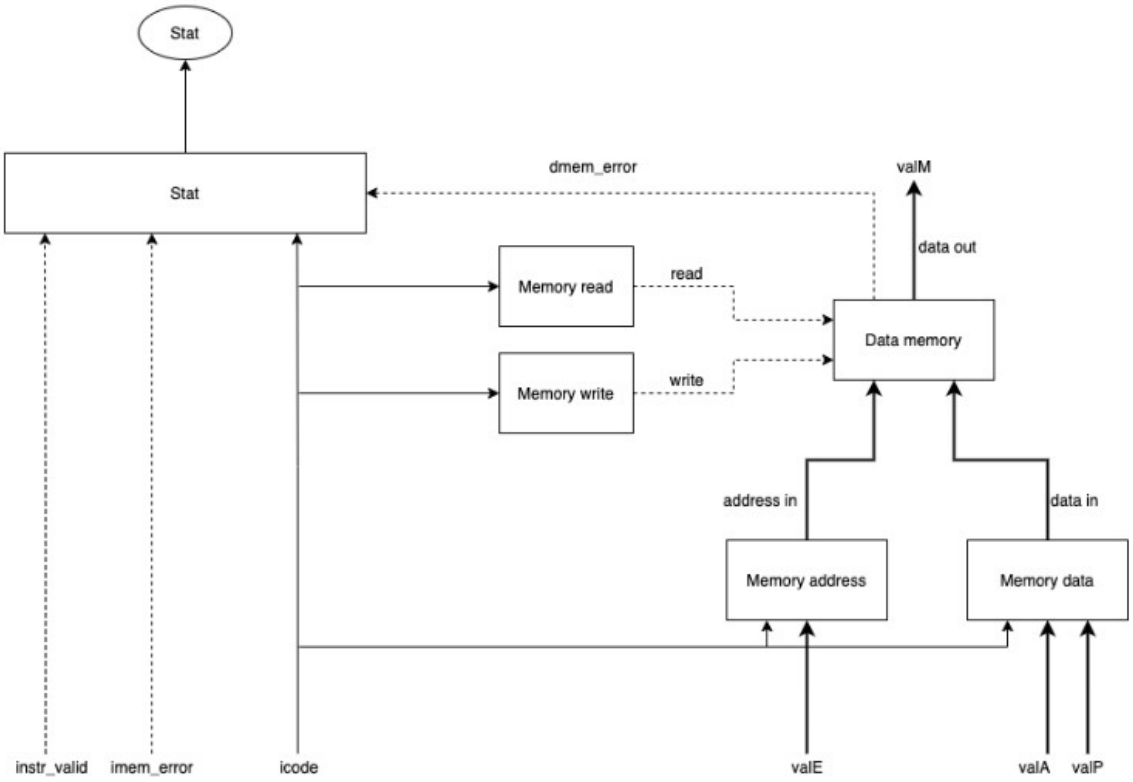
DECODE AND WRITEBACK STAGES:



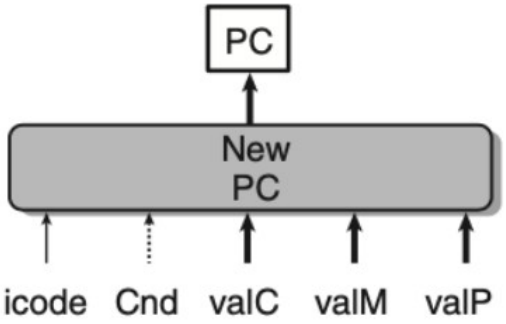
EXECUTE STAGE:



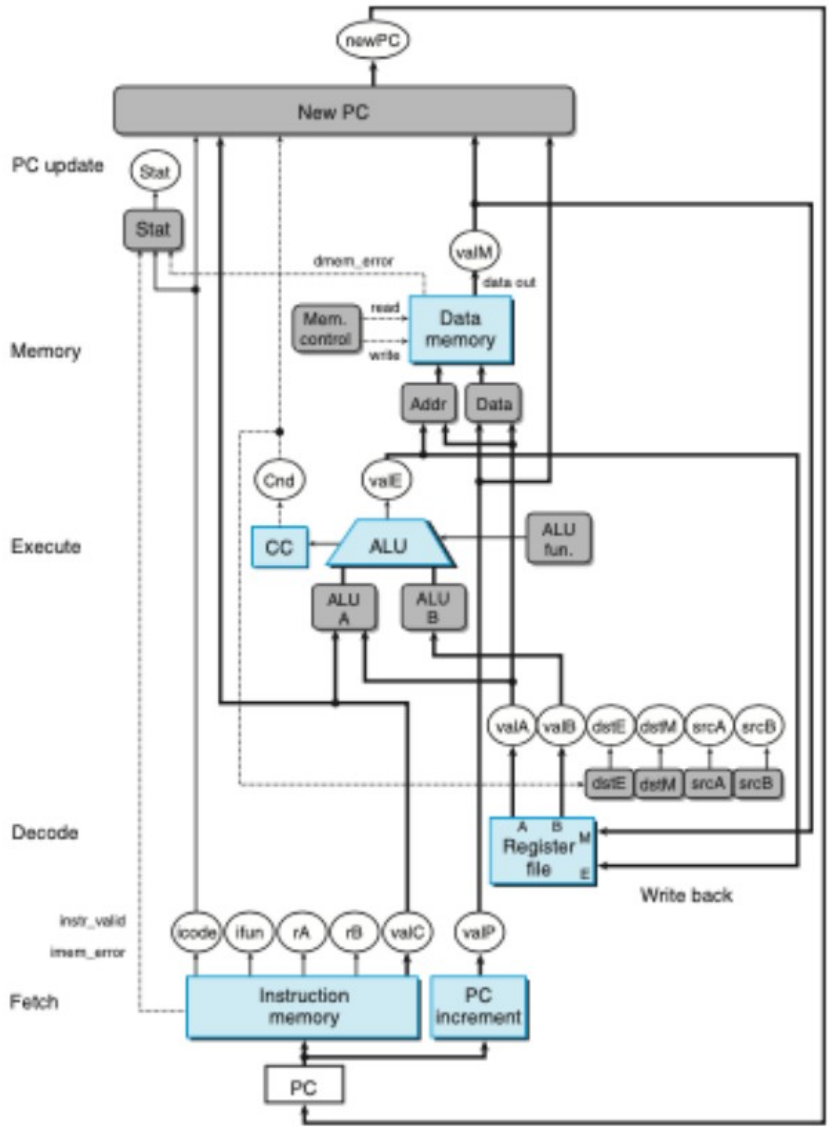
MEMORY STAGE:

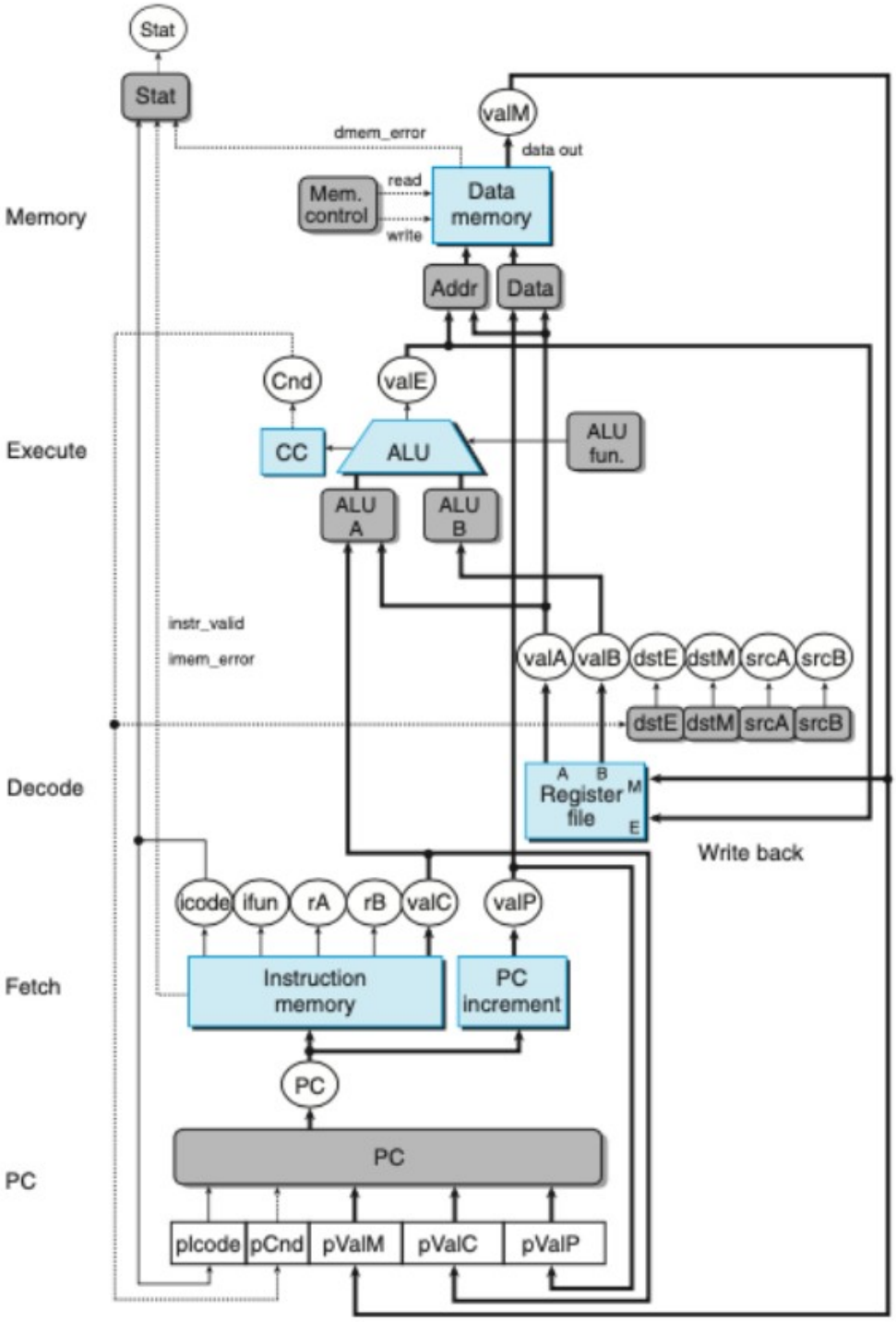


PC UPDATE STAGE:

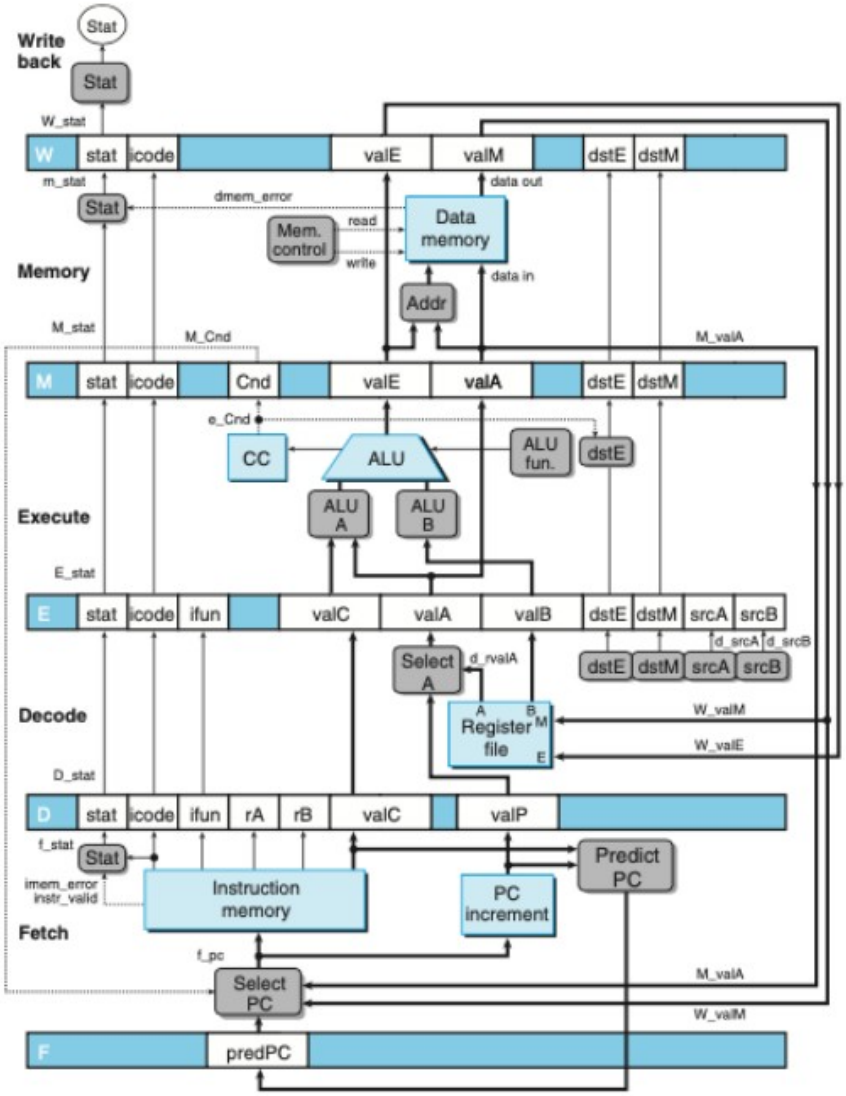


OVERALL STRUCTURE:





Pipelined processor hardware structure



Instructions supported

The supported instructions are

- halt
- nop
- cmovXX
- irmovq
- rmmovq

- OPq
- jXX
- call
- ret
- pushq
- popq

The processor supports all instructions in the Y86□64 instruction set.

Challenges faced in Pipeline:

The challenges faced in pipeline is that the hazards are most difficult.

As we need to check and any wrong prediction might give an error.

The probability of getting the error is great when compared to sequential if something goes wrong.

Pipeline Output to a example input:

Challenges Faced:

Among the difficulties I encountered are the following:

1) Transitioning from conceptualising the Y86-64 processor's functional level design to its physical circuit level required some effort.

2) It was also difficult to appropriately figure out and implement the control logic, particularly in the case of the pipelined CPU.