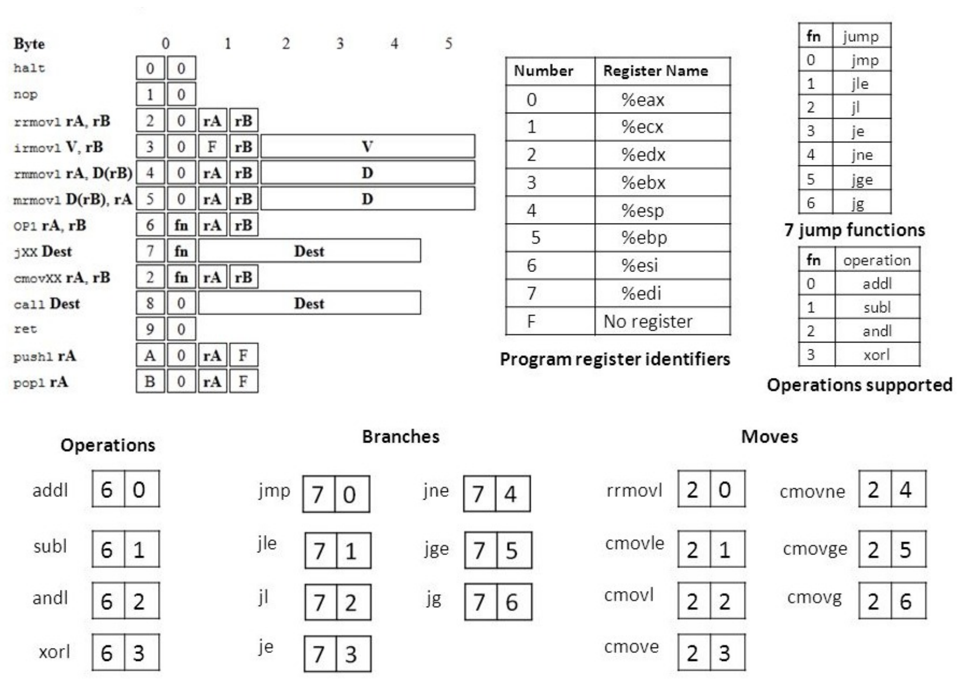
Intro to Processor Architecture: Y86 Processor Design

Koluguri Sri Rama Rathan Reddy
2022102072
koluguri.reddy@students.iiit.ac.in

Karthikeya Busam
2022102042
karthikeya.busam@students.iiit.in

February 2024



Byte	0	1	2	3	4	5
halt	0 0					
nop	1 0					
rrmovl rA, rB	2 0	rA rB				
irmovl V, rB	3 0	F rB	V			
rmmovl rA, D(rB)	4 0	rA rB	D			
mrmovl D(rB), rA	5 0	rA rB	D			
OPl rA, rB	6 fn	rA rB				
jXX Dest	7 fn	Dest				
cmovXX rA, rB	2 fn	rA rB				
call Dest	8 0	Dest				
ret	9 0					
pushl rA	A 0	rA F				
popl rA	B 0	rA F				

Number	Register Name
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
F	No register

Program register identifiers

fn	jump
0	jmp
1	jle
2	jl
3	je
4	jne
5	jge
6	jg

7 jump functions

fn	operation
0	addl
1	subl
2	andl
3	xorl

Operations supported

Operations

addl	6 0
subl	6 1
andl	6 2
xorl	6 3

Branches

jmp	7 0	jne	7 4
jle	7 1	jge	7 5
jl	7 2	jg	7 6
je	7 3		

Moves

rrmovl	2 0	cmovne	2 4
cmovle	2 1	cmovge	2 5
cmovl	2 2	cmovg	2 6
cmove	2 3		

Contents

1 Introduction

Processor architecture studies how processors are designed and implemented to execute instructions and perform computations. Processor architecture can be classified into two main types: sequential and pipe-lined. Sequential processors execute one instruction at a time, while pipe-lined processors divide the instruction execution into multiple stages and overlap the execution of different instructions. Pipe-lined processors can achieve higher performance and efficiency than sequential processors. Still, they also introduce challenges such as pipeline hazards, which occur when the execution of one instruction depends on the outcome of another instruction still in the pipeline.

In this project, we aim to design and implement a processor architecture based on the Y86 instruction set architecture (ISA), a simplified version of the X86 ISA. The Y86 ISA consists of nine instructions: halt, nop, rrmovq, irmovq, rmmovq, mrmovq, Opq, jXX, and cmovXX. The project has two main objectives:

- to develop a sequential processor design that executes all the Y86 instructions
- to develop a pipelined processor design that executes all the Y86 instructions, and supports features to eliminate pipeline hazards.

The project requires us to Implement the above tasks in Verilog, create test cases to verify the functionality and correctness of our processor design, and to write a report that describes the design details, the supported features, and the challenges encountered.

The project will help us to understand the fundamental concepts and principles of processor architecture, such as instruction encoding, instruction execution, pipeline stages, pipeline hazards, hazard detection, hazard resolution, and performance evaluation. The project will also enhance our skills in using Verilog, a hardware description language, to model and simulate processor designs.

2 Arithmetic Logic Unit

An Arithmetic Logic Unit performing 4 Operations on two 64-bit numbers is implemented in Verilog. The four operations are:

- Addition
- Subtraction
- And
- Xor

The operation is decoded based on the control signal input to the ALU and the respective blocks operate.

2.1 Control Signal

It is a two-bit number.

- $S = 00 \Rightarrow$ Addition is to be performed
- $S = 01 \Rightarrow$ Subtraction is to be performed
- $S = 10 \Rightarrow$ And is to be performed
- $S = 11 \Rightarrow$ Xor is to be performed

An if-else statement decodes the proper operation in the Verilog code.

2.2 Full Adder

It is used to add three single-bit numbers input to the block.

The diagram below explains the implementation of the full Adder:

2.3 Adder Subtractor Block

We can use 64 such Full Adders in series for a 64-bit full adder or 1 Full Adder can be used with registers to store inputs, and outputs and give back the output carry as input carry to the next stage. We use ”generate” a statement in Verilog to achieve this.

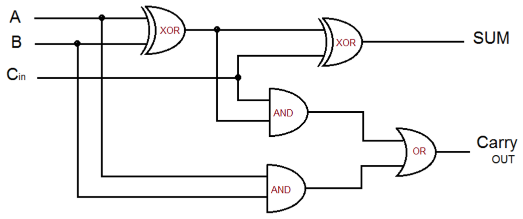


Figure 1: Full Adder Implementation using Gates

We can make a single block combining both adder and subtractor. In subtraction, we use 2's complement method. For this, the second input needs to be in 2's complement and we implement this by xoring it with one. Xoring with 0 gives us the same number. Therefore, we Xor the second input with 0 for addition and 1 for subtraction. In Verilog, we use the input "Mode" for decoding it to be addition or subtraction.

2.3.1 Addition

The below image gives us an idea of How addition is implemented:

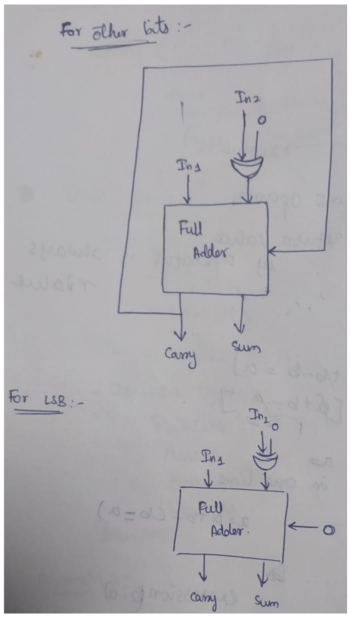


Figure 2: Adder implementation using full Adder

Here, $Mode = 0$

2.3.2 Subtraction

The below image gives us an idea of How subtraction is implemented:

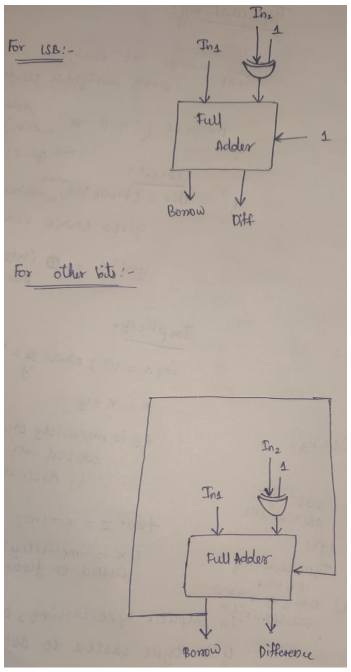


Figure 3: Subtractor implementation using full Adder

Here, $Mode = 1$

2.4 And Block

We store the inputs in registers and send the data to and gate and store the Output data in registers. Shift registers can be used in hardware implementation. The "generate" statement is used in Verilog implementation.

2.5 Xor Block

We store the inputs in registers, send the data to the xor gate, and store the Output data in registers. Shift registers can be used in hardware implementation. The "generate" statement is used in Verilog implementation.

3 Sequential Implementation

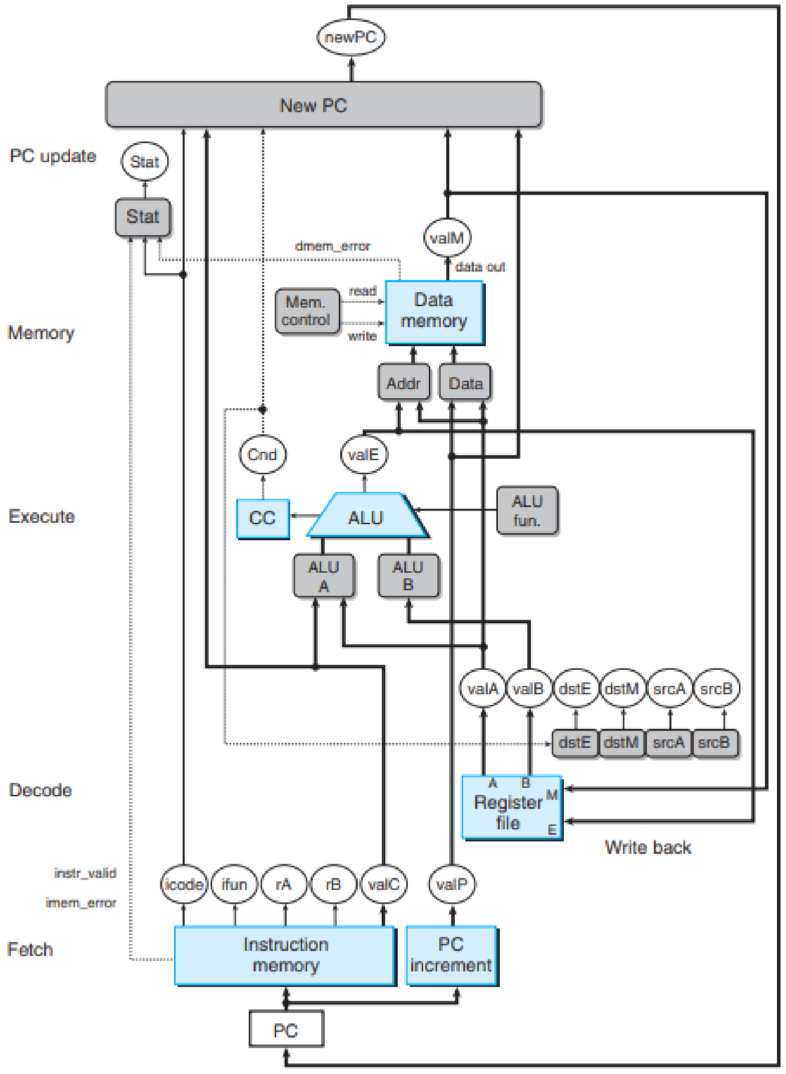


Figure 4: Y86-64 Processor implementation

The implementation of the Y86-64-bit processor in the Sequential Method has 6 stages:

- Fetch stage
- Decode Stage
- Execute Stage

- Memory
- Write Back Stage
- PC Update Stage

3.1 Fetch stage

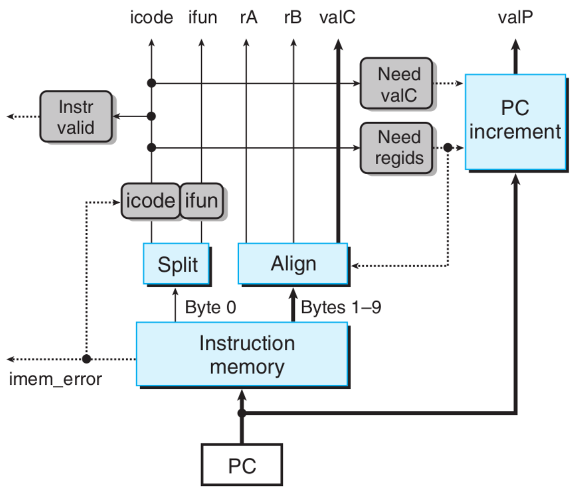


Figure 5: Fetch Stage implementation

We will be given an instruction of a maximum length of 10 bytes(80 bits), a clock signal, and the Program Counter's present value.

We need to get the instruction, separate it properly, and store it in different variables.

Byte 0: the first byte is divided into two parts of 4 bits each and stored in two variables namely icode and ifun.

The 12 operations that can be performed by the Y86 processor are :

- Halt
- No Operation
- rrmovq or cmovXX (Copying from one register to another)
- irmovq (Copying the input value into a register)
- rmmovq (Copy from Register to memory)
- mrmovq (Copy form Memory to Register)

- Opq (Arithmetic Operations)
- jXX (Conditional jump)
- call (Call a function)
- ret (Return to the function)
- pushq (Push a data into stack)
- popq (Pop a data from stack)

Byte 1: It gives the address of the data(stored in the form of register IDs) on which the operation has to be performed.
All the above operations do not require register IDs.

- Halt, noop, jXX, call, ret do not require any register id
- cmovXX, irmovq, rmmovq, mrmovq, opq, pushq, popq require 2 register ids.

If register id is required, Bytes 2 to 9 are for valC which is the address of the function to be called or the address to where the function must jump. // If register ID is not required, the bytes 1 to 8 are for valC.

- Only irmovq, mrmovq, mrmovq, jXX, call require valC.

last Byte: stored in variable valP. valP updates the program counter for the next instruction.
ValP = $p + 1 + r + 8i$ r = 1 if register IDs are required
i = 1 if valC is required.

3.2 Decode Stage and Write Back Stage

Both the stages access the same register files, one to read the inputs to the execute stage (Decode) and the other to write the outputs of the Execute stage(Writeback stage).

The register file has four ports(valA, valB, valE, valM), allowing for two simultaneous reads (via ports A and B) and two simultaneous writes (via ports E and M). Each port includes both an address connection, representing a register ID, and a data connection, comprising 64 wires serving as either output or input words for read and write operations respectively. The read ports are assigned address inputs srcA and srcB, while the write ports are assigned address inputs dstE and dstM. The special identifier 0xF on an address port indicates that no register should be accessed.

Register ID srcA and srcB indicate which register should be read to generate valA and valB respectively. Register ID dstE and dstM indicate the destination register for write port E and M respectively, where the computed value valE and read value valM are stored.

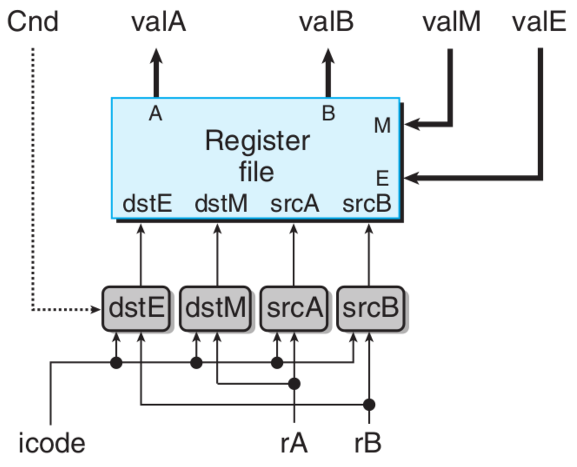


Figure 6: Decode Stage and Write Back Stage implementation

Possibilities for srcA are:

- rrmovq, irmovq, Opq, pushq ⇒ srcA = rA.
- rte, popq ⇒ srcA = rsp.

Possibilities for srcB are:

- rmmovq, mrmovq, Opq ⇒ srcB = rB.
- call, ret, pushq, popq ⇒ srcA = rsp.

Possibilities for dstE are:

- rrmovq, irmovq,Opq ⇒ dstE = rB.
- call, ret, pushq, popq ⇒ dstE = rsp.

Possibilities for dstM are:

- mrmovq, popq ⇒ dstM = rA.

3.3 Execute Stage

This Stage includes the ALU. The ALU can perform 4 operations ADD (aluA + aluB), SUBTRACT (aluA - aluB), AND (aluA & aluB) and XOR (aluA ⊕ aluB) on inputs aluA and aluB which depends on the instruction to be executed.

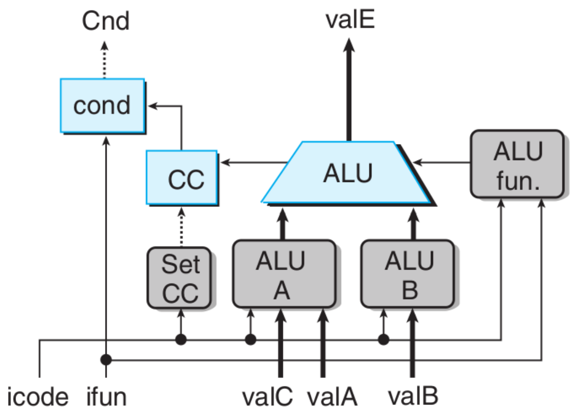


Figure 7: Execute Stage implementation

- for rrmovq, opq: aluA = valA
- for irmovq, rmmovq, mrmovq: aluA = valC
- for call and pushq: aluA = -8 (decrement stack pointer)
- for ret and popq: aluA = +8 (increment stack pointer)

ALU is mostly used as an adder in the execute stage. However, we can decode the operations to be performed in the Opq instruction are decoded based on the value of ifun.

The ALU also includes a condition codes register which is of 3 bits in length. We set these codes only in case of Opq instruction.

- first bit: Zero flag $\Rightarrow$ true when output is zero.
- second bit: Sign Flag $\Rightarrow$ true when output is negative
- third bit: Overflow Flag $\Rightarrow$ true when the output is 65 bits (more than 64 bits).

Also, the Condition signal is set in this stage which is used for conditional jump (jXX) and Conditional move (cmovXX) instructions.

3.4 Memory

It has the task of reading and writing the program data. It takes in valE, and valP as inputs to perform the write operation. valM is produced as the output of the read operation. Write operation can write either data or address into memory.

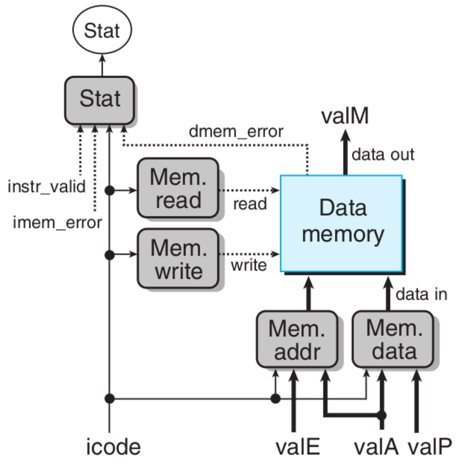


Figure 8: Memory Stage implementation

1. Inputs for writing address into memory:
   - for rmmovq, mrmovq, call, pushq ⇒ input = valE
   - for ret, popq ⇒ input = valA
2. Inputs for writing data into memory:
   - for rmmovq, pushq ⇒ input = valA
   - for call ⇒ input = valP
3. memory is read-only in case of mrmovq, popq, ret.

Memory also computes the status code resulting from the instruction according to the values of code, INS, ADR, and DatMemError flags.

3.5 PC Update Stage

The final stage of the processor is to update the program counter for the next program to be executed. The new PC will be valC, valM, or valP, depending on the instruction type and whether or not a branch should be taken.

- for call instruction, new PC = valC
- for a jump instruction
  - if the condition is satisfied, new PC = valC
  - if the condition is not satisfied, new PC = valP

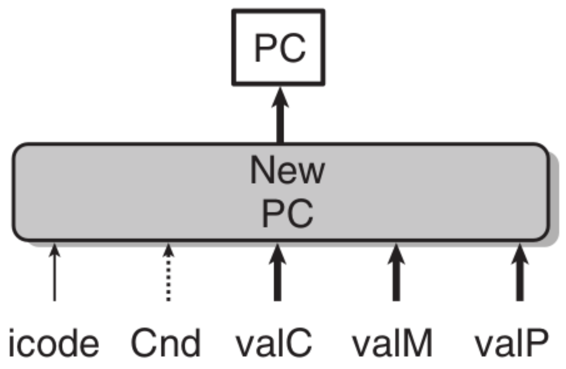


Figure 9: Program Counter Updating Stage implementation

- for ret instruction, new PC = valM
- for any other instruction, new PC = valP by default

3.6 Conclusion of Sequential Y86 Processor

In conclusion, we have designed a Y86-64 processor by organizing instruction execution steps into a streamlined process. While this approach minimizes hardware complexity and uses a single clock, known as SEQ, it suffers from slow processing speeds. The clock must operate slowly to allow signals to traverse all stages within a cycle.

However, this results in under-utilization of hardware units, as they remain active for only part of the cycle.

To address this issue, we will use the implementation of pipe lining, a technique that promises improved performance by overlapping instruction execution stages.

