TinyRISC Documentation

people

June 2025

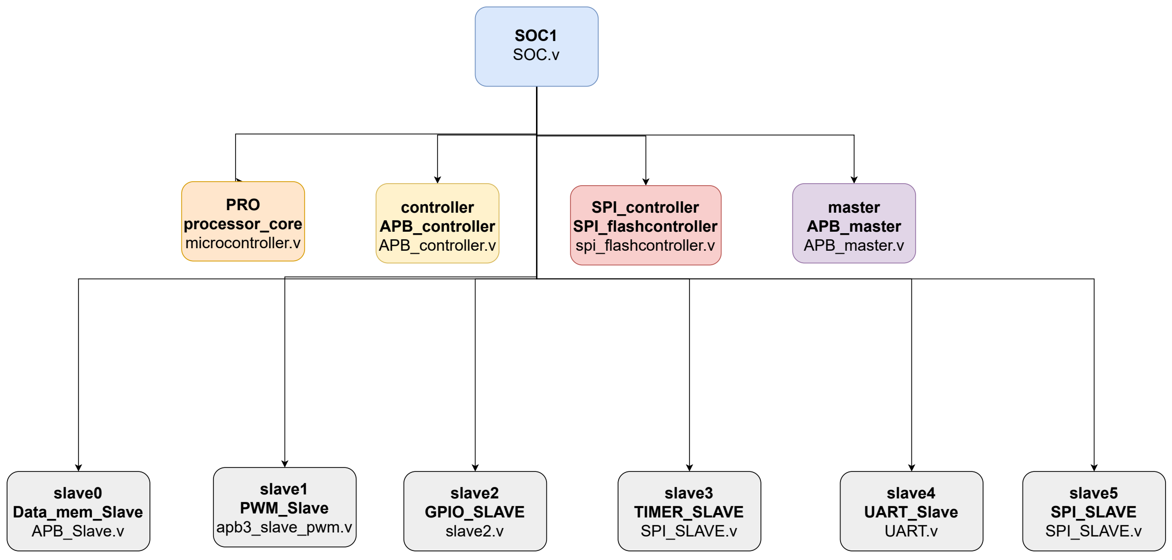
Contents

1 Overview

The design features a top-level module called **SoC1**, which integrates the complete processing system.
This includes:

- The **Processor Core**
- An **APB Master** for initiating peripheral transactions
- An **APB Controller** for coordinating APB protocol
- A **SPI Flash Controller** for program loading from external flash
- APB peripherals such as: **Timer, UART, SPI, PWM**, and internal memory

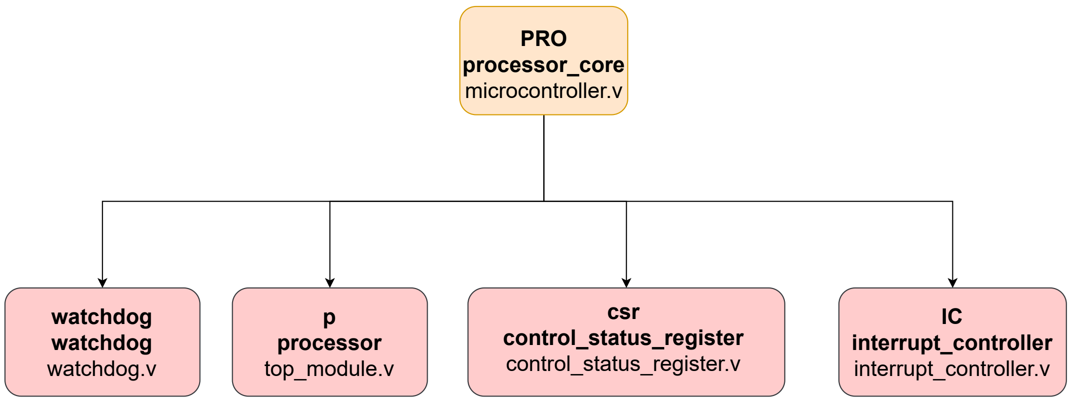
SoC1 Block Diagram:



Within the Processor Core, the following modules are instantiated:

- **p_processor_top**: The main top-level module for the pipelined processor
- **watchdog**: Ensures timely resets for safety
- **csr**: Control and status register module
- **interrupt_controller**: Handles external and internal interrupts

Processor Core Block Diagram:



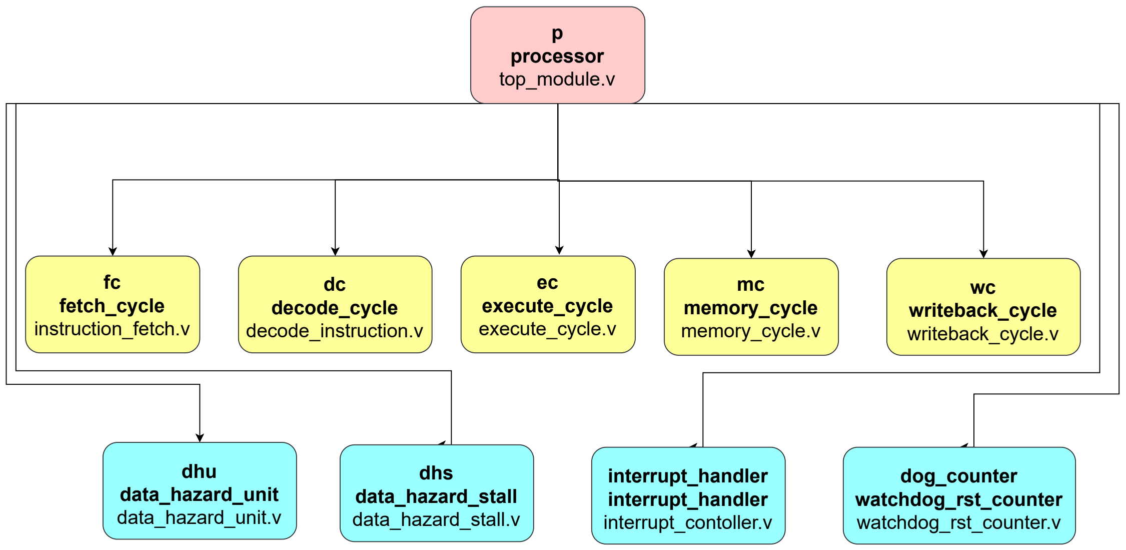
Inside `p_processor_top`, the processor implements a **5-stage pipeline**:

1. **Fetch (IF)**
2. **Decode (OF)**
3. **Execute (EX)**
4. **Memory Access (MA)**
5. **Write Back (WB)**

Additional internal modules include:

- **Data Hazard Unit** – Handles forwarding logic
- **Data Stall Unit** – Stalls pipeline during unresolved hazards
- **Dog Counter** – Supports watchdