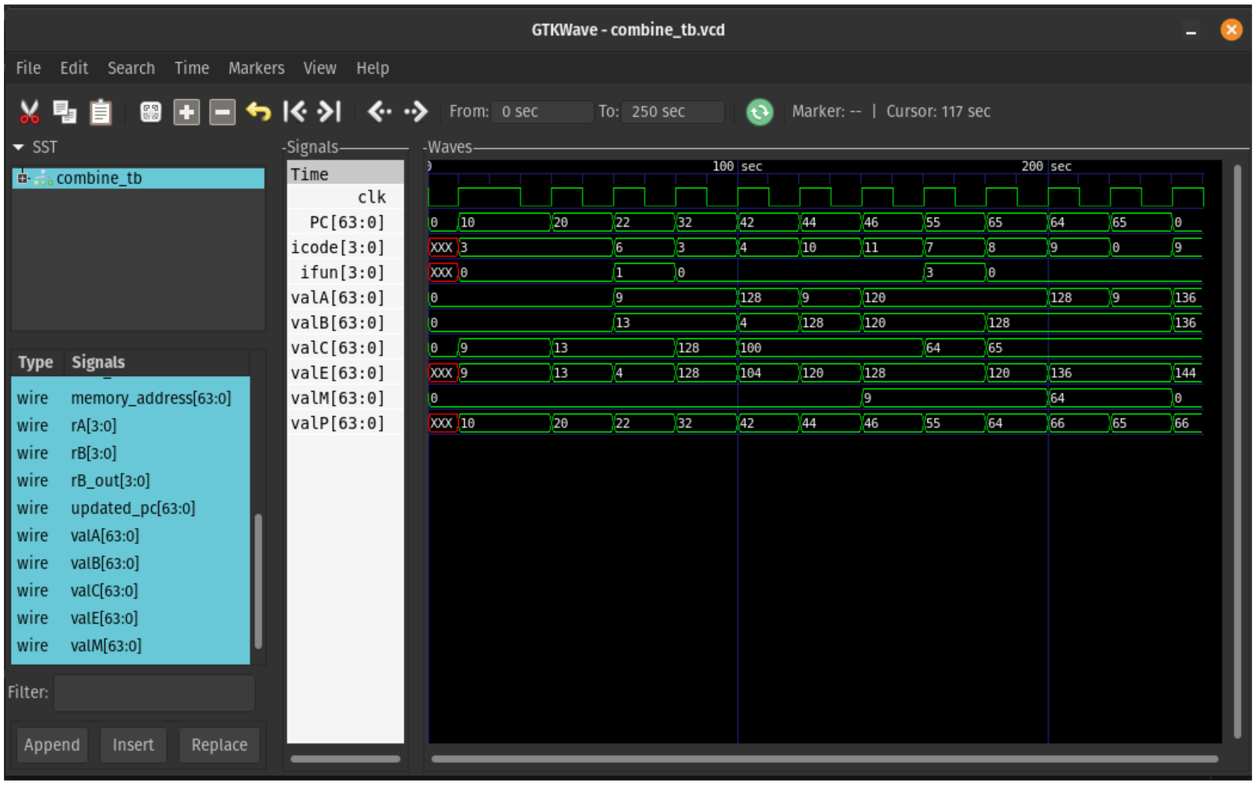
IPA Project Report

Sequential Implementation

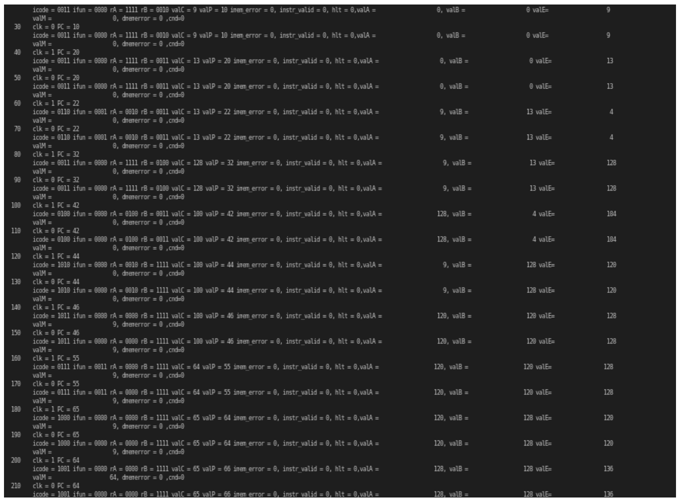
TestCase:
A test case with all instructions along with call return and jump.

```
    irmovq $9,  %rdx
    irmovq $21, %rbx
    subq %rdx, %rbx
    irmovq $128,%rsp
    rmmovq %rsp, 100(%rbx)
    pushq %rdx
    popq  %rax
    je done
    call proc
done:
    halt
proc:
    ret
```

Final_Output Of Processor GTKwave:



Output Terminal:



```
     icode = 0011 ifun = 0000 rA = 1111 rB = 0010 valC = 9 valP = 10 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    0, valB =                    0 valE=                    9
     valM =                    0, dmemerror = 0 ,cnd=0
 30  clk = 0 PC = 10
     icode = 0011 ifun = 0000 rA = 1111 rB = 0010 valC = 9 valP = 10 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    0, valB =                    0 valE=                    9
     valM =                    0, dmemerror = 0 ,cnd=0
 40  clk = 1 PC = 20
     icode = 0011 ifun = 0000 rA = 1111 rB = 0011 valC = 13 valP = 20 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    0, valB =                    0 valE=                   13
     valM =                    0, dmemerror = 0 ,cnd=0
 50  clk = 0 PC = 20
     icode = 0011 ifun = 0000 rA = 1111 rB = 0011 valC = 13 valP = 20 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    0, valB =                    0 valE=                   13
     valM =                    0, dmemerror = 0 ,cnd=0
 60  clk = 1 PC = 22
     icode = 0110 ifun = 0001 rA = 0010 rB = 0011 valC = 13 valP = 22 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                   13 valE=                    4
     valM =                    0, dmemerror = 0 ,cnd=0
 70  clk = 0 PC = 22
     icode = 0110 ifun = 0001 rA = 0010 rB = 0011 valC = 13 valP = 22 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                   13 valE=                    4
     valM =                    0, dmemerror = 0 ,cnd=0
 80  clk = 1 PC = 32
     icode = 0011 ifun = 0000 rA = 1111 rB = 0100 valC = 128 valP = 32 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                   13 valE=                  128
     valM =                    0, dmemerror = 0 ,cnd=0
 90  clk = 0 PC = 32
     icode = 0011 ifun = 0000 rA = 1111 rB = 0100 valC = 128 valP = 32 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                   13 valE=                  128
     valM =                    0, dmemerror = 0 ,cnd=0
100  clk = 1 PC = 42
     icode = 0100 ifun = 0000 rA = 0100 rB = 0011 valC = 100 valP = 42 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  128, valB =                    4 valE=                  104
     valM =                    0, dmemerror = 0 ,cnd=0
110  clk = 0 PC = 42
     icode = 0100 ifun = 0000 rA = 0100 rB = 0011 valC = 100 valP = 42 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  128, valB =                    4 valE=                  104
     valM =                    0, dmemerror = 0 ,cnd=0
120  clk = 1 PC = 44
     icode = 1010 ifun = 0000 rA = 0010 rB = 1111 valC = 100 valP = 44 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                  128 valE=                  120
     valM =                    0, dmemerror = 0 ,cnd=0
130  clk = 0 PC = 44
     icode = 1010 ifun = 0000 rA = 0010 rB = 1111 valC = 100 valP = 44 imem_error = 0, instr_valid = 0, hlt = 0,valA =                    9, valB =                  128 valE=                  120
     valM =                    0, dmemerror = 0 ,cnd=0
140  clk = 1 PC = 46
     icode = 1011 ifun = 0000 rA = 0000 rB = 1111 valC = 100 valP = 46 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  120 valE=                  128
     valM =                    9, dmemerror = 0 ,cnd=0
150  clk = 0 PC = 46
     icode = 1011 ifun = 0000 rA = 0000 rB = 1111 valC = 100 valP = 46 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  120 valE=                  128
     valM =                    9, dmemerror = 0 ,cnd=0
160  clk = 1 PC = 55
     icode = 0111 ifun = 0011 rA = 0000 rB = 1111 valC = 64 valP = 55 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  120 valE=                  128
     valM =                    9, dmemerror = 0 ,cnd=0
170  clk = 0 PC = 55
     icode = 0111 ifun = 0011 rA = 0000 rB = 1111 valC = 64 valP = 55 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  120 valE=                  128
     valM =                    9, dmemerror = 0 ,cnd=0
180  clk = 1 PC = 65
     icode = 1000 ifun = 0000 rA = 0000 rB = 1111 valC = 65 valP = 64 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  128 valE=                  120
     valM =                    9, dmemerror = 0 ,cnd=0
190  clk = 0 PC = 65
     icode = 1000 ifun = 0000 rA = 0000 rB = 1111 valC = 65 valP = 64 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  120, valB =                  128 valE=                  120
     valM =                    9, dmemerror = 0 ,cnd=0
200  clk = 1 PC = 64
     icode = 1001 ifun = 0000 rA = 0000 rB = 1111 valC = 65 valP = 66 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  128, valB =                  128 valE=                  136
     valM =                   64, dmemerror = 0 ,cnd=0
210  clk = 0 PC = 64
     icode = 1001 ifun = 0000 rA = 0000 rB = 1111 valC = 65 valP = 66 imem_error = 0, instr_valid = 0, hlt = 0,valA =                  128, valB =                  128 valE=                  136
```

Fetch stage :

The fetch stage reads 10 bytes from memory using the PC as the address of the first byte. This byte is split into two 4-bit quantities. The control logic computes instruction and function codes based on the values read from memory or nop if the address is invalid. Three 1-bit signals are computed: instr_valid (detects illegal instruction), need_regids (checks for register specifier byte), and need_valC (checks for constant word). instr_valid and imem_error generate the status code in the memory stage.

Decode stage :

Decode stage gives valA and valB from the register file with index of rA rB or %rsp which in our implementation is a text file.

Execute stage:

In the execute stage, the ALU calculates valE for Opq,memory addresses, or changes R[%rsp] val for push pop instructions. Condition codes also are calculated.. In cmovxx update destination registers. Similarly, for jump instructions, it sees to it that a jump should be taken or not.

Memory stage :

It writes or reads from memory based on valE and valM

PC_Update:

Updates PC to valM or valP based on the icode and condition codes

Single-Cycle Execution(Sequential processor timing Details)

1. **Clock Cycle**: Executes an entire instruction.
2. **Combinational Logic**: Independent of the clock, propagates values when inputs change.
3. **Control Elements:** PC, conditional control register, data memory, and register file update in a single cycle.
4. **PC Update:** New instruction loaded into PC at the positive edge of the clock.

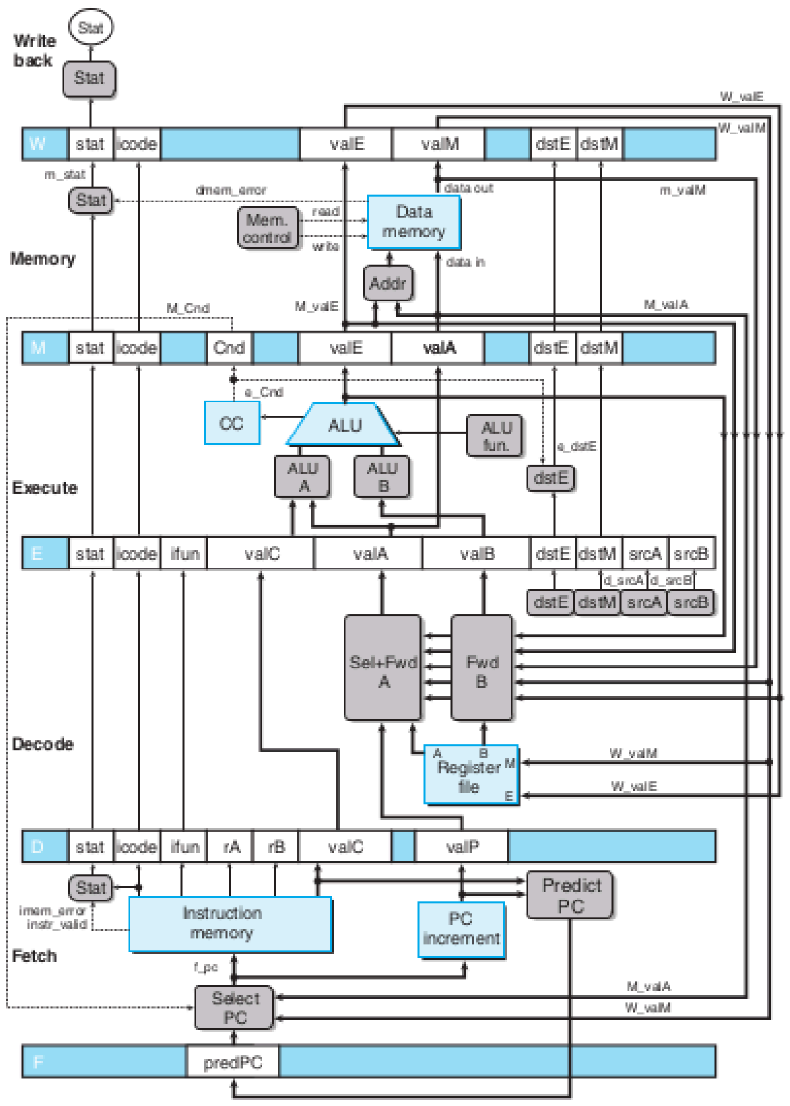
5. **Conditional Control**: Loaded only during integer operations.
6. **Data Memory Write:** Occurs for rmmovq, pushq, and call instructions.
7. **Status Updates:** Calculated every cycle but updated only at the positive clock edge.

Problems Faced

Bro this is not what it is

1. For the given sample testcase we got a random value for valC so we had to change the reading of instruction from 0,1,2,..9 to 0,1,9,8…2 .
2. Initially we put always @(*) for everything we realized it wont work unless writeback is posedge, execute is always, fetch is posedge

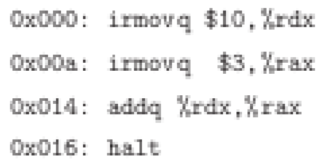
Pipeline Architecture



Pipeline architecture for handling data hazards

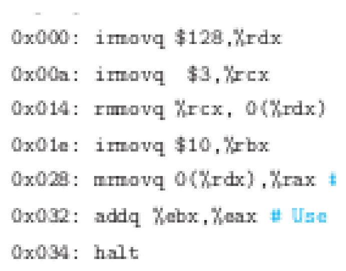
TestCases:

1)data forwarding



```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

2) load/data hazard



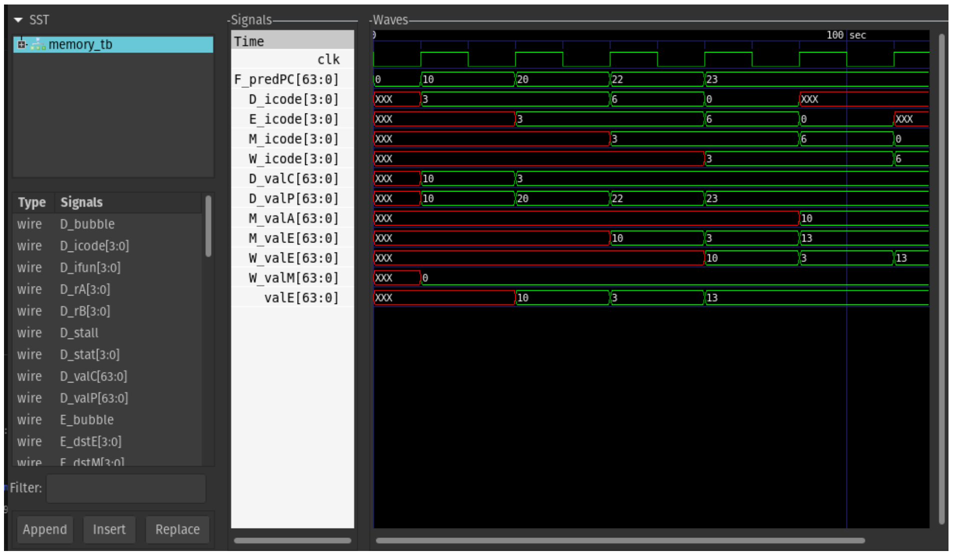
```
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax #
0x032: addq %ebx,%eax # Use
0x034: halt
```

3)call,ret,jmp,data forwarding, load/data hazard

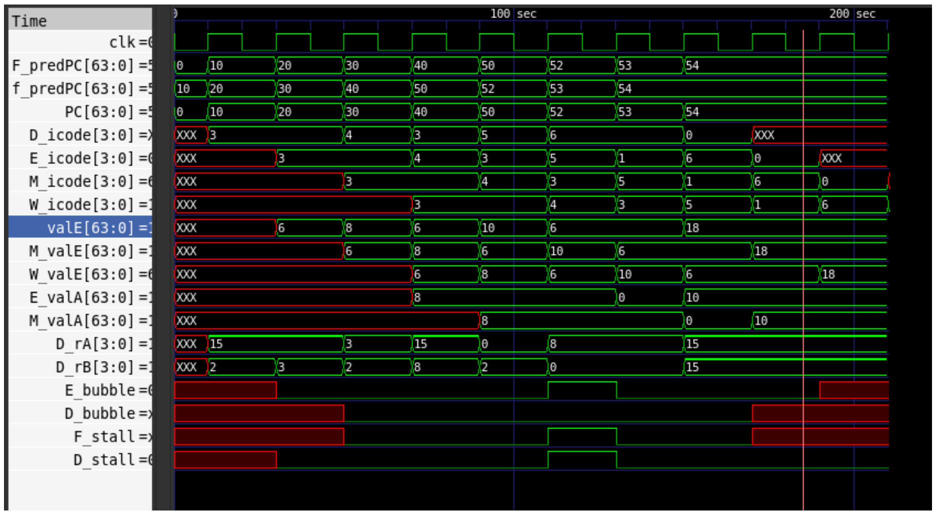
```
    irmovq $9,  %rdx
    irmovq $21, %rbx
    subq %rdx, %rbx
    irmovq $128,%rsp
    rmmovq %rsp, 100(%rbx)
    pushq %rdx
    popq  %rax
    je done
    call proc
done:
    halt
proc:
    ret
```

GTKwave Plot:

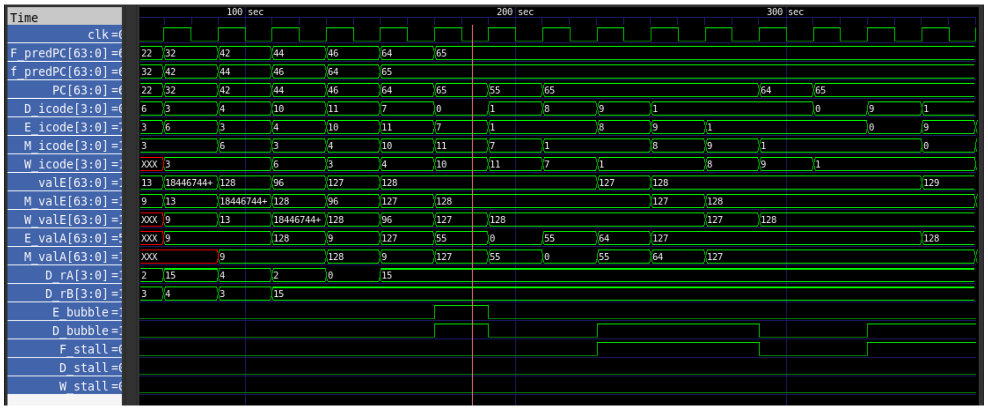
1)



2)



3)



Fetch

1. **Program Counter (PC) Selection**:
    - Three sources: Mispredicted branch (valP from pipeline register M), ret instruction (W_valM), or predicted value (F_predPC).
    - For call/jump instructions, valC is chosen; otherwise, valP is used.
2. **Logic Blocks**:
    - Need rA and rB and valC (same as SEQ) computes f_valP.
3. **Instruction Status Computation**:

- In fetch stage:
    - Test for memory error (out-of-range instruction address).
    - Detect illegal instructions or halt instructions.
- Invalid data address detection deferred to memory stage.

Decode and Writeback

1. **Register IDs for Write Ports**:
    - Unlike SEQ, where regIDs come from the decode stage, in PIPE, they come from the write-back stage (W_dstE and W_dstM). This ensures writes occur to the correct destination registers specified by the instruction in the write-back stage.
2. **Sel+Fwd A Block**:
    - The main difference between seq and pipeline except the pipeline logic is the data forwarding in the decode stage which introduces
    - Merges valP into valA to reduce pipeline register state.
    - Exploits that only call and jump instructions need valP in later stages
    - Five forwarding sources with data words and destination register IDs.
3. **Priority of Forwarding Sources**:
    - Priority: Execute stage, memory stage, and finally, write-back stage.

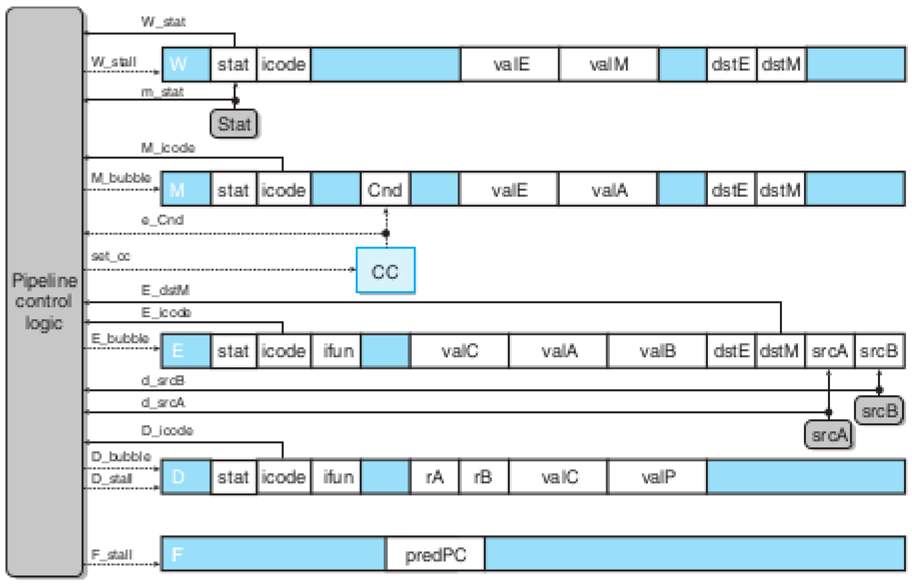
Execute

- Identical to SEQ.
- e_valE and e_dstE are there for data forwarding
- Setcc logic sets condition codes

Memory

Does a similar function as seq but does data forwarding now.

Pipeline control mechanism



The pipeline control mechanism in a processor manages flow through the pipeline to ensure correctness of output and avoid hazards. This handles data hazards, control hazards, and structural hazards to take care of the functioning of the processor.

1. Instruction Flow Control: Ensuring that instructions progress through the pipeline stages in the correct order without conflicts.

2. Hazard Detection and Resolution: Identifying and resolving hazards that can arise due to dependencies between instructions, such as data hazards (e.g., read-after-write) and control hazards (e.g., branches).

3. Stall and Bubble Insertion: Introducing stalls (delays) or bubbles (no-operation cycles) in the pipeline to resolve hazards and maintain data consistency.

4. Branch Prediction: Predicting the outcome of branch instructions to minimize pipeline stalls in case of mispredictions.

5. Data Forwarding: Routing data directly from one pipeline stage to another to prevent stalls caused by data dependencies.

6. Control Signals Generation: Generating control signals to coordinate the activities of different pipeline stages and ensure proper instruction execution.

Pipeline control mechanism optimizes instruction throughput by reducing idle cycles, and maximizing the processor's performance as we have seen in the SEQ isn't as good with a lot of idle cycles.

Challenges Faces

1. The main problem was when we were data forwarding, we didn't assign any values at the end instead output the values directly when calculated so the problem came that stuff was getting updated in the previous cycles itself.
2. We forgot to initialize a few variables like cnd and cc in the execute so it was in don't care .That took a while to find out
3. d_srcB should be d_rB and e_dstE should be hf we interchanged that by mistake and had a hard time debugging.