4 Pipeleined Implementation

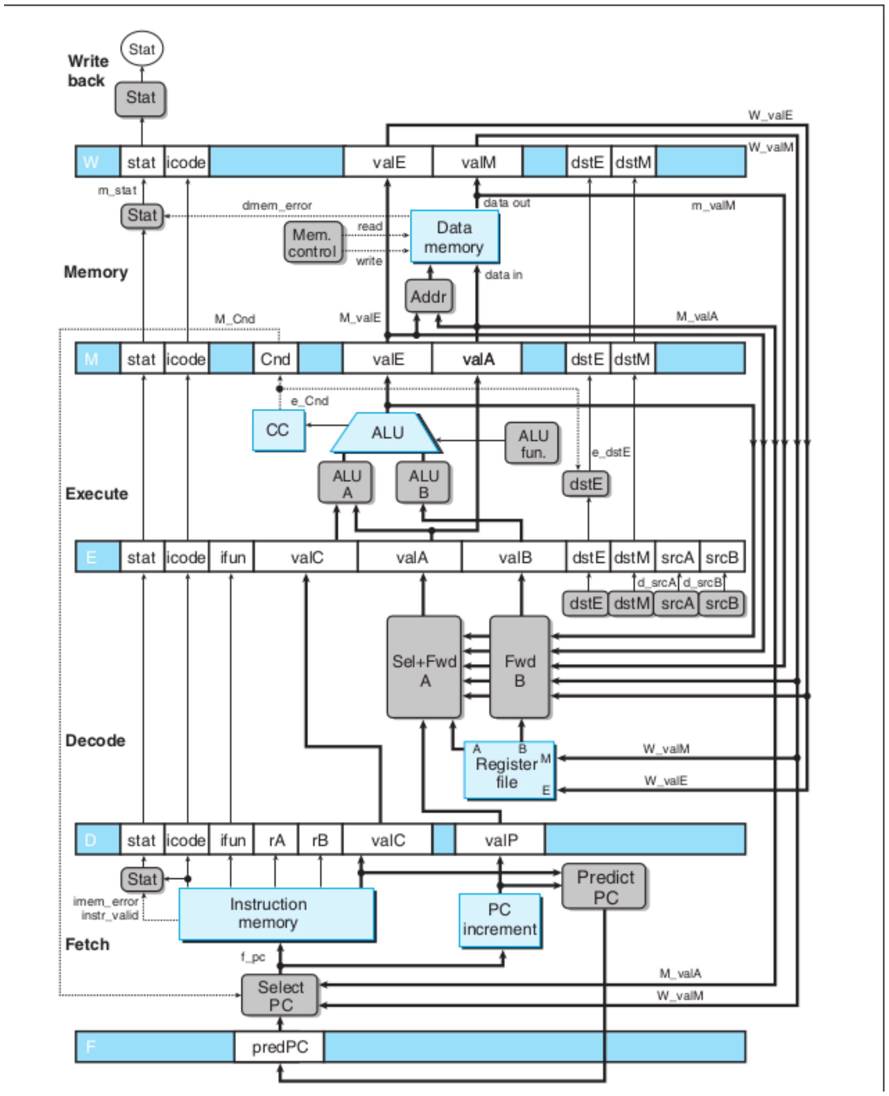


Figure 10: Complete Pipeline Implementation

We start the Pipelining implementation by modifying our sequential implementation. We club the Fetch and PC update stages and then insert the Pipelining Registers in between the stages.

The pipelining Registers are:

- F holds a predicted value of the program counter

- D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Another Register called "Stat" holds the status of each stage of the pipelined architecture.

4.1 Predicting next PC

In every clock cycle, a new instruction is being fetched into the Fetch stage. The instruction fetched depends on the PC value. so it is very important to update the PC to fetch the next instruction properly.

We get valP in the fetch stage itself which helps us find the next instruction. But there would be a problem in the case of conditional move, conditional jump, call, and ret functions.

- **For call:** We get the value of valC in the fetch stage itself so, we jump to that instruction and fetch it.
- **For unconditional jump and move:** The jump happens for sure so we update the new PC to valP.
- **For conditional jump and move:** We assume that the condition is satisfied and update the PC accordingly. If the condition is not satisfied, we use some hazard-handling techniques to ensure the correct processing of the instructions. This is called Branch prediction.
- **For ret instruction:** We wait until memory is read for the correct value of the PC.

4.2 PC Selection and Fetch Stage

This stage selects a current value for the Program counter and also predicts a PC value for the next clock cycle.

The PC selection logic within the pipeline architecture operates across three distinct program counter sources. In the event of a mispredicted branch entering the memory stage, the corresponding

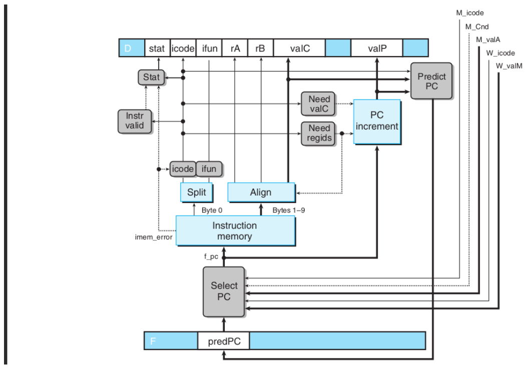


Figure 11: Implementation of Fetch with PC prediction

valP value, denoting the address of the subsequent instruction, is obtained from pipeline register M through the M_valA signal. During the write-back stage, when processing a "ret" instruction, the return address is retrieved from pipeline register W via the W_valM signal. In all remaining scenarios, the predicted PC value is utilized, residing in pipeline register F and accessed through the F_predPC signal. The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise.

In contrast to SEQ, in our current system, we need to divide the assessment of instruction status into two phases. During the fetch stage, we have the capability to identify memory errors caused by out-of-range instruction addresses, as well as detect illegal or halt instructions. However, the detection of an invalid data address is postponed until the memory stage.

4.3 Decode and Write-back Stage

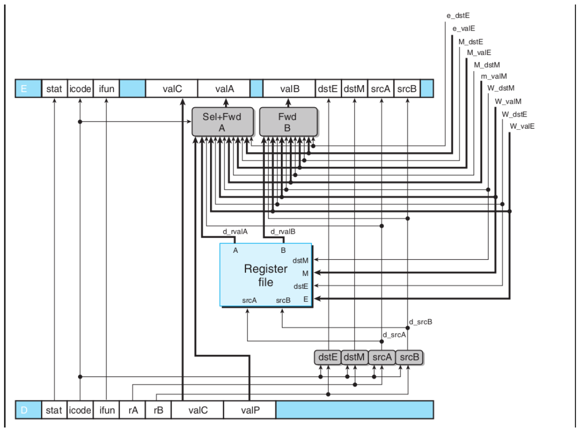


Figure 12: Implementations of Decode and Write-back stages

The components labeled dstE, dstM, srcA, and srcB closely resemble their equivalents in the SEQ implementation. Notably, the register IDs fed to the write ports originate from the write-back stage signals (W_dstE and W_dstM), rather than the decode stage. This ensures that writes target the registers specified by instructions in the write-back stage. The complexity of this stage primarily lies in the forwarding logic. The "Sel+Fwd A" block has a dual function: combining valP with valA to streamline pipeline register state and managing forwarding for source operand valA. The merging of valA and valP exploits the fact that only call and jump instructions require valP in later stages, and these instructions don't require the value from register file port A. The selection is dictated by the icode signal; when D_icode aligns with the instruction code for call or jXX, D_valP is chosen as the output. If none of the forwarding criteria are met, d_valA, the value read from register port A, is selected as the output.

The processor's overall status, denoted as Stat, is determined by a module that evaluates the status value found in pipeline register W. This status can signify regular operation (AOK) or one of three potential exception conditions. Given that pipeline register W stores the state of the most recently executed instruction, it is logical to derive the processor's overall status from this value. The exception to this rule occurs when there is a stall in the write-back stage, which is considered normal operation. In such cases, it is necessary for the status code to remain as AOK.

4.4 Execute Stage

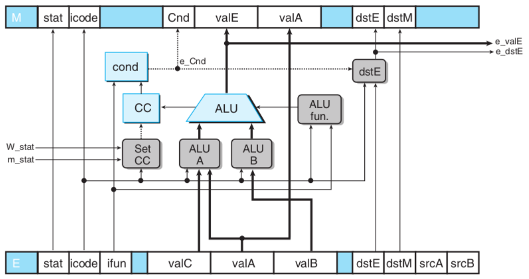


Figure 13: Implementation of Execute stage

The hardware components and logical blocks mirror those found in SEQ, even though with signal names adjusted accordingly. Notably, signals such as e_valE and e_dstE are routed to the decode stage, serving as a forwarding resource. An alteration lies in the "Set CC" logic, responsible for deciding the necessity of condition code updates, which now incorporates inputs from m_stat and W_stat signals. These inputs help identify instances where instructions causing exceptions progress through subsequent pipeline stages, prompting the suppression of condition code updates.

4.5 Memory Stage

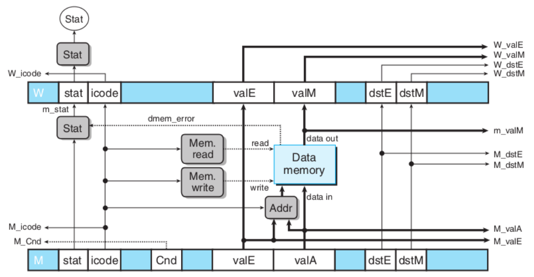


Figure 14: Implementation of Memory Stage

In contrast to the memory stage in the SEQ architecture, it is evident that the section labeled ”Memdata” in SEQ is absent in PIPE. Previously, this segment was responsible for choosing between data sources, namely valP (pertaining to call instructions) and valA. However, this selection function is now fulfilled by the component labeled ”Sel+Fwd A” within the decode stage. Most of the other components within this stage closely resemble their equivalents in SEQ, albeit with appropriate signal renaming. Additionally, the diagram illustrates that numerous values stored in pipeline registers M and W are disseminated to other sections of the circuit as part of the forwarding and pipeline control mechanisms.

4.6 Control Logic: Stalls, Bubbles and Exception Handling

This logic must handle the following four control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

1. **Load/use hazards** The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
2. **Processing ret** The pipeline must stall until the ret instruction reaches the write-back stage.
3. **Mispredicted branches** By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.
4. **Exceptions** When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

4.6.1 Stalling

The processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. Our processor can avoid data hazards by holding back an instruction in the decode stage until the instructions generating its source operands have passed through the write-back stage.

4.6.2 Bubble

Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline. But for cases of normally handling the instruction, we need to inject a bubble into the execute stage. A bubble is like a dynamically generated nop instruction—it does not cause any changes to the registers, the memory, the condition codes, or the program status.

This mechanism can be implemented fairly easily, but the resulting performance is not very good. There are numerous cases in which one instruction updates a register and a closely following instruction uses the same register. This will cause the pipeline to stall for up to three cycles, reducing the overall throughput significantly.

4.6.3 Forewarding

In our PIPE- design, we retrieve source operands from the register file during the decode stage. However, it's possible for one of these source registers to have a pending write operation in the write-back stage. Instead of halting operations until the write is finished, our approach involves directly transferring the value set to be written to the pipeline register E, thereby ensuring a continuous flow of source operands. This technique of passing a result value directly from one pipeline stage to an earlier one is commonly known as data forwarding (or simply forwarding, and sometimes bypassing).

4.6.4 Load/Use Hazards

One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. We can avoid a load/use data hazard with a combination of stalling and forwarding. This requires modifications of the control logic, but it can use existing bypass paths.

Employing a stall to manage a load/use hazard is referred to as a load interlock. When combined with forwarding, load interlocks effectively address all potential types of data hazards. Given that only load interlocks diminish pipeline throughput, we can almost attain our objective of issuing a new instruction on each clock cycle.

4.6.5 Control Hazards

Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. Control hazards can only occur in our pipelined processor for ret and jump instructions. Moreover, the latter case only causes difficulties when the direction of a conditional jump is mispredicted.

- **Misprediction Branch:**
  A mispredicted branch occurs when the processor wrongly predicts the outcome of a branch instruction. This error disrupts the smooth flow of instructions through the pipeline. To rectify this issue, the pipeline must be flushed, discarding instructions fetched after the mispredicted branch. The correct branch target is then recalculated based on the actual branch condition, and fetching resumes from the correct target address. Mispredicted branches introduce stalls in the pipeline, impacting overall performance. Efficient handling of mispredictions is essential for maintaining high throughput in pipelined architectures, often achieved through advanced branch prediction algorithms.
- **Processing ret:**
  A "ret" hazard arises when a "ret" instruction conflicts with earlier pipeline instructions due to data dependencies. To manage this hazard, The pipeline may stall to resolve dependencies between the "ret" instruction and earlier instructions. Forwarding techniques can be employed to pass data directly from the execution stage to earlier pipeline stages, reducing the need for stalling. Efficient management of "ret" hazards ensures correct program execution while optimizing pipeline throughput and performance.

4.6.6 Exception Handling

A variety of activities in a processor can lead to exceptional control flow, where the normal chain of program execution gets broken. Exceptions can be generated either internally, by the executing program, or externally, by some outside signal. Our instruction set architecture includes three different internally generated exceptions, caused by

1. a halt instruction
2. an instruction with an invalid combination of instruction and function code
3. an attempt to access an invalid address, either for instruction fetch or data read or write

To handle these exceptions, we propagate the fetched exception till the writeback stage and then finish the program so as that the previously fetched instructions i.e., the ones fetched before the exception.

4.6.7 Special Control cases

Condition	Trigger
Processing `ret`	IRET ∈ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Cnd
Exception	m_stat ∈ {SADR, SINS, SHLT} || W_stat ∈ {SADR, SINS, SHLT}

Figure 15: Detection conditions for pipeline control logic

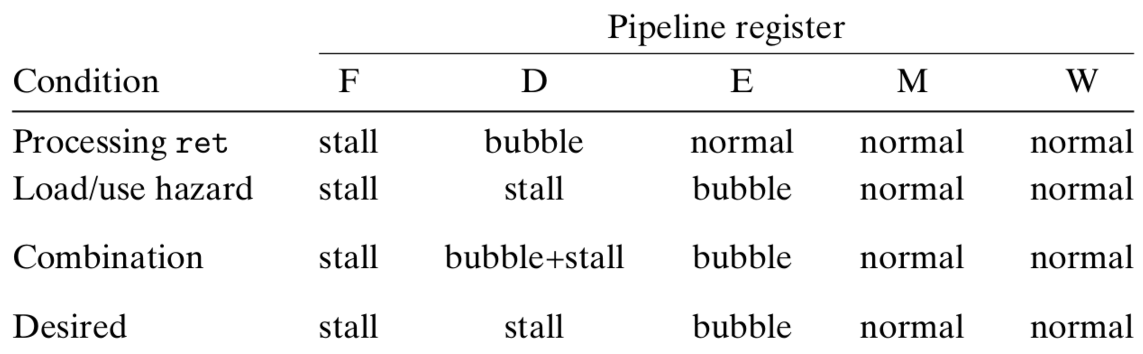
	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Figure 16: Actions for pipeline control logic

4.6.8 Combinations of Control Conditions

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

Figure 17: Combination of Misprediction Branch and ret at a time



	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

Figure 18: Combination of Load/Use hazard and Ret

4.7 Limitations

Pipelined Implementation increases the throughput of the processor but it has some limitations too.

4.7.1 Non Uniform Partitioning

The rate at which the clock changes is dependent on the slowest process in the processor. It's very hard to design a processor with all stages having approximately the same runtime. If the runtime for a particular stage is very high, it's a disadvantage.

4.7.2 Diminishing Returns of Deep Pipelining

We need to insert Pipeline Registers for every pipelined stage. If there are more number of pipelined stages, we need to insert more no. of pipe lined registers. Even the pipelined registers have some delay. So, this may cause the processor to to slow than the expected rate. if you double the no.of pipelined stages, the throughput will not be doubled due to this delay.