8-Bit Pipelined CPU Implementation Guide (Lab-Style Instructions)

Based on Digital Simulator - Lab 10 Reference Format

PART 1: ISA DEFINITION & SETUP

Step 1.1: Define Your 8-Bit ISA (Document This)

CPU Specifications:

- **Data Width:** 8 bits
- **Instruction Width:** 16 bits (for flexibility with immediates and addresses)
- **Registers:** 8 registers (R0-R7), each 4 bits
- **Memory:**
  - ROM: 256 bytes (8-bit addressable) - instruction storage
  - RAM: 256 bytes (8-bit addressable) - data storage
- **Flags:** Zero (Z), Non-Zero (NZ), Carry (C)

Instruction Format:

```
Format 1 (Reg-Reg): [Opcode(8)] [Reg1(3)] [Reg2(3)] [Unused(2)]
Format 2 (Reg-Imm): [Opcode(8)] [Reg(3)] [Immediate(5)]
Format 3 (Addr):    [Opcode(8)] [Address(8)]
```

Opcode Table (Use for documentation & ROM):

Instruction	Opcode	Format	Encoding
NOP	0	-	00000000
LDI	1	Reg-Imm	0000 01 Rd[2:0] Imm[4:0]
LD	2	Addr	0000 10 Addr[7:0]
ADD	10	Reg-Reg	0000 1010 Rd[2:0] Rs[2:0]
SUB	11	Reg-Reg	0000 1011 Rd[2:0] Rs[2:0]

AND	20	Reg-Reg	0001 0100 Rd[2:0] Rs[2:0]
OR	21	Reg-Reg	0001 0101 Rd[2:0] Rs[2:0]
NOT	23	Reg	0001 0111 Rd[2:0] 000
ST	30	Addr	0001 1110 Addr[7:0]
WR	32	Reg	0010 0000 Reg[2:0] 00000
JMP	40	Addr	0010 1000 Addr[7:0]
BZ	41	Addr	0010 1001 Addr[7:0]

PART 2: BUILDING BASIC COMPONENTS (Like Lab 10.2 - 10.6)

Step 2.1: Create Half Adder Sub-circuit

File: `halfAdder.dig`

Instructions:

1. Open Digital → File → New → Create blank canvas
2. **Create inputs:**
   - Add → Components → IO → Input (2 times)
   - Label first input: `A` (right-click → Set Label)
   - Label second input: `B`
3. **Create half adder logic:**
   - Add → Components → Logic → XOR
   - Add → Components → Logic → AND
   - Wire: `A` and `B` to XOR → Label output: `Sum`
   - Wire: `A` and `B` to AND → Label output: `Carry`
4. **Create outputs:**
   - Add → Components → IO → Output (2 times)
   - Wire XOR output to first Output
   - Wire AND output to second Output
5. **Save:** File → Save As → `halfAdder.dig`

TA Checking: Simulate by clicking Play. Test with:

- A=0, B=0 → Sum=0, Carry=0 ✓
- A=1, B=0 → Sum=1, Carry=0 ✓
- A=1, B=1 → Sum=0, Carry=1 ✓

Step 2.2: Create Full Adder Sub-circuit

File: `fullAdder.dig`

Instructions:

1. Open Digital → File → New
2. **Add two half adders:**
   - Add → Components → Custom → `half_adder` (2 times)
   - Space them horizontally
3. **Create inputs (1-bit each):**
   - Add 3 × Input components
   - Label them: `A`, `B`, `Cin` (Carry-in)
4. **Wire first half adder:**
   - `A` → first half_adder input A
   - `B` → first half_adder input B
   - First half_adder Sum → second half_adder input A
   - `Cin` → second half_adder input B
5. **Wire outputs (1-bit each):**
   - Second half_adder Sum → Output, label: `Sum`
   - Add → Components → Logic → OR (2-input)
   - First half_adder Carry → OR input 1
   - Second half_adder Carry → OR input 2

- OR output → Output, label: `Cout`

6. **Test the circuit:**
   - Add → Components → Misc → Test
   - Right-click Test → Edit Detached
   - Add test cases:

```
A B Cin | Sum Cout
0 0 0  | 0   0
0 1 0  | 1   0
1 0 0  | 1   0
1 1 0  | 0   1
0 1 1  | 0   1
1 1 1  | 1   1
```

- Click "Run Tests" → Should pass all

7. **Save:** File → Save As → `fullAdder.dig`

TA Checking: Show passing test results. _______________

Step 2.3: Create 8-Bit Ripple-Carry Adder

File: `adder8bit.dig`

Instructions:

1. Open Digital → File → New
2. **Add inputs:**
   - Add → Components → IO → Input
   - Set Properties: Data Bits = 8
   - Label: `A`
   - Repeat for `B`, label: `B`
   - Add 1-bit Input, label: `Cin`

3. **Add splitter for A:**
    - Add → Components → Wires → Splitter/Merger
    - Properties: Input Splitting: 8 → Output Splitting: 1, 1, 1, 1, 1, 1, 1, 1
    - Wire `A` input → Splitter
    - Label outputs: `A[0]`, `A[1]`, ... `A[7]`
4. **Repeat for B:** (splitter outputs labeled `B[0]`-`B[7]`)
5. **Add 8 full adders:**
    - Add → Components → Custom → `full_adder` (8 times, arrange in column)
    - Label each: `FA0`, `FA1`, ... `FA7`
6. **Wire full adders (ripple carry chain):**
    - `A[0]` → `FA0` input A
    - `B[0]` → `FA0` input B
    - `Cin` → `FA0` input Cin
    - `FA0` Cout → `FA1` input Cin
    - `A[1]` → `FA1` input A
    - `B[1]` → `FA1` input B
    - ... (continue pattern for FA2-FA7)
7. **Add output merger:**
    - Add → Components → Wires → Splitter/Merger
    - Properties: Input: 1, 1, 1, 1, 1, 1, 1, 1 → Output: 8
    - Wire all `Sum` outputs from FA0-FA7 to merger
    - Merger output → Output component
    - Label output: `Result[7:0]` (Data Bits = 8)
8. **Add carry out:**
    - `FA7` Cout → Output component
    - Label: `Cout`

9. **Test:**
   - Add → Components → Misc → Test
   - Test cases:

```
A[7:0] B[7:0] Cin | Result[7:0] Cout
00000101 00000011 0 | 00001000 0
11111111 00000001 0 | 00000000 1
01111111 00000001 0 | 10000000 0
```

- Run tests

10. **Save:** File → Save As → `adder8bit.dig`

TA Checking: Show correct 8-bit addition with carry: _______________

Step 2.4: Create 8-Bit ALU

File: `alu8bit.dig`

Instructions:

1. Open Digital → File → New
2. **Add the 8-bit adder:**
   - Add → Components → Custom → `adder8bit`
   - Label: `Adder`
3. **Add inputs (8-bit each):**
   - Add 2 × Input (Data Bits = 8)
   - Label: `A[7:0]`, `B[7:0]`
   - Add 1 × Input (Data Bits = 3)
   - Label: `OpCode[2:0]`
4. **Wire adder:**
   - `A[7:0]` → Adder A

- B[7:0] → Adder B
- Constant 0 → Adder Cin

5. **Create output splitter:**
   - Add → Components → Wires → Splitter/Merger
   - Properties: Input: 8 → Output: 1, 1, 1, 1, 1, 1, 1, 1
   - Adder Result → Splitter input
6. **Zero flag logic:**
   - Add → Components → Logic → NOR (change Properties: Inputs = 8)
   - All Result bits → NOR inputs
   - NOR output = Zero_Flag (1 when result is 0)
7. **Flag register:**
   - Add → Components → Memory → Register
   - Properties: Data Bits = 1
   - NOR output → Register D input
   - Add 1-bit Input, label: `FlagWrite`, connect to Register en (enable)
   - Add 1-bit Input, label: `Clock`, connect to Register C (clock)
   - Register Q → 1-bit Output, label: `Zero_Flag`
8. **ALU result output:**
   - Add → Components → Wires → Splitter/Merger
   - Properties: Input: 1, 1, 1, 1, 1, 1, 1, 1 → Output: 8
   - All Adder output bits → Merger
   - Merger → 8-bit Output, label: `Result[7:0]`
9. **Carry flag:**
   - Adder Cout → Output, label: `Carry_Flag`
10. **Save:** File → Save As → `alu8bit.dig`

TA Checking: Test ALU Zero Flag Latching

1. **Set for Zero:** Input A=00001000, B=11111000 (result = 0)
   - Show NOR output = 1 ✓
2. **Latch the 1:** Set FlagWrite=1, advance Clock
   - Show Zero_Flag output = 1 ✓
3. **Hold the 1:** Change inputs to A=00000001, B=00000001 (result ≠ 0)
   - Show NOR = 0, but Zero_Flag still = 1 ✓
4. **Unlatch condition:** Set FlagWrite=0, advance Clock
   - Show Zero_Flag remains = 1 (held) ✓
5. **Latch the 0:** Set FlagWrite=1, advance Clock
   - Show Zero_Flag = 0 ✓

PART 3: MEMORY AND REGISTER COMPONENTS

Step 3.1: Create Register File (8 × 8-bit)

File: `registerFile8.dig`

Instructions:

1. Open Digital → File → New
2. **Add Register File component:**
   - Add → Components → Memory → Register File
   - Properties: Data Bits = 8, Address Bits = 3 (8 registers)
3. **Create inputs:**
   - Add 8-bit Input → Label: `Din[7:0]`, connect to register's Din
   - Add 1-bit Input → Label: `WriteEnable`, connect to we
   - Add 3-bit Input → Label: `WriteReg[2:0]`, connect to Rw
   - Add 1-bit Input → Label: `Clock`, connect to C
   - Add 3-bit Input → Label: `ReadReg1[2:0]`, connect to Ra
   - Add 3-bit Input → Label: `ReadReg2[2:0]`, connect to Rb

4. **Create outputs:**
   - Add 8-bit Output → Label: `ReadData1[7:0]`, connect from Da
   - Add 8-bit Output → Label: `ReadData2[7:0]`, connect from Db
5. **Save:** File → Save As → `registerFile8.dig`

TA Checking: Register File Read/Write

1. **Write to R2:**
   - WriteEnable = 1
   - WriteReg = 010 (binary for 2)
   - Din = 11010101 (binary for 213)
   - Clock tick (advance clock)
   - Right-click RegisterFile → View to see data stored ✓
2. **Read from R2 on Port 1:**
   - ReadReg1 = 010
   - Show ReadData1 = 11010101 ✓
3. **Read from R2 on Port 2:**
   - ReadReg2 = 010
   - Show ReadData2 = 11010101 ✓
4. **Verify Independence:**
   - Write value 01001010 to R3 (WriteReg = 011)
   - Read R3 from ReadReg1 → Show 01001010 ✓
   - Read R2 still shows 11010101 (unchanged) ✓

Step 3.2: Create Program Counter (PC)

File: `programCounter8.dig`

Instructions:

1. Open Digital → File → New

2. **Add counter component:**
    - Add → Components → Memory → Counter with preset
    - Properties: Data Bits = 4 (for 16 ROM locations, can extend to 8)

3. **Create inputs:**
    - Add 1-bit Input → Label: `Clock`, connect to C
    - Add 4-bit Input → Label: `LoadData[3:0]`, connect to in
    - Add 1-bit Input → Label: `Load`, connect to ld
    - Add 1-bit Input → Label: `Clear`, connect to clr

4. **Wire enable:**
    - Add → Components → Wires → Supply Voltage
    - Connect to counter en (enable)

5. **Wire direction:**
    - Add → Components → Wires → Ground
    - Connect to counter dir (always count up)

6. **Create output:**
    - Add 4-bit Output → Label: `PC[3:0]`, connect from counter out

7. **Save:** File → Save As → `programCounter8.dig`

TA Checking: PC Behavior

1. **Count mode:** Clock pulses with Load=0, Clear=0
    - Show PC counts: 0→1→2→3→4→5→6→7→0→1... ✓

2. **Load mode:** Set Load=1, LoadData=1010 (10), Clock tick
    - Show PC = 1010 ✓

3. **Clear mode:** Set Clear=1, Clock tick
    - Show PC = 0000 ✓

Step 3.3: Create Program ROM

Step 3.3: Create Program ROM

File: `programROM.dig`

Instructions:

1. Open Digital → File → New
2. **Add ROM component:**
   - Add → Components → Memory → ROM
   - Properties: Address Bits = 4, Data Bits = 16 (16-bit instructions)
3. **Create input:**
   - Add 4-bit Input → Label: `Address[3:0]`, connect to A
4. **Wire select:**
   - Add → Components → Wires → Supply Voltage
   - Connect to sel (always selected)
5. **Create output:**
   - Add 16-bit Output → Label: `Instruction[15:0]`, connect from D
6. **Load test program:**
   - Right-click ROM → Edit Content
   - Enter 16-bit test program (see test program section)
   - Example:

```
Address 0: 0000010001010011 (LDI R1, 5)
Address 1: 0000010010001010 (LDI R2, 10)
Address 2: 0000101000010010 (ADD R1, R2)
Address 3: 0010000000000001 (WR R1)
Address 4: 0010100000000000 (JMP 0)
```

7. **Save:** File → Save As → `programROM.dig`

TA Checking: ROM Instruction Fetch

- Set Address = 0 → Show Instruction = 0000010001010011 ✓
- Set Address = 1 → Show Instruction = 0000010010001010 ✓

- Set Address = 2 → Show Instruction = 0000101000010010 ✓

Step 3.4: Create Data RAM

File: `dataRAM.dig`

Instructions:

1. Open Digital → File → New
2. **Add RAM component:**
   - Add → Components → Memory → RAM, separate Ports
   - Properties: Address Bits = 8, Data Bits = 8
3. **Create inputs:**
   - Add 8-bit Input → Label: `Address[7:0]`, connect to A
   - Add 8-bit Input → Label: `DataIn[7:0]`, connect to Din
   - Add 1-bit Input → Label: `MemWrite`, connect to str
   - Add 1-bit Input → Label: `MemRead`, connect to ld
   - Add 1-bit Input → Label: `Clock`, connect to C
4. **Create output:**
   - Add 8-bit Output → Label: `DataOut[7:0]`, connect from Dout
5. **Save:** File → Save As → `dataRAM.dig`

TA Checking: RAM Store/Load Operations

1. **Store operation:**
   - Set Address = 00000101 (5)
   - Set DataIn = 11110000
   - Set MemWrite = 1
   - Clock tick
   - Right-click RAM → View, show data stored at address 5 ✓

2. **Load operation:**
   - Set Address = 00000101
   - Set MemRead = 1
   - Clock tick
   - Show DataOut = 11110000 ✓
3. **Multiple values:**
   - Store 10101010 at address 3, Clock tick
   - Store 01010101 at address 7, Clock tick
   - Load from address 3 → Show 10101010 ✓
   - Load from address 7 → Show 01010101 ✓

PART 4: PIPELINE REGISTERS

Step 4.1: Create FD_Reg (Fetch-Decode Pipeline Register)

File: `FD_Register.dig`

Instructions:

1. Open Digital → File → New
2. **Add register components:**
   - Add → Components → Memory → Register (2 times)
   - Set Properties (first): Data Bits = 16, Label: `InstrReg`
   - Set Properties (second): Data Bits = 4, Label: `PCReg`
3. **Create inputs:**
   - Add 16-bit Input → `Instruction[15:0]` → InstrReg D input
   - Add 4-bit Input → `PC[3:0]` → PCReg D input
   - Add 1-bit Input → `Clock` → both register C inputs
4. **Create outputs:**

- Add 16-bit Output ← InstrReg Q output, label: `OutInstr[15:0]`
- Add 4-bit Output ← PCReg Q output, label: `OutPC[3:0]`

5. **Save:** File → Save As → `FD_Register.dig`

TA Checking: Clock pulse advances instruction/PC to next stage _______________

Step 4.2: Create DE_Reg (Decode-Execute Pipeline Register)

File: `DE_Register.dig`

Instructions:

1. Open Digital → File → New
2. **Add register components:**
   - Add → Components → Memory → Register (5 times)
   - Data Bits: 8, 8, 3, 8, 3 (for: Data1, Data2, Imm, OpCode, DestReg)
3. **Create inputs (connect to registers' D):**
   - 8-bit: `RegData1[7:0]`
   - 8-bit: `RegData2[7:0]`
   - 3-bit: `Immediate[2:0]`
   - 8-bit: `OpCode[7:0]`
   - 3-bit: `DestReg[2:0]`
   - 1-bit: `Clock` (common to all)
4. **Create outputs (from registers' Q):**
   - 8-bit: `OutRegData1[7:0]`
   - 8-bit: `OutRegData2[7:0]`
   - 3-bit: `OutImmediate[2:0]`
   - 8-bit: `OutOpCode[7:0]`
   - 3-bit: `OutDestReg[2:0]`
5. **Save:** File → Save As → `DE_Register.dig`

5. **Save:** File → Save As → `DE_Register.dig`

Step 4.3: Create EM_Reg & MW_Reg

File: `EM_Register.dig` & `MW_Register.dig`

Instructions (similar pattern):

EM_Reg holds:

- 8-bit: ALU Result
- 3-bit: DestReg
- 3-bit: Flags (Z, NZ, C)
- 1-bit: MemWrite signal
- 1-bit: MemRead signal

MW_Reg holds:

- 8-bit: Final Result (ALU or RAM)
- 3-bit: DestReg
- 1-bit: WriteEnable
- 3-bit: Flags

(Follow same steps as FD_Register and DE_Register)

PART 5: CONTROL UNIT

Step 5.1: Create Instruction Decoder

File: `controlUnit.dig`

Instructions:

1. Open Digital → File → New
2. **Add inputs:**
   - Add 16-bit Input → `Instruction[15:0]`

- Add 3-bit Input → `Flags[2:0]`

3. **Extract opcode (bits 8-15):**
   - Add → Components → Wires → Splitter/Merger
   - 16-bit input splitter with output: 1,1,1,1,1,1,1,1 (16 individual bits)
   - Take bits 8-15, merge into 8-bit output: `OpCode[7:0]`
4. **Extract register fields (bits 5-7, 2-4):**
   - Splitter bits 5-7 → merge into 3-bit: `ReadReg1[2:0]`
   - Splitter bits 2-4 → merge into 3-bit: `ReadReg2[2:0]`
5. **Create control logic using multiplexers/ROM:**
   - **Option A (Simple):** Add ROM with 256 entries
     - Address input = OpCode
     - Each ROM entry outputs: ReadReg1, ReadReg2, WriteReg, WriteEnable, MemRead, MemWrite, ALU_Op
   - **Option B (Logic gates):** Use cascading multiplexers for each opcode
6. **Create outputs:**
   - 3-bit: `ReadReg1[2:0]` → RegisterFile
   - 3-bit: `ReadReg2[2:0]` → RegisterFile
   - 3-bit: `WriteReg[2:0]` → RegisterFile write port
   - 1-bit: `WriteEnable` → RegisterFile
   - 1-bit: `MemRead`, `MemWrite` → RAM
   - 8-bit: `ALU_Op[7:0]` → ALU
7. **Branch condition logic:**
   - Add → Components → Logic → AND/OR gates
   - Logic: IF (OpCode==BZ) AND (Zero_Flag==1) → BranchTaken=1
   - Output: 1-bit `BranchTaken`
8. **Save:** File → Save As → `controlUnit.dig`

TA Checking: Test control signals

- Input: Instruction for "LDI R1, 5" → Show WriteReg=001, WriteEnable=1 ✓
- Input: Instruction for "ADD R1, R2" → Show ALU_Op=1010 ✓
- Input: Instruction for "BZ" with Zero_Flag=1 → Show BranchTaken=1 ✓

PART 6: COMPLETE CPU INTEGRATION

Step 6.1: Create Main CPU Circuit

File: `CPU_8bit.dig`

Instructions:

1. Open Digital → File → New (main canvas)
2. **Add all sub-circuits:**
    - Add → Components → Custom:
        - `programCounter8` - label: "PC"
        - `programROM` - label: "ROM"
        - `FD_Register` - label: "FD_Reg"
        - `controlUnit` - label: "Control"
        - `registerFile8` - label: "RegFile"
        - `DE_Register` - label: "DE_Reg"
        - `alu8bit` - label: "ALU"
        - `EM_Register` - label: "EM_Reg"
        - `dataRAM` - label: "RAM"
        - `MW_Register` - label: "MW_Reg"
3. **Arrange hierarchically (left to right, top to bottom):**

```
[PC] → [ROM] → [FD_Reg]
              ↓
```

```
[Control] ← [RegFile] ← [DE_Reg]
    ↓
[ALU] → [EM_Reg]
         ↓
[RAM] ← [MW_Reg] → [RegFile Write]
```

4. **Wire pipeline data flow: Fetch Stage:**
   - PC out[3:0] → ROM Address[3:0]
   - ROM Instruction[15:0] → FD_Reg input

   Decode Stage:
   - FD_Reg OutInstr[15:0] → Control Instruction[15:0]
   - Control ReadReg1[2:0], ReadReg2[2:0] → RegFile Ra, Rb
   - RegFile Da[7:0], Db[7:0] → DE_Reg RegData1, RegData2
   - Control outputs → DE_Reg inputs (OpCode, Immediate, DestReg)
   - FD_Reg OutPC[3:0] → (keep for branch calculation)

   Execute Stage:
   - DE_Reg OutRegData1[7:0], OutRegData2[7:0] → ALU A, B
   - DE_Reg OutOpCode[7:0] → ALU OpCode
   - ALU Result[7:0], Flags[2:0] → EM_Reg inputs
   - DE_Reg OutDestReg[2:0] → EM_Reg input

   Memory Stage:
   - EM_Reg ALU_Result (as address) → RAM Address[7:0]
   - EM_Reg MemWrite → RAM MemWrite
   - EM_Reg MemRead → RAM MemRead
   - RAM DataOut[7:0] → MW_Reg input

   Writeback Stage:
   - MW_Reg Result[7:0] → RegFile Din (write port)
   - MW_Reg DestReg[2:0] → RegFile WriteReg
   - MW_Reg WriteEnable → RegFile WriteEnable

5. **Add hazard detection (optional but recommended):**
   - Add → Components → Wires → AND/OR gates
   - Compare: EM_Reg DestReg == DE_Reg ReadReg1/ReadReg2
   - If match AND MemRead: Insert NOP (set DE_Reg instruction to 0)
   - Hold PC (don't increment)
6. **Add I/O ports:**
   - Add 8-bit Input → `InputPort[7:0]` (connect to RD instruction)
   - Add 8-bit Output → `OutputPort[7:0]` (connect to WR instruction)
   - Route through output buffer
7. **Add global clock:**
   - Add → Components → Wires → Clock
   - Connect to: PC Clock, FD_Reg, DE_Reg, EM_Reg, MW_Reg, RegFile Clock, RAM Clock, ALU FlagWrite
8. **Add branch control:**
   - Control BranchTaken → PC Load signal
   - Control BranchAddress → PC LoadData
   - Route to PC load/branch when BZ/JMP taken
9. **Add test points (probes):**
   - Add → Components → IO → Probe (multiple)
   - Monitor: PC value, Instruction at each stage, RegFile writes, Output value
   - This helps debugging
10. **Save:** File → Save As → `CPU_8bit.dig`

TA Checking: CPU Initialization & Basic Operation

- Show all components instantiated and connected
- Clock signal reaches all pipeline stages ✓
- PC increments on each clock cycle ✓

- ROM outputs instruction correctly to FD_Reg ✓

PART 7: TESTING WITH ASSEMBLY PROGRAM

Step 7.1: Write Test Assembly Program

Test Program (in comments, then convert to hex):

```assembly
; Simple test: Load, Add, Store, Output
START:
  LDI R1, 5      ; Load 5 into R1
  LDI R2, 3      ; Load 3 into R2
  ADD R1, R2     ; R1 = R1 + R2 = 8
  WR R1          ; Write R1 to output (should show 8)

  LD R0, 0       ; Load from RAM[0]
  ST R1, 0       ; Store R1 to RAM[0]

  JMP START      ; Loop back (infinite loop for demo)
```

Step 7.2: Load Program into ROM

Instructions:

1. Right-click CPU_8bit ROM component → Edit Content
2. Convert assembly to machine code:
   - LDI R1, 5: Opcode=1, Rd=001, Imm=00101 → `0000010001000101`
   - LDI R2, 3: Opcode=1, Rd=010, Imm=00011 → `0000010010000011`
   - ADD R1, R2: Opcode=10, Rd=001, Rs=010 → `0000101000010010`
   - WR R1: Opcode=32, Reg=001 → `0010000000010000`
   - LD R0, 0: Opcode=2, Addr=00000000 → `0000100000000000`
   - ST R1, 0: Opcode=30, Addr=00000000 → `0001111000000000`
   - JMP START: Opcode=40, Addr=00000000 → `0010100000000000`

3. Enter into ROM editor:

```
Address 0: 0000010001000101  (LDI R1, 5)
Address 1: 0000010010000011  (LDI R2, 3)
Address 2: 0000101000010010  (ADD R1, R2)
Address 3: 0010000000010000  (WR R1)
Address 4: 0000100000000000  (LD R0, 0)
Address 5: 0001111000000000  (ST R1, 0)
Address 6: 0010100000000000  (JMP 0)
Address 7: 0000000000000000  (NOP - padding)
```

4. Click Save/OK in ROM editor

Step 7.3: Simulate the CPU

Instructions:

1. **Start simulation:**
   - Click Play button (or Simulate → Run)
2. **Single-step through program:**
   - Use Simulate → Step (or keyboard shortcut)
   - After each step, observe:
     - PC value (should increment 0→1→2→3...)
     - Current instruction in FD_Reg
     - Register file contents
     - Output port value
3. **Expected behavior: Clock Cycle 0:**
   - PC = 0
   - ROM outputs instruction at address 0: LDI R1, 5
   - FD_Reg holds this instruction

   Clock Cycle 1:
   - PC = 1
   - Previous instruction moves through pipeline

- ROM outputs instruction at address 1: LDI R2, 3
- Control unit decodes: WriteReg=001, WriteEnable=1, WriteData=5
- Register file writes 5 to R1

Clock Cycle 2:

- PC = 2
- LDI R2, 3 in execute stage (RegFile writes 3 to R2)
- ROM outputs instruction at address 2: ADD R1, R2

Clock Cycle 3:

- PC = 3
- ADD R1, R2 in execute stage
- ALU receives: A=5 (R1), B=3 (R2), OpCode=ADD
- ALU outputs: Result=8, Zero_Flag=0, Carry=0

Clock Cycle 4:

- PC = 4
- ALU result (8) flows to writeback stage
- RegFile writes 8 to R1
- ROM outputs instruction at address 3: WR R1 (Write to output)

Clock Cycle 5:

- PC = 5
- WR R1 executes: OutputPort should show 8
- ROM outputs instruction at address 4: LD R0, 0

TA Checking: Step through program and verify each stage

1. **After Clock 0:** Show PC=0, FD_Reg displays first instruction ✓
2. **After Clock 1:**
   - Show PC=1
   - Show R1 being written with value 5 ✓
3. **After Clock 2:**

3. **After Clock 2:**
   - Show PC=2
   - Show R2 being written with value 3 ✓
4. **After Clock 3:**
   - Show PC=3
   - Show ALU executing ADD operation ✓
5. **After Clock 4:**
   - Show R1 updated to 8 (5+3)
   - Show WR instruction in decode stage ✓
6. **After Clock 5:**
   - Show OutputPort = 8 (WR R1 result) ✓

PART 8: ADVANCED TESTING & DEBUGGING

Step 8.1: Test Branch Instructions

File: Create new test program for branching

Instructions:

1. **Create BZ (Branch if Zero) test program:**

```assembly
START:
  LDI R1, 0      ; Load 0 into R1
  LDI R2, 5      ; Load 5 into R2
  SUB R1, R2     ; R1 = R1 - R2 = -5 (not zero)
  BZ ZERO_LABEL  ; Branch if zero (should NOT branch)
  LDI R0, 99     ; This should execute
  WR R0          ; Output 99
  JMP START

ZERO_LABEL:
  LDI R0, 1      ; This should NOT execute
```

```
    WR R0
    JMP START
```

2. **Convert to machine code and load into ROM**
3. **Step through:**
   - After SUB: Zero_Flag should = 0 (result ≠ 0)
   - BZ should NOT take branch (PC continues to next instruction)
   - Output should show 99, not 1 ✓
4. **Test BZ when condition is true:**
   - Modify: LDI R1, 5; LDI R2, 5; SUB R1, R2
   - Now R1 = 0, so Zero_Flag = 1
   - BZ should take branch
   - Output should show 1 ✓

TA Checking: Branch correctly taken/not taken based on flag _______________

Step 8.2: Test Memory Operations (ST/LD)

Instructions:

1. **Create memory test program:**

```assembly
; Store values to RAM, then read them back
START:
    LDI R1, 42      ; Load 42
    ST R1, 0        ; Store to RAM[0]

    LDI R2, 100     ; Load 100
    ST R2, 1        ; Store to RAM[1]

    LDI R3, 0       ; Load address 0
    LD R4, R3       ; Load from RAM[0] into R4
    WR R4           ; Output should be 42

```

```
  JMP START
```

2. **Right-click RAM component → View before and after store operations**

    - Verify values written to correct addresses

3. **Monitor outputs:**
    - First WR R4 should show 42 ✓
    - Later LD operations should retrieve correct values ✓

TA Checking: Verify RAM store/load operations _______________

Step 8.3: Test Pipeline Hazard Detection

Instructions (Advanced):

1. **Create program with data dependency:**

```assembly
START:
  LDI R1, 10     ; Write to R1
  ADD R1, R2     ; Read R1 (dependent on previous instruction!)
  WR R1
  JMP START
```

2. **Observe pipeline behavior:**
    - **Without forwarding:** ADD would stall for 1-2 cycles waiting for LDI to complete
    - **With forwarding (bonus feature):** ALU result forwarded directly, no stall
    - **With stalls:** NOP inserted in DE_Reg

3. **Show hazard detection working:**
    - Monitor EM_Reg DestReg vs DE_Reg ReadReg1/ReadReg2
    - When match detected: Show NOP in pipeline ✓

TA Checking: Hazard detection prevents incorrect results _______________

Step 8.4: Test Interrupt Handling (Bonus)

Instructions:

1. **Create interrupt vector in ROM (address 200-207):**

assembly

```assembly
; Main program starts at 0
START:
  LDI R1, 5
  LDI R2, 10
  ADD R1, R2      ; Normal operation

  ; ... more code ...

  JMP START

; Interrupt handler starts at 200
INT_HANDLER:
  LDI R0, 255    ; Load special value
  WR R0          ; Signal interrupt occurred
  RTI            ; Return from interrupt
```

2. **Add interrupt input:**
   - In main CPU circuit: Add 1-bit Input labeled `IRQ` (Interrupt Request)
   - Route to Control Unit
3. **Add interrupt logic:**
   - When IRQ=1 during Decode stage:
     - Save return address (PC) to stack (or R7)
     - Load PC with interrupt vector address (200)
     - Disable interrupts (set interrupt flag in SR)
4. **Test sequence:**
   - Run normal program for 5-10 cycles
   - Set IRQ=1
   - Observe PC jump to 200 (interrupt handler)

- Handler executes, outputs 255
- RTI returns to main program ✓

TA Checking: Interrupt correctly diverts execution _______________

PART 9: OPTIMIZATION FEATURES (BONUS POINTS - EXTRA CREDIT)

Step 9.0: Implement Basic Branch Prediction

File: `branchPredictor.dig`

Purpose: Predict branch direction to reduce pipeline flush penalties

Instructions:

1. **Create branch prediction table (BPT):**
   - Add → Components → Memory → RAM (separate ports)
   - Properties: Address Bits = 4, Data Bits = 1 (stores one prediction bit per branch)
   - This allows 16 entries for different branch addresses
   - Each entry: 0=Not Taken, 1=Taken
2. **Create Branch History Register (BHR):**
   - Add → Components → Memory → Register
   - Properties: Data Bits = 1
   - Stores last branch outcome
   - Updated after branch resolves
3. **Create prediction logic:**
   - **Predictor type:** Simple 1-bit predictor (toggles on every misprediction)
   - When branch instruction detected:
     - Read BPT[PC] to get prediction
     - Speculatively fetch next instruction based on prediction

    - Don't flush pipeline yet

  - **After branch resolves:**
    - If prediction correct: Continue speculatively
    - If prediction wrong: Flush pipeline & update BPT entry

4. **Integration in CPU:**
   - Add BPT component to CPU circuit
   - PC → BPT Address input
   - BPT output → Prediction signal to PC multiplexer
   - If BZ instruction detected:
     - Predicted PC = (prediction) ? BranchAddress : PC+1
   - After EM stage resolves actual branch: Update BPT if mismatch

5. **Benefits:**
   - Reduces branch penalty from 3-4 cycles to 0 cycles (if correct)
   - Improves CPI (cycles per instruction)
   - More realistic modern CPU design

TA Checking: Branch prediction with accuracy measurement

1. Load program with 10 sequential branches
2. Show prediction table entries updating
3. Count correct vs incorrect predictions
4. Demonstrate performance improvement over naive (always flush)

Expected accuracy: 80-90% for regular loop patterns ✓

Step 9.0B: Implement L1 Instruction Cache

File: `L1_ICache.dig`

Purpose: Reduce ROM access time for frequently used instructions

Instructions:

1. **Create instruction cache memory:**
   - Add → Components → Memory → RAM (separate ports)
   - Properties: Address Bits = 4, Data Bits = 24
   - (Stores: Valid bit + PC[3:0] + Instruction[16:0])
   - 16 cache lines (entries)
2. **Create cache tag comparison logic:**
   - Add → Components → Logic → Comparator (4-bit)
   - Compare incoming PC[3:0] with stored PC in cache line
   - Output: Tag match signal (1 if match, 0 if miss)
3. **Create cache hit/miss logic:**
   - 1-bit AND gate:
     - Input 1: Tag match (from comparator)
     - Input 2: Valid bit (from cache entry)
   - Output: Cache_Hit signal
4. **Create cache control logic:**
   - If Cache_Hit = 1:
     - Use cached instruction (fast path, 1 cycle)
     - Do NOT access ROM
   - If Cache_Hit = 0:
     - Access ROM (slow path, 2 cycles)
     - Cache miss → insert NOP in pipeline
     - After ROM returns: Write to cache
5. **Cache replacement policy:**
   - **Simple FIFO:** Replace oldest entry
   - Counter for replacement pointer
   - Increments on every cache miss

- Increments on every cache miss

6. **Integration in CPU:**
   - Replace direct ROM access with:

```
PC → L1_ICache
 ├── Hit: Instruction[15:0] → FD_Reg (1 cycle)
 └── Miss: PC → ROM (2 cycles) → FD_Reg + update cache
```

- Add multiplexer to select between cached and ROM instruction

7. **Benefits:**
   - ~50-70% cache hit rate on loop-heavy programs
   - Reduces average ROM access latency
   - Increases throughput (fewer stalls)
   - Power efficient (ROM accessed less)

TA Checking: L1 Cache operation

1. Run loop program (same branch repeatedly)
2. Show cache fills up:
   - First iteration: 4 cache misses, 4 ROM accesses
   - Second iteration: 4 cache hits, 0 ROM accesses ✓
3. Measure performance:
   - Without cache: 2 cycles per loop iteration
   - With cache: 1 cycle per loop iteration ✓
4. Show cache validity/tag fields updating correctly

Step 9.0C: Implement L1 Data Cache

File: `L1_DCache.dig`

Purpose: Cache frequently accessed data to speed up memory operations

Instructions:

1. **Create data cache memory:**
   - Add → Components → Memory → RAM (separate ports)
   - Properties: Address Bits = 3, Data Bits = 12
   - (Stores: Valid bit + Dirty bit + Address[7:0] + Data[7:0])
   - 8 cache lines (entries)
2. **Create write-back policy:**
   - **Write policy:** Write-back (dirty bit tracks modifications)
   - When data written: Set Dirty = 1
   - When cache line replaced: If Dirty = 1, write back to main RAM
3. **Cache coherency logic:**
   - If address match in cache:
     - **Read hit:** Return cached data (1 cycle)
     - **Write hit:** Update cache + set Dirty bit
   - If no match:
     - **Read miss:** Access RAM (2 cycles) + load to cache
     - **Write miss:** Access RAM (2 cycles) + update cache
4. **Integration in CPU:**
   - ST instruction: RAM address → L1_DCache
     - If hit: Write to cache (fast)
     - If miss: Write to RAM (slow)
   - LD instruction: RAM address → L1_DCache
     - If hit: Return cached data
     - If miss: Load from RAM → cache
5. **Benefits:**
   - Reduces RAM access latency
   - Useful for array operations and stack accesses
   - Improves data-intensive program performance

- Improves data-intensive program performance

TA Checking: Data cache with write-back

1. Store 5 values to same address repeatedly
   - Show cache hit every time after first write ✓
2. Replace cache line with dirty data
   - Show write-back to main RAM before replacement ✓
3. Load data multiple times
   - Show cache hits reducing RAM accesses ✓

Step 9.1: Implement Register Forwarding (Innovation #3)

File: `forwardingUnit.dig`

Instructions:

1. **Create forwarding control logic:**
   - Add → Components → Logic → Comparators
   - Compare: EM_Reg.DestReg == DE_Reg.ReadReg1
   - If match: Forward EM_Reg.Result to ALU input A
   - Compare: MW_Reg.DestReg == DE_Reg.ReadReg1
   - If match: Forward MW_Reg.Result to ALU input A (if not forwarded from EM)
2. **Create forwarding multiplexer:**
   - Add → Components → Plexers → Multiplexer (8-bit, 3-input selector)
   - Input 0 (sel=00): Normal RegFile output (ReadData1)
   - Input 1 (sel=01): EM_Reg result (bypassing writeback delay)
   - Input 2 (sel=10): MW_Reg result (from previous instruction)
   - Selector from comparator logic
3. **Integration in main CPU:**
   - Replace direct RegFile→ALU connection with Forwarding Mux

- Add forwarding unit outputs to ALU inputs

4. **Benefits:**
   - Reduces stalls in dependent instructions
   - Increases throughput
   - More realistic CPU design ✓

TA Checking: Dependent instructions execute without stalls _______________

Step 9.2: Implement Multi-Level Interrupts (Innovation #5)

File: `interruptPriority.dig`

Instructions:

1. **Add multiple interrupt inputs:**
   - Add 3 × 1-bit Inputs: `IRQ0`, `IRQ1`, `IRQ2`
   - Priorities: IRQ2 > IRQ1 > IRQ0 (higher index = higher priority)
2. **Create priority encoder:**
   - Add → Components → Logic → OR/AND gates
   - Logic:

```
IF IRQ2 = 1:
  Priority = 2, Vector = 200
ELSE IF IRQ1 = 1:
  Priority = 1, Vector = 192
ELSE IF IRQ0 = 1:
  Priority = 0, Vector = 184
ELSE:
  Priority = -1, Vector = 0 (no interrupt)
```

3. **Wire to CPU:**
   - Priority encoder output → Control Unit
   - Selected vector → PC load address when interrupt taken

4. **Benefits:**
   - Handle multiple interrupt sources
   - Prioritize critical events
   - More sophisticated interrupt handling ✓

TA Checking: Higher priority interrupts serviced first _______________

TA Checking: Higher priority interrupts serviced first _______________

Step 9.3B: Implement Stack Pointer & Push/Pop Instructions

File: `stackOperations.dig`

Purpose: Enable efficient subroutine calls and local variables

Instructions:

1. **Designate R7 as Stack Pointer (SP):**
   - Initialize SP = 0xFF (top of RAM)
   - Decrement SP on PUSH, increment on POP
2. **Add new instructions:**
   - **PUSH:** Decrement SP, write register to RAM[SP]
   - **POP:** Read RAM[SP], increment SP, write to register
   - **CALL:** Push PC+1 to stack, jump to address
   - **RET:** Pop from stack to PC (return from subroutine)
3. **Create push/pop control logic:**
   - PUSH R1:

```
SP = SP - 1
RAM[SP] = R1
```

- POP R1:

```
R1 = RAM[SP]
SP = SP + 1
```

4. **Add SP arithmetic (decrement/increment):**
   - Create small 8-bit up/down counter for SP
   - Connect to RegisterFile R7 write path
5. **Benefits:**
   - Enable function calls with parameters on stack
   - Support local variable storage
   - Professional subroutine architecture

Test Program with Stack:

```assembly
MAIN:
  LDI R1, 10
  CALL FUNC
  WR R1          ; Should output result from function
  JMP MAIN

FUNC:
  PUSH R1        ; Save R1 to stack
  LDI R1, 20
  ADD R1, [SP]   ; Add stack value to R1
  POP R1         ; Restore and pop
  RET
```

TA Checking: Stack operations work correctly ________________

Step 9.3C: Implement Single-Cycle Instruction Optimization

File: `fastPath.dig`

Purpose: Bypass pipeline for simple instructions to reduce latency

Instructions:

1. **Identify simple instructions (no memory access):**
   - LDI (Load Immediate)
   - ADD, SUB, AND, OR, NOT (Arithmetic/Logic)
   - NOT accessing memory → Can complete in 2-3 cycles instead of 5
2. **Create fast-path multiplexer:**
   - If instruction is LDI or ALU-only:
     - Route directly: Decode → Execute → WriteBack (skip Memory stage)
     - Use NOP in Memory stage position
   - If instruction is LD/ST/RD/WR:
     - Use full 5-stage pipeline (includes Memory)
3. **Comparator logic:**
   - OpCode == LDI OR OpCode == ADD OR ... → FastPath_Enable = 1
   - Else → FastPath_Enable = 0 (use normal pipeline)
4. **Control signal routing:**
   - IF FastPath_Enable = 1:
     - EM_Reg set to NOP (instruction bypasses Memory)
     - ExecuteResult goes directly to WriteBacks in next cycle
   - ELSE:
     - Normal pipeline flow
5. **Benefits:**
   - 40% reduction in latency for LDI instructions
   - Improves CPI for compute-heavy programs
   - Shows deep architecture understanding

Performance Comparison:

- Normal LDI: 5 cycles (Fetch→Decode→Execute→Memory(NOP)→Writeback)
- Fast-path LDI: 3 cycles (Fetch→Decode→Execute+Writeback combined)

TA Checking: Instruction latency reduction ______________

Step 9.3D: Implement Instruction-Level Parallelism (Out-of-Order Hazard Avoidance)

File: `OoO_Scheduler.dig` (Optional advanced feature)

Purpose: Allow independent instructions to execute without waiting

Instructions:

1. **Add instruction dependency analyzer:**
   - Compare ReadReg/WriteReg across consecutive instructions
   - If no register overlap → Can execute in parallel
2. **Create issue logic:**
   - Load 2 instructions per cycle (if no dependencies)
   - Each can enter pipeline independently
3. **Example:**

```
Instruction 1: LDI R1, 5     (no dependencies)
Instruction 2: LDI R2, 10    (no dependencies with #1)
→ Both can execute simultaneously!

Instruction 3: ADD R1, R2    (depends on R1, R2 from above)
→ Must wait for R1, R2 ready
```

TA Checking: Multiple independent instructions execute in parallel ______________

Step 9.4: Add Output Buffer/Latch (I/O Enhancement)

File: `outputBuffer.dig`

Instructions:

1. **Create persistent output register:**
   - Add → Components → Memory → Register (8-bit)
   - Properties: Data Bits = 8
2. **Create control logic:**
   - Add 1-bit Input: `WriteOutput` (high when WR instruction executes)
   - Add 8-bit Input: `DataToOutput[7:0]`
   - Connect to Register:
     - DataToOutput → Register D input
     - WriteOutput → Register en (enable)
     - Clock → Register C
3. **Output:**
   - Register Q → 8-bit Output component (labeled OutputPort)
4. **Benefits:**
   - Output persists even after instruction completes
   - LEDs/displays don't flicker
   - Realistic I/O behavior ✓

TA Checking: Output value latches and holds ______________

PART 10: DOCUMENTATION & SUBMISSION

Step 10.1: Create Technical Documentation

Document structure (5-10 pages PDF):

1. **Title Page**
   - Project: 8-Bit Pipelined CPU
   - Team members
   - Date
2. **Executive Summary (1 page)**

- Brief overview of CPU design
- Key features implemented
- Performance metrics (CPI, throughput)

3. **Instruction Set Architecture (1-2 pages)**
   - Complete opcode table with encoding
   - Instruction format diagrams
   - Example instruction encodings
4. **Architecture Design (2-3 pages)**
   - Overall block diagram (from Part 1 circuit diagram)
   - Pipeline stage descriptions with flowcharts
   - Data path and control path explanations
   - Sub-circuit hierarchy diagram
5. **Module Descriptions (1-2 pages)**
   - **RegisterFile:** Dual-port design, timing
   - **ALU:** Supported operations, flag generation
   - **ControlUnit:** Opcode decoding logic
   - **Memory:** ROM instruction format, RAM access timing
   - **Pipeline Hazard Detection:** Stall conditions
6. **Test Program & Results (1-2 pages)**
   - Annotated assembly code with comments
   - Simulation screenshots showing:
     - Program execution at each stage
     - Register and memory contents
     - Output values
     - Flag updates
7. **Challenges & Solutions (0.5-1 page)**
   - Hazard detection implementation

- Hazard detection implementation
- Timing synchronization of pipeline
- ROM instruction encoding
- Solutions applied

8. **Innovation Features (0.5-1 page)**
   - **If implemented:** Register forwarding diagram and benefits
   - **If implemented:** Interrupt priority logic and examples
   - Performance improvements achieved
9. **Appendix (optional)**
   - Complete machine code listing
   - Simulation waveform printouts
   - Sub-circuit schematics

Step 10.2: Create Video Demonstration

Video structure (5-10 minutes):

Segment 1: Overview (1 minute)

- Show complete CPU circuit in Digital
- Point out main components (ROM, RegisterFile, ALU, Pipeline stages)
- Explain data flow

Segment 2: Component Deep Dive (2 minutes)

- Zoom into RegisterFile → show read/write in action
- Zoom into ALU → show arithmetic operations
- Zoom into each pipeline register → show data flowing through

Segment 3: Simple Program Execution (2 minutes)

- Run basic program: LDI R1, 5; LDI R2, 3; ADD R1, R2; WR R1

- Step through each clock cycle
- Show register values updating
- Show output changing to 8

Segment 4: Branch Testing (1-2 minutes)

- Run program with BZ instruction
- Show Zero_Flag being set
- Show branch taken/not taken correctly
- Show different output based on branch path

Segment 5: Advanced Features (1-2 minutes)

- Memory operations (ST/LD) with RAM view
- Hazard detection (if implemented)
- Interrupt handling (if implemented)
- Forwarding logic (if implemented)

Segment 6: Conclusion (0.5 minute)

- Summary of functionality
- Highlight innovative features
- Closing remarks

Step 10.3: Prepare for Presentation

Presentation checklist:

- [ ] All .dig files organized in folder
- [ ] PDF documentation complete (5-10 pages)
- [ ] Video demo recorded and saved
- [ ] CPU circuit tested and working
- [ ] Assembly programs tested
- [ ] Screenshots/waveforms captured
- [ ] All team members can explain design

☐ All team members can explain design

☐ Bonus features documented (forwarding, interrupts, etc.)

☐ Live demo ready (laptop + projector setup tested)

Live demo talking points:

- "This is our 8-bit pipelined CPU with 5 stages"
- "The pipeline allows multiple instructions in parallel"
- "We implemented hazard detection for data dependencies"
- "This optimization feature (forwarding/interrupts) improves performance"
- "Our test program demonstrates all required instructions"

PART 11: GRADING CHECKLIST

Functionality (60%)

☐ All 8 registers (R0-R7) working

☐ All required instructions implemented (LD, ADD, SUB, AND, OR, NOT, ST, RD, WR, JMP, BZ, BNZ, BC)

☐ 5-stage pipeline implemented (Fetch, Decode, Execute, Memory, Writeback)

☐ ROM correctly stores and retrieves instructions

☐ RAM stores and retrieves data

☐ Branches work correctly (taken/not taken based on flags)

☐ Flags (Z, NZ, C) update correctly

☐ I/O ports (input/output) functional

☐ Test program runs successfully

Design Quality (20%)

☐ Modular design with proper sub-circuits

☐ Clear, organized circuit layout

☐ Proper naming and labeling of components

☐ Efficient data paths

☐ Minimal gate count where possible

☐ Documented signal flow

Documentation (15%)

☐ ISA clearly documented

☐ Architecture diagrams provided

☐ Module descriptions complete

☐ Test program annotated

☐ Simulation results shown

☐ Professional formatting (PDF)

☐ Challenges and solutions discussed

Innovation (5%)

☐ Register forwarding implemented (reduces stalls)

☐ Multi-level interrupts (advanced interrupt handling)

☐ Output latch (persistent I/O)

☐ Optimization showing performance improvement

☐ Creative feature demonstrating architecture knowledge

Optional Extensions (Extra Credit - Up to 15% bonus):

☐ **L1 Instruction Cache** (5% bonus): Caches recently accessed instructions

- Fast path for loop-heavy programs
- ~50-70% hit rate on typical loops

☐ **L1 Data Cache** (5% bonus): Caches frequently accessed data

- Write-back policy with dirty bit
- Reduces main RAM access frequency

☐ **Branch Prediction** (5% bonus): Predicts branch direction

- Simple 1-bit predictor with branch history table
- Reduces branch penalty from 3-4 cycles to 0
- 80-90% accuracy on regular loops

☐ **Stack Operations** (3% bonus): PUSH/POP/CALL/RET instructions

- Enables subroutine calls with parameters
- R7 as Stack Pointer
- Professional function call support

- [ ] **Fast Path Optimization** (3% bonus): Bypass Memory stage for simple instructions
  - LDI instructions: 5→3 cycles
  - ALU-only operations: 40% latency reduction
- [ ] **Instruction-Level Parallelism** (5% bonus): Multiple independent instructions per cycle
  - Advanced out-of-order execution concept
  - Dependency analyzer for safety
- [ ] **Status Flag History** (2% bonus): Track flag state changes
  - More complex conditional logic
- [ ] **Dynamic Clock Scaling** (3% bonus): Adjust clock based on instruction type
  - Complex instructions: slower clock
  - Simple instructions: faster clock

Interrupt Features (10% Bonus)

- [ ] Interrupt request (IRQ) input
- [ ] Interrupt handler routine
- [ ] Return from interrupt (RTI) instruction
- [ ] Interrupt masking/enable-disable
- [ ] Priority handling (if multiple IRQs)

FINAL SUBMISSION TEMPLATE

Due: November 2nd

Submit as ZIP file containing:

```
CPU_Project/
├── CircuitFiles/
│   ├── halfAdder.dig
│   ├── fullAdder.dig
│   ├── adder8bit.dig
│   ├── alu8bit.dig
│   ├── registerFile8.dig
│   ├── programCounter8.dig
│   ├── programROM.dig
│   ├── dataRAM.dig
│   ├─── FD_Register.dig
│   ├─── DE_Register.dig
│   ├─── EM_Register.dig
│   ├─── MW_Register.dig
│   ├─── controlUnit.dig
│   ├─── forwardingUnit.dig (optional)
│   ├─── interruptPriority.dig (optional)
│   └─── CPU_8bit.dig (MAIN FILE)
│
├─── Documentation/

│   ├─── CPU_Design_Report.pdf (5-10 pages)
│   ├─── ISA_Reference.txt
│   ├─── Assembly_Programs.txt
│   └─── Screenshots/
│       ├─── architecture_diagram.png
│       ├─── simulation_fetch.png
│       ├─── simulation_execute.png
│       └─── simulation_output.png
│
├─── Video/
│   └─── CPU_Demo.mp4 (5-10 minutes)
│
└─── README.txt
  (Brief instructions on how to run the CPU)
```

README.txt content:

```
8-Bit Pipelined CPU Project


Team Members: [Names]
Date: [Submission Date]


How to Run:
1. Open CPU_8bit.dig in Digital simulator
2. Click Play to start simulation
3. Use Step button to advance clock cycles
4. Monitor PC, registers, and output port
5. See Architecture_diagram.png for overall design


Key Features:
```

```
- 5-stage pipeline (Fetch, Decode, Execute, Memory, Writeback)
- 8 general-purpose registers (R0-R7)
- Status flags (Zero, Non-Zero, Carry)
- Branch instructions with hazard detection
- Interrupt support (bonus feature)
- Register forwarding optimization (bonus feature)

Test Program:
Assembly code for test program in Documentation/Assembly_Programs.txt
Machine code loaded in programROM.dig


For more details, see CPU_Design_Report.pdf
```

QUICK REFERENCE: Common Issues & Solutions

Issue	Symptom	Solution
PC not incrementing	PC stays at 0	Check: Clock connected to Counter, enable pin to Vcc, dir to Gnd
Instructions not fetching	ROM outputs garbage	ROM not loaded; or Address bits mismatch
Registers not writing	RegFile shows zeros	Check: WriteEnable=1, Clock connected, data timing
ALU always shows 0	Result always zero	Check: Adder inputs connected correctly, OpCode routing
Flags not updating	Always shows 0	Check: NOR gate inputs, FlagWrite pulse, Register clock
Pipeline stalling too much	Very slow execution	Review hazard logic; consider adding forwarding
Branch never taken	Always skips branch	Check: Flag calculation, branch condition logic, BZ opcode
Output doesn't change	Output port stuck	Check: WR instruction encoding, output buffer clock
Simulation crashes	Digital freezes	Likely infinite loop in ROM; load simpler test program
Timing errors	Inconsistent results	Add delays; check clock frequency not too high

---

TIMELINE REMINDER

- **Week 1:** Build adder, ALU, register file (Parts 1-3)