Day 100 of 100 Days RTL Challenge

Today's topic: Pipelined MIPS-32 Processor Design

Introduction:

Designing a Pipelined MIPS-32 Processor in Verilog!

Modern processors rely on pipelining to achieve high-speed execution by overlapping instruction execution across multiple stages. In this implementation, I've designed a 5-stage pipelined MIPS-32 processor using Verilog with a two-phase clocking mechanism for efficient instruction processing.

This design follows a structured fetch-decode-execute-memory-write-back (IF-ID-EX-MEM-WB) pipeline, handling ALU operations, memory access, and branching while minimizing execution stalls. The architecture includes:

1. 32-bit instruction memory and register file
2. Support for arithmetic, logic, load/store, and branch instructions
3. Branch handling mechanism for BEQZ & BNEQZ
4. Multi-phase clocking for better resource utilization

Let's dive into how this processor works and its impact on performance! 💡

Key Features:

1. **Two-Phase Clocking (clk1 and clk2):**
    - The design operates on two non-overlapping clock phases.

- Odd stages execute on clk1, and even stages execute on clk2.

2. **Pipeline Stages:**
   - Instruction Fetch (IF)
   - Instruction Decode (ID)
   - Execute (EX)
   - Memory Access (MEM)
   - Write-Back (WB)
3. **Register and Memory:**
   - Reg[0:31]: 32 general-purpose registers.
   - Mem[0:1023]: 1KB memory for instructions and data.
4. **Instruction Set (Opcode-Based):**
   - **R-type ALU operations:** ADD, SUB, AND, OR, SLT, MUL
   - **Immediate ALU operations:** ADDI, SUBI, SLTI
   - **Memory operations:** LW (load word), SW (store word)
   - **Branch operations:** BEQZ (branch if zero), BNEQZ (branch if not zero)
   - **Halt (HLT)** for stopping execution.

Pipeline Stage Breakdown:

1. Instruction Fetch (IF)

- Fetches the instruction from Mem[PC].
- If a branch is taken, updates PC accordingly.

- Stores the fetched instruction in IF_ID_IR.
- Stores the next PC value in IF_ID_NPC.

2. Instruction Decode (ID)

- Reads registers based on instruction fields.
- Sign-extends the immediate value for immediate instructions.
- Determines the type of instruction (ALU, LOAD, STORE, BRANCH, HALT).
- Stores values in pipeline registers.

3. Execution (EX)

- **ALU operations** perform arithmetic/logical computations.
- **Immediate ALU instructions** compute with immediate values.
- **Load/Store instructions** calculate memory address.
- **Branching:** Evaluates branch condition.

4. Memory Access (MEM)

- **Loads:** Reads memory content into MEM_WB_LMD.
- **Stores:** Writes values to memory if no branch is taken.

5. Write-Back (WB)

- Writes results back to registers for ALU and LOAD instructions.
- If HLT is encountered, stops execution.

Verilog Code :

```
module pipe_mips32(
// generating the two phase clock
input clk1,
input clk2
);
// Internal Registers
reg [31:0] PC, IF_ID_IR, IF_ID_NPC;
reg [31:0] ID_EX_IR, ID_EX_NPC, ID_EX_A, ID_EX_B, ID_EX_Imm;
reg [2:0] ID_EX_type, EX_MEM_type, MEM_WB_type;
reg [31:0] EX_MEM_IR, EX_MEM_ALUOUT, EX_MEM_B;
reg EX_MEM_cond;
reg [31:0] MEM_WB_IR, MEM_WB_ALUOUT, MEM_WB_LMD;
reg [31:0] Reg [0:31];
reg [31:0] Mem [0:1023];

parameter ADD = 6'b000000,
          SUB = 6'b000001,
          AND = 6'b000010,
          OR  = 6'b000011,
          SLT = 6'b000100,
          MUL = 6'b000101,
          HLT = 6'b111111,
          LW  = 6'B001000,
          SW  = 6'b001001,
          ADDI= 6'b001010,
          SUBI= 6'b001011,
          SLTI= 6'b001100,
          BNEQZ=6'b001101,
          BEQZ =6'b001110;
```

```
parameter RR_ALU = 3'b000, RM_ALU = 3'b001,
        LOAD = 3'b010, STORE = 3'b011,
        BRANCH = 3'b100, HALT = 3'B101;
reg HALTED;
reg TAKEN_BRANCH;
//INFORMATION FETCHING STAGE (IF Stage)
always @ (posedge clk1)
if (HALTED == 0)
begin
if (((EX_MEM_IR[31:26] == BEQZ) && (EX_MEM_cond == 1)) || ((EX_MEM_IR[31:26]
== BNEQZ) && (EX_MEM_cond == 0)))
begin
IF_ID_IR <= #2 Mem[EX_MEM_ALUOUT];
TAKEN_BRANCH <= #2 1'b1;
IF_ID_NPC <= #2 EX_MEM_ALUOUT + 1;
PC <= #2 EX_MEM_ALUOUT + 1;
end
else begin
IF_ID_IR <= #2 Mem[PC];
IF_ID_NPC <= #2 PC+1;
PC <= #2 PC+1;
end
end
//INFORMATION DECODE STAGE (ID Stage)
always @ (posedge clk2)
if (HALTED == 0)
begin
if (IF_ID_IR[25:21] == 5'B00000)
ID_EX_A <= 0;
else
```

```
ID_EX_A <= #2 Reg[IF_ID_IR[25:21]];
if (IF_ID_IR[20:16] == 5'b00000)
ID_EX_B <= 0;
else
ID_EX_B <= #2 Reg[IF_ID_IR[20:16]];
ID_EX_NPC <= #2 IF_ID_NPC;
ID_EX_IR <= #2 IF_ID_IR;
ID_EX_Imm <= #2 {{16{IF_ID_IR[15]}},{IF_ID_IR[15:0]}};
case (IF_ID_IR[31:26])
ADD,SUB,AND,OR,SLT,MUL : ID_EX_type <= #2 RR_ALU;
ADDI,SUBI,SLTI : ID_EX_type <= #2 RM_ALU;
LW    : ID_EX_type <= #2 LOAD;
SW    : ID_EX_type <= #2 STORE;
BNEQZ,BEQZ : ID_EX_type <= #2 BRANCH;
HLT : ID_EX_type <= #2 HALT;
default  : ID_EX_type <= #2 HALT;
endcase
end
//EXECUTION STAGE (EX Stage)
always @ (posedge clk1)
if(HALTED == 0)
begin
EX_MEM_type <= #2 ID_EX_type;
EX_MEM_IR <= #2 ID_EX_IR;
TAKEN_BRANCH <= #20;
case (ID_EX_type)
RR_ALU : begin
case (ID_EX_IR[31:26])
ADD : EX_MEM_ALUOUT <= #2 ID_EX_A + ID_EX_B;
SUB : EX_MEM_ALUOUT <= #2 ID_EX_A - ID_EX_B;
```

```
AND : EX_MEM_ALUOUT <= #2 ID_EX_A & ID_EX_B;
OR : EX_MEM_ALUOUT <= #2 ID_EX_A | ID_EX_B;
SLT : EX_MEM_ALUOUT <= #2 ID_EX_A < ID_EX_B;
MUL : EX_MEM_ALUOUT <= #2 ID_EX_A * ID_EX_B;
default : EX_MEM_ALUOUT <= #2 32'hxxxxxxxx;
endcase
end
RM_ALU : begin
case (ID_EX_IR[31:26])
ADDI : EX_MEM_ALUOUT <= #2 ID_EX_A + ID_EX_Imm;
SUBI : EX_MEM_ALUOUT <= #2 ID_EX_A - ID_EX_Imm;
SLTI : EX_MEM_ALUOUT <= #2 ID_EX_A < ID_EX_Imm;
default : EX_MEM_ALUOUT <= #2 32'hxxxxxxxx;
endcase
end
LOAD,STORE : begin
EX_MEM_ALUOUT <= #2 ID_EX_A + ID_EX_Imm;
EX_MEM_B <= #2 ID_EX_B;
end
BRANCH : begin
EX_MEM_ALUOUT <= #2 ID_EX_NPC + ID_EX_Imm;
EX_MEM_cond <= #2 (ID_EX_A == 0);
end
endcase
end
//MEMORY STAGE (MEM Stage)
always @ (posedge clk2)
if (HALTED == 0)
begin
MEM_WB_type <= #2 EX_MEM_type;
```

```
MEM_WB_IR <= #2 EX_MEM_IR;
case(EX_MEM_type)
RR_ALU,RM_ALU : MEM_WB_ALUOUT <= #2 EX_MEM_ALUOUT;
LOAD : MEM_WB_LMD <= #2 Mem[EX_MEM_ALUOUT];
STORE : if (TAKEN_BRANCH == 0)
Mem[EX_MEM_ALUOUT] <= #2 EX_MEM_B;
endcase
end
//WRITE BACK STAGE (WB Stage)
always @ (posedge clk1)
begin
if (TAKEN_BRANCH == 0)
case(MEM_WB_type)
RR_ALU : Reg[MEM_WB_IR[15:11]] <= #2 MEM_WB_ALUOUT;
RM_ALU : Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_ALUOUT;
LOAD : Reg[MEM_WB_IR[20:16]] <= #2 MEM_WB_LMD;
HALT : HALTED <= #2 1'B1;
endcase
end
endmodule
```

TESTBENCH :

EXAMPLE 1:

• ADD three numbers 10,20,30 stored in processor registers.

→ The Steps :

• Initialize register R1 with 10

• Initialize register R2 with 20

• Initialize register R3 with 30

ADD the three numbers and store the sum in R4,R5

ASSEMBLY LANGUAGE PROGRAM:

```
ADDI  R1,R0,10
ADDI  R2,R0,20
ADDI  R3,R0,25
ADD   R4,R1,R2
ADD   R5,R4,R3
HLT
```

TESTBENCH CODE:

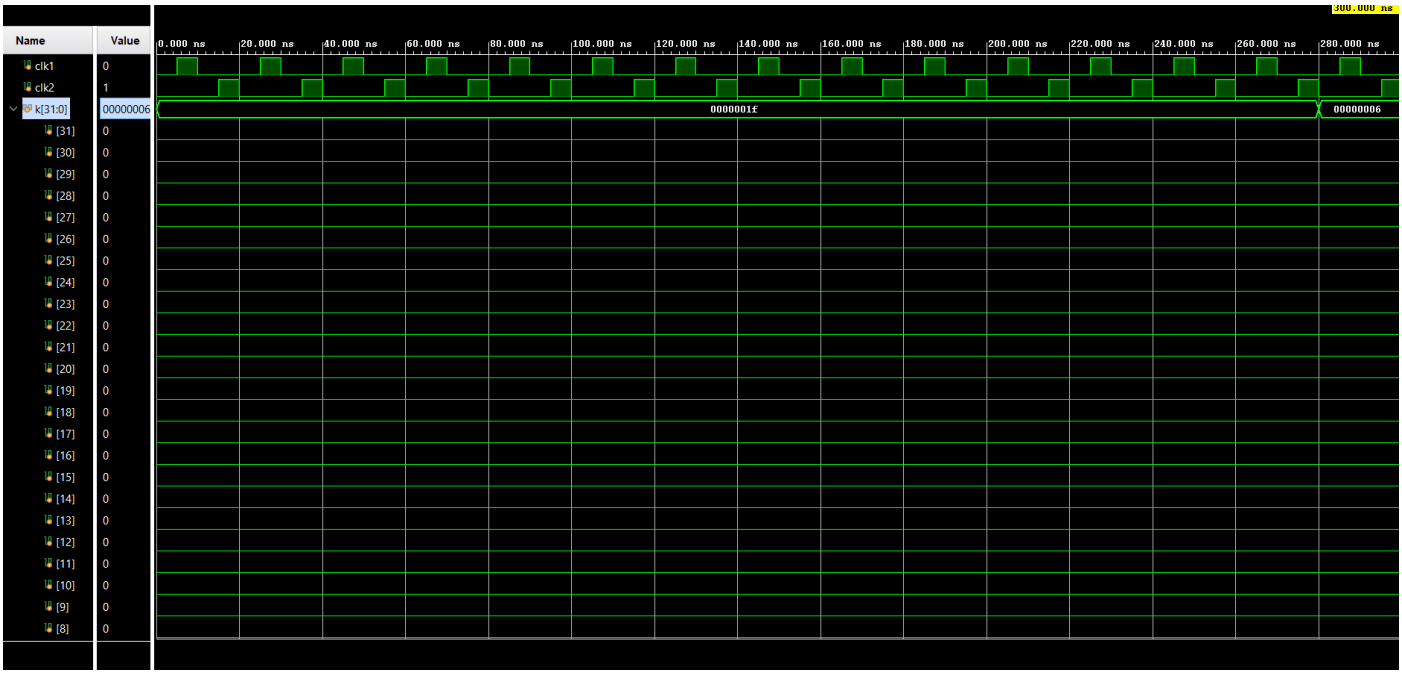
```
22
23      module test_mips32();
24
25      reg clk1,clk2;
26
27      integer k;
28
29      pipe_mips32 DUT (clk1,clk2);
30
31      initial
32      begin
33
34      clk1=0; clk2=0;
35
36      repeat(20)
37
38      begin
39
40      #5 clk1=1; #5 clk1=0;
41
42      #5 clk2=1; #5 clk2=0;
43
44      end
45      end
46
47      initial
48
49      begin
50
51      $display("MIPS32 Processor Test Case - 1");
52
53      for (k=0 ; k<31 ; k=k+1)
54
55      DUT.Reg [k] = k;
56
57      DUT.Mem[0] = 32'h2801000a;
58
59      DUT.Mem[1] = 32'h28020014;
60
61      DUT.Mem[2] = 32'h28030019;
```

```
61      DUT.Mem[2] = 32'h28030019;
62
63      DUT.Mem[3] = 32'h0ce77800;
64
65      DUT.Mem[4] = 32'h0ce77800;
66
67      DUT.Mem[5] = 32'h00222000;
68
69      DUT.Mem[6] = 32'h0ce77800;
70
71      DUT.Mem[7] = 32'h00832800;
72
73      DUT.Mem[8] = 32'hfc000000;
74
75      DUT.HALTED = 0;
76
77      DUT.PC = 0;
78
79      DUT.TAKEN_BRANCH = 0;
80
81      #280
82
83      for (k=0 ; k<6 ; k=k+1)
84
85
86
87      $display("R%1d - %2d",k,DUT.Reg[k]);
88
89      end
90
91      initial begin
92
93      $dumpfile("DUT.vcd");
94      $dumpvars(0,test_mips32);
95
96      #300 $finish;
97
98      end
99
100     endmodule
```

OUTPUT MESSAGE:

```
MIPS32 Processor Test Case - 1
R0 -  0
R1 - 10
R2 - 20
R3 - 25
R4 - 30
R5 - 55
```

SIMULATION OUTPUT:



TESTBENCH :

EXAMPLE 2:

• Load a word stored in memory location 120,add 45 to it, and Store the result in memory location 121.

→ **The Steps:**

• Initialize register R1 with the memory address 120.

• Load the contents of memory location 120 into register R2.

• ADD 45 to register R2.

• Store the result in memory location 121.

ASSEMBLY LANGUAGE PROGRAM:

ADDI R1,R0,120

LW R2,0(R1)

ADDI R2,R2,45

SW R2,1(R1)

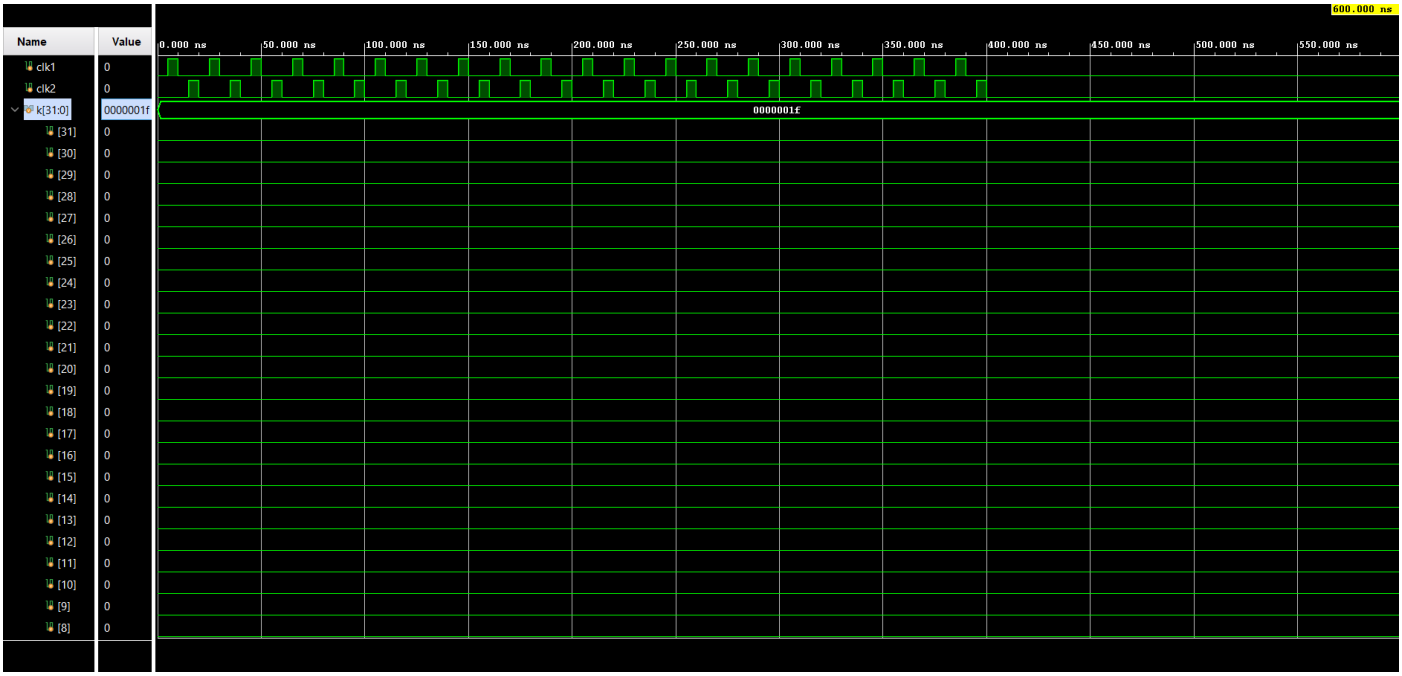
HLT

TESTBENCH CODE:

```
module test_mips32();

reg clk1,clk2;

integer k;

pipe_mips32 DUT (clk1,clk2); //Instantiating the module

initial
begin

clk1=0; clk2=0; //Two phase clock

repeat(20)

begin

#5 clk1=1; #5 clk1=0;

#5 clk2=1; #5 clk2=0;

end
end

initial

begin

$display("MIPS32 Processor Test Case - 2");

for (k=0 ; k<31 ; k=k+1)

DUT.Reg [k] = k;

DUT.Mem[0] = 32'h28010078; //ADDI R1,R0,120

DUT.Mem[1] = 32'h0c631800; //OR R3,R3,R3 (Dummy instruction)

DUT.Mem[2] = 32'h20220000; //LW R2,0(R1)
```

```
DUT.Reg [k] = k;

DUT.Mem[0] = 32'h28010078; //ADDI R1,R0,120

DUT.Mem[1] = 32'h0c631800; //OR R3,R3,R3 (Dummy instruction)

DUT.Mem[2] = 32'h20220000; //LW R2,0(R1)

DUT.Mem[3] = 32'h0c631800; //OR R3,R3,R3 (Dummy instruction)

DUT.Mem[4] = 32'h2842002d; //ADDI R2,R2,45

DUT.Mem[5] = 32'h0c631800; //OR R3,R3,R3 (Dummy instruction)

DUT.Mem[6] = 32'h24220001; //SW R2,1,(R1)

DUT.Mem[7] = 32'hfc000000; //HLT

DUT.Mem[120] = 85;

DUT.PC = 0;

DUT.HALTED = 0;

DUT.TAKEN_BRANCH = 0;

#500 $display("Mem[120]: %4d \nMem[121]: %4d",DUT.Mem[120],DUT.Mem[121]);

end

initial begin

$dumpfile("DUT.vcd");
$dumpvars(0,test_mips32);

#600 $finish;

end

endmodule
```

SIMULATION OUTPUT:



OUTPUT MESSAGE:

```
MIPS32 Processor Test Case - 2
Mem[120]:    85
Mem[121]:   130
```

TESTBENCH:

EXAMPLE 3:

• Compute the factorial of a number N stored in memory location 198.

→ **The Steps:**

• Initialize register R10 with the memory address 200.

• Load the contents of memory locations 200 into register R3.

• Initialize register R2 with the value 1.

• In a loop , multiply R2 and R3 stored the product in R2.

• Decrement R3 by 1 ; if not zero repeat the loop.

• store the result (from R3) in memory location 198.

ASSEMBLY LANGUAGE PROGRAM:

```
ADDI R10,R0,200
ADDI R2,R0,1
LW R3,0(R10)
LOOP : MUL R2,R2,R3
SUBI R3,R3,1
BNEQZ R3,LOOP
SW R2,-2(R10)
HLT
```

TESTBENCH CODE:

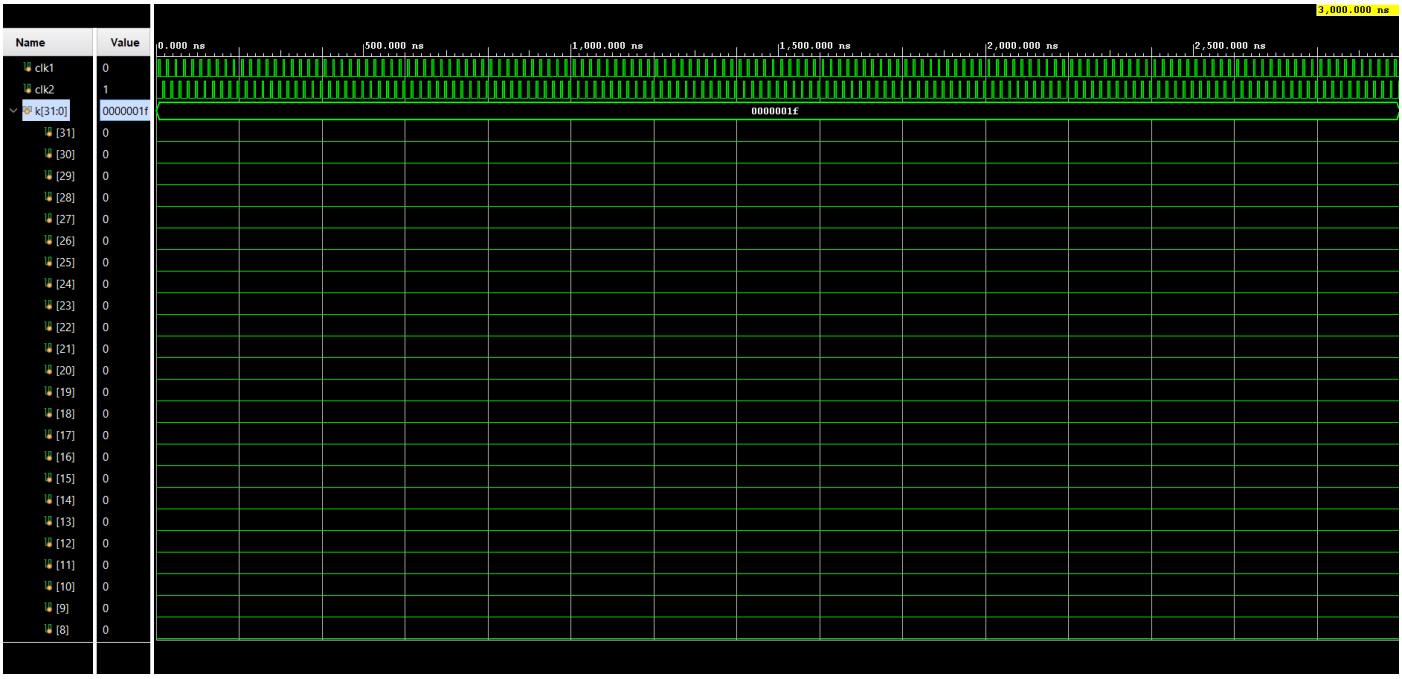
```
module test_mips32();

reg clk1,clk2;

integer k;

pipe_mips32 DUT (clk1,clk2); //Instantiating the module

initial
begin

clk1=0; clk2=0; //Two phase clock

repeat(3000)

begin

#5 clk1=1; #5 clk1=0;

#5 clk2=1; #5 clk2=0;

end
end

initial

begin

$display("MIPS32 Processor Test Case - 3");

for (k=0 ; k<31 ; k=k+1)

DUT.Reg [k] = k;

DUT.Mem[0] = 32'h280a00c8; //ADDI R10,R0,200

DUT.Mem[1] = 32'h28020001; //ADDI R2,R0,1

DUT.Mem[2] = 32'h0e94a000; //OR R20,R20,R20 (Dummy instruction)
```

```
DUT.Mem[2] = 32'h0e94a000; //OR R20,R20,R20 (Dummy instruction)

DUT.Mem[3] = 32'h21430000; //LW R3,0,(R10)

DUT.Mem[4] = 32'h0e94a000;  //OR R20,R20,R20 (Dummy instruction)

DUT.Mem[5] = 32'h14431000; //LOOP: MUL R2,R2,R3

DUT.Mem[6] = 32'h2c630001; //SUBI R3,R3,1

DUT.Mem[7] = 32'h0e94a000; //OR R20,R20,R20 (Dummy instruction)

DUT.Mem[8] = 32'h3460fffc; //BNEQZ R3,LOOP

DUT.Mem[9] = 32'h2542fffe; //SW R2,-2(R10)

DUT.Mem[10] = 32'hfc000000; //HLT

DUT.Mem[200] = 7; //Find factorial of 7

DUT.PC = 0;

DUT.HALTED = 0;

DUT.TAKEN_BRANCH = 0;

#2000 $display("Mem[200] = %2d, Mem[198] = %6d", DUT.Mem[200], DUT.Mem[198]);

end

initial begin

$dumpfile("DUT.vcd");
$dumpvars(0,test_mips32);
$monitor ("R2: %4d", DUT.Reg[2]);
#3000 $finish;

end

endmodule
```

OUTPUT MESSAGE:

```
MIPS32 Processor Test Case - 3
R2:    2
R2:    1
R2:    7
R2:   42
R2:  210
R2:  840
R2: 2520
R2: 5040
Mem[200] =   7, Mem[198] =   5040
```

SIMULATION:



Conclusion:

Efficient instruction execution with pipelining!

→ This 5-stage pipelined MIPS-32 processor showcases how pipelining enhances CPU performance by executing multiple instructions simultaneously. With clock-phase optimization, structured pipeline registers, and effective branch handling, this design balances speed and efficiency.

Key takeaways from this project:

✔ Instruction pipelining reduces execution time

- ✔ Branching requires careful handling to avoid stalls
- ✔ Memory operations (load/store) influence pipeline behavior

Pipelining is the backbone of modern RISC processors, making it a crucial concept in computer architecture and FPGA-based processor design

ATTACHMENTS:

For More Details Check Out the Below Link 🔽

GITHUB LINK : https://lnkd.in/gUSs6NZJ