VHDL-MCU-8

Detailed Implementation Manual

Block-Level Behavioral Description

Revision: 2.0
Date: December 11, 2025

1 PROGRAM COUNTER (PC) IMPLEMENTATION

1.1 Functional Overview

The Program Counter (PC) is the central navigation unit of the VHDL-MCU-8 processor. It is an 8-bit synchronous register responsible for generating the address of the next instruction to be fetched from the Program Memory (ROM). The PC does not merely count; it must support sequential execution, relative branching (for loops and conditional logic), and absolute jumping (for subroutine calls and resets).

1.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk`	Input	1	System Clock (Active Rising Edge).
`rst`	Input	1	System Reset (Active High). Forces PC to `0x00`.
`pc_en`	Input	1	PC Write Enable. Asserted by CU during FETCH.
`pc_src`	Input	2	MUX Selector: 00=Inc, 01=Branch, 10=Jump, 11=Zero.
`offset`	Input	8	Signed 8-bit offset for relative branching (k).
`target`	Input	8	Absolute 8-bit target address for jumps/-calls.
`pc_out`	Output	8	Current instruction address (to ROM).

1.3 Behavioral Description

1.3.1 1. Reset State

Upon assertion of the asynchronous `rst` signal, the internal PC register is immediately cleared to `0x00`. This points the processor to the Reset Vector, which is the first location in Program ROM.

1.3.2 2. Next Address Logic (Combinational)

A 4-to-1 Multiplexer determines the `next_pc` value based on the `pc_src` control signal provided by the Control Unit:

- **Normal Operation (00):** The default behavior. An internal adder computes $PC_{current} + 1$. This occurs during standard sequential instruction execution.
- **Relative Branching (01):** Used for `RJMP`, `BRBS`, `BRBC`. The arithmetic unit adds the current PC to the sign-extended `offset` input (k). Note that since the PC increments during fetch, the effective calculation must be $PC + 1 + k$. The implementation must handle signed addition correctly to allow backward jumps.
- **Absolute Jump (10):** The PC is loaded directly with the `target` input. Used for returns from interrupts or subroutines (`RET`) where the address comes from the Stack or a register.

- **Hard Reset/Clear (11):** Forces `next_pc` to `0x00`. Used for software resets or trap handling.

1.3.3 3. Synchronous Update

On the rising edge of `clk`, if `pc_en` is '1', the calculated `next_pc` is latched into the internal register. If `pc_en` is '0' (e.g., during a multi-cycle EXECUTE stage), the PC holds its current value, stalling the fetch pipeline.

1.4 VHDL Implementation Considerations

- **Adder Width:** The adder for branch calculation ($PC + 1 + k$) should be 8-bit. Overflow is typically ignored in simple MCUs (wrapping behavior), allowing jumps across the 255-0 memory boundary.
- **Signal Types:** Use `signed` types for the relative offset calculation to automatically handle 2's complement math, then cast back to `std_logic_vector`.

2 INSTRUCTION REGISTER (IR) IMPLEMENTATION

2.1 Functional Overview

The Instruction Register (IR) serves as the pipeline barrier between the memory system and the execution core. It is a 16-bit storage element that latches the word fetched from Program ROM. By holding the instruction stable, it allows the Control Unit and ALU to process the data purely combinationally during the EXECUTE phase, even as the PC prepares the address for the next cycle.

2.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk`	Input	1	System Clock.
`ir_load`	Input	1	Write Enable. Active during FETCH state.
`rom_data`	Input	16	Raw instruction word from Program ROM.
`opcode`	Output	5	Bits 15:11 (Operation Code).
`rd_addr`	Output	3	Bits 10:8 (Destination Register Index).
`rs_addr`	Output	3	Bits 7:5 (Source Register Index).
`imm_val`	Output	8	Bits 7:0 (Immediate Constant / Offset).

2.3 Behavioral Description

2.3.1 1. The Fetch Cycle

The IR is updated only when the Control Unit is in the **FETCH** state. The signal `ir_load` is asserted, and on the rising clock edge, the 16-bit value currently present on the `rom_data` bus is copied into the IR. This value represents the machine code for the operation to be performed.

2.3.2 2. Hardwired Decoding (Bit Slicing)

Unlike the PC, the IR does not perform logic. Its primary behavior is structural "slicing." The 16-bit storage is continuously fanned out to specific subsystems based on the fixed instruction format of the VHDL-MCU-8. This parallelism is critical for single-cycle execution:

- **Opcode (15 downto 11):** Sent immediately to the Control Unit to drive the Finite State Machine (FSM).
- **Rd (10 downto 8):** Sent to the Register File's "Write Address" port and "Read Address Port B".
- **Rs (7 downto 5):** Sent to the Register File's "Read Address Port A".
- **Immediate (7 downto 0):** Sent to the ALU (Multiplexer B) and the PC Adder.

2.3.3 3. Data Stability

Once the processor transitions from FETCH to EXECUTE, `ir_load` is de-asserted. The IR output remains stable regardless of changes on the `rom_data` bus (which might be changing as the PC increments). This stability ensures no glitches occur in the control signals or ALU operands during the critical path of execution.

2.4 VHDL Implementation Considerations

- **Aliases:** VHDL `alias` commands are highly recommended to name the slices (e.g., `alias opcode is ir_reg(15 downto 11);`). This makes the code significantly more readable than using magic numbers throughout the architecture.
- **Reset:** While the IR does not strictly require a reset (since it will be loaded with the first instruction at address 0x00), it is good practice to reset it to `NOP` (0x0000) to prevent spurious control signals during system startup.

3 REGISTER FILE (REGFILE) IMPLEMENTATION

3.1 Functional Overview

The Register File is a fast, multi-ported Static RAM block embedded within the CPU core. It provides temporary high-speed storage for operands and results. It consists of eight 8-bit registers ($R0 - R7$). To support single-cycle execution of instructions like `ADD Rd, Rs`, the block must support simultaneous access to two source operands and one destination write within a single clock period.

3.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk`	Input	1	System Clock.
`we`	Input	1	Write Enable.
`addr_a`	Input	3	Read Address A (Selects Source Rs).
`addr_b`	Input	3	Read Address B (Selects Dest Rd for reading).
`addr_w`	Input	3	Write Address (Selects Dest Rd for writing).
`data_in`	Input	8	Data to be written (from ALU or RAM).
`data_a`	Output	8	Data output from Port A.
`data_b`	Output	8	Data output from Port B.

3.3 Behavioral Description

3.3.1 1. Storage Element

The core storage is an array type: `array(0 to 7) of std_logic_vector(7 downto 0)`. This structure is mapped by synthesis tools to distributed RAM (LUT-based RAM) or discrete flip-flops depending on the FPGA architecture and size.

3.3.2 2. Asynchronous Read (Combinational)

The read operation is asynchronous. Changes on `addr_a` or `addr_b` immediately propagate to `data_a` and `data_b` respectively. This is crucial for the ALU to receive valid data early in the EXECUTE cycle.

$$Data\_A \Leftarrow Registers(to\_integer(addr\_a));$$

3.3.3 3. Synchronous Write

Writing is synchronous. If `we` is high on the rising edge of `clk`, the value on `data_in` is stored into the register selected by `addr_w`.

$$Registers(to\_integer(addr\_w)) \Leftarrow data\_in;$$

3.3.4 4. Transparency / Forwarding

A critical implementation detail is "Read-After-Write" behavior. If the write address matches a read address in the same cycle, the design must define whether the output reflects the *old* value or the *new* value. For this 2-stage pipeline, the Read occurs in the combinational phase of the cycle, and the Write occurs at the very end (clock edge). Therefore, the instruction reads the *current* (old) value, computes the result, and overwrites it at the clock edge.

3.4 VHDL Implementation Considerations

- **Reset:** General Purpose Registers are typically *not* reset in hardware to save resources. Software code is expected to initialize registers before use. However, for simulation purposes, initializing the array to zero is helpful.
- **Synthesis Hint:** Ensure the read addresses are not registered; otherwise, the pipeline would require an extra "Read" stage, breaking the single-cycle execution model.

4 ARITHMETIC LOGIC UNIT (ALU) IMPLEMENTATION

4.1 Functional Overview

The ALU is the computational engine of the processor. It is a strictly combinational block (no clock, no memory). It accepts two 8-bit operands and a 4-bit operation code, producing an 8-bit result and a set of status flags (Z, C, N, V, H).

4.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`op_a`	Input	8	Operand A (from RegFile or Immediate).
`op_b`	Input	8	Operand B (from RegFile).
`alu_sel`	Input	4	Operation Selector (decoded from Opcode).
`cin`	Input	1	Carry In (from SREG, for ADC/SBC).
`result`	Output	8	The computed 8-bit result.
`flags`	Output	5	Status Flags: Z, C, N, V, H.

4.3 Behavioral Description

4.3.1 1. Operation Logic

The ALU is implemented as a large `CASE` statement or multiplexer tree based on `alu_sel`.

- **Arithmetic:** ADD, SUB, ADC (Add with Carry), SBC (Subtract with Carry). These require an internal 9-bit adder structure to capture the Carry Out.
- **Logic:** AND, OR, XOR, COM (1's complement), NEG (2's complement).
- **Transfer:** MOV (Pass Through B), LDI (Pass Through A).

4.3.2 2. Flag Generation

Flag generation is arguably more complex than the result calculation. Flags must be valid for the specific operation performed:

- **Z (Zero):** Set to '1' if the 8-bit `result` is "00000000". Implemented using a NOR reduction of all result bits.
- **N (Negative):** Direct copy of `result(7)` (the MSB).
- **C (Carry):**
  - For ADD: Bit 8 of the internal (9-bit) addition.
  - For SUB: Logic '1' if `op_b` > `op_a` (Borrow).
  - For Logic Ops: Usually cleared or left unchanged (specification dependent).
- **V (Overflow):** Indicates 2's complement overflow.

$$V = (A_7 \cdot B_7 \cdot \overline{R_7}) + (\overline{A_7} \cdot \overline{B_7} \cdot R_7) \quad \text{(For ADD)}$$

- **H (Half Carry):** Carry from bit 3 to bit 4. Required for BCD arithmetic.

4.4 VHDL Implementation Considerations

- **Optimization:** Synthesizers are good at optimizing arithmetic. It is often better to describe behavior (e.g., `Res <= A + B`) rather than building gate-level adders.
- **Flag Masking:** Not all instructions update all flags (e.g., `MOV` usually doesn't affect flags, while `ADD` does). The ALU generates "raw" flags; the Control Unit or SREG must decide whether to latch them. Alternatively, the ALU can output a mask indicating which flags are valid.

5 STATUS REGISTER (SREG) IMPLEMENTATION

5.1 Functional Overview

The SREG is a specialized 8-bit register that maintains the processor's context state. It aggregates the flags from the ALU and control bits for interrupts. It is the decision-making source for conditional branches. Unlike general registers, individual bits in the SREG have distinct meanings and distinct update rules.

5.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk`	Input	1	System Clock.
`rst`	Input	1	System Reset (Clears I-flag).
`alu_flags`	Input	5	Flags input from ALU (Z, C, N, V, H).
`flag_we`	Input	1	Write Enable for ALU flags.
`set_bit`	Input	3	Bit index to set (for BSET/SEI).
`clr_bit`	Input	3	Bit index to clear (for BCLR/CLI).
`sreg_out`	Output	8	Current SREG value.

5.3 Behavioral Description

5.3.1 1. Bit Definitions

- **Bit 7 (I):** Global Interrupt Enable.
- **Bit 6 (T):** Bit Copy Storage (used by BST/BLD instructions).
- **Bit 5 (H):** Half Carry.
- **Bit 4 (S):** Sign Bit ($S = N \oplus V$).
- **Bit 3 (V):** 2's Complement Overflow.
- **Bit 2 (N):** Negative.
- **Bit 1 (Z):** Zero.
- **Bit 0 (C):** Carry.

5.3.2 2. Update Logic

The SREG has complex write logic compared to a standard register:

1. **Arithmetic Update:** When `flag_we` is asserted by the Control Unit (e.g., after an ADD instruction), bits 0-5 are updated with `alu_flags`. Bits 6 and 7 are usually preserved.
2. **Bit Manipulation:** Instructions like `SEI` (Set Global Interrupt) or `CLC` (Clear Carry) target specific bits. The implementation requires logic to Set or Clear individual bits based on the instruction opcode.
3. **T-Bit Operations:** The `BST` instruction copies a specific bit from a general register into the T-flag. The `BLD` instruction does the reverse.

5.3.3 3. Reset Behavior

On Reset, the I-flag (Global Interrupt) must be cleared to '0' to disable interrupts during initialization. Other flags are typically indeterminate or cleared, depending on the specific MCU specification (clearing all to 0 is the safest implementation).

5.4 VHDL Implementation Considerations

- **Process Structure:** A single clocked process is best. Inside, use an `IF/ELSIF` structure to prioritize Reset, then Specific Bit Ops, then ALU Flag Updates.
- **Branching:** The `sreg_out` is sent to the Control Unit. The CU uses a MUX (controlled by the instruction's condition field) to select one bit (e.g., Z) to determine if a `BRNE` (Branch if Not Equal/Zero) should be taken.

6 CONTROL UNIT (CU) IMPLEMENTATION

6.1 Functional Overview

The Control Unit is the "conductor" of the processor. It translates the Opcode stored in the IR into precise electrical signals that orchestrate the datapath. It is implemented as a Finite State Machine (FSM) combined with a complex Combinational Decoder.

6.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk/rst`	Input	1	Clock and Reset.
`opcode`	Input	5	From IR(15:11).
`sreg`	Input	8	From SREG (for conditional branches).
`pc_src`	Output	2	Selects PC Mux source.
`alu_sel`	Output	4	Selects ALU operation.
`reg_we`	Output	1	Register File Write Enable.
`mem_we`	Output	1	Data RAM Write Enable.
`ir_load`	Output	1	IR Latch Enable.

6.3 Behavioral Description

6.3.1 1. Finite State Machine (FSM)

The FSM follows a Moore architecture (outputs depend only on current state), although the Next State logic depends on inputs.

- **State FETCH:**
  - **Action:** Assert `ir_load`. Configure PC to increment (`pc_src="00"`).
  - **Next State:** Always transitions to `EXECUTE`.
- **State EXECUTE:**
  - **Action:** Decode Opcode. Assert `alu_sel`, `reg_we`, etc. Check SREG for branches.
  - **Next State:**
    - For single-cycle ops (ADD, MOV): Transition back to `FETCH`.
    - For multi-cycle ops (LD, ST, RET): Transition to `EXECUTE_2`.
- **State EXECUTE_2 (Optional):** Handles the second cycle of memory access or stack operations. Transitions back to `FETCH`.

6.3.2 2. Instruction Decoder

A large Combinational Process acts as a Look-Up Table (LUT).

- **Input:** 5-bit Opcode.
- **Logic:**

```
    case opcode is
      when OP_ADD =>
          alu_sel <= ALU_ADD; reg_we <= '1'; flag_we <= '1';
      when OP_ST =>
          mem_we <= '1'; -- Write to RAM
      when OP_BRBS =>
          if sreg(bit_index) = '1' then
             pc_src <= "01"; -- Branch
          end if;
      ...
    end case;
```

6.4 VHDL Implementation Considerations

- **Default Assignments:** To prevent "Latch Inference" (a common VHDL error), assign default values to all control signals at the start of the combinational process (e.g., `reg_we <= '0';`). Only override them in the specific `CASE` statements.
- **Timing:** The Control Unit is typically the critical path. Ensure the decoding logic is as flat as possible to maximize clock frequency.

7 DATA RAM (MEMORY INTERFACE) IMPLEMENTATION

7.1 Functional Overview

The Data RAM is a 256-byte storage block separate from the Program ROM (Harvard Architecture). While the RAM block itself is often an IP core or inferred array, the *Interface* logic surrounding it is critical. It must handle address selection between immediate addresses (Static) and register pointers (Dynamic/Indirect).

7.2 Port Interface

CorpBlue!20 **Signal Name**	**Direction**	**Width**	**Description**
`clk`	Input	1	System Clock.
`mem_we`	Input	1	Write Enable (from CU).
`addr_mode`	Input	1	0=Direct (Immediate), 1=Indirect (Pointer).
`imm_addr`	Input	8	Address from IR(7:0).
`ptr_addr`	Input	8	Address from X-Register (R7).
`data_w`	Input	8	Data to Write (from RegFile Rd).
`data_r`	Output	8	Data Read (to RegFile WB Mux).

7.3 Behavioral Description

7.3.1 1. Address Multiplexing

Before the RAM can be accessed, the address must be resolved.

- **Direct Addressing (LDS/STS):** The address is embedded in the instruction. The CU selects `imm_addr` (IR 7:0) as the RAM address.
- **Indirect Addressing (LD/ST):** The address is stored in a register (typically R7, acting as the X-pointer). The CU selects `ptr_addr` (RegFile Port A output) as the RAM address.

7.3.2 2. Read Operation (Load)

For a Load instruction:

1. Address is presented to the RAM.
2. `mem_we` is held Low ('0').
3. RAM output `data_r` becomes valid after the access time.
4. This data is routed to the Register File Write Input. The CU asserts `reg_we` to latch this data into the destination register.

7.3.3 3. Write Operation (Store)

For a Store instruction:

1. Address is presented to the RAM.

2. Data from the source register (RegFile Port B) is presented to `data_w`.
3. `mem_we` is asserted High ('1') for one clock cycle.
4. Data is written on the rising clock edge.

7.4 VHDL Implementation Considerations

- **Synchronous RAM:** FPGA block RAM is strictly synchronous. This means a Read takes 1 clock cycle.
- **Pipeline Impact:** Because the Read takes 1 cycle, a `LD` instruction cannot complete in a single EXECUTE cycle. The data is only available at the *end* of the cycle.
- **Solution:** The `LD` instruction must stall the pipeline or use a 2-cycle EXECUTE state. In cycle 1, the address is setup. In cycle 2, the data is available at the RAM output and written to the Register File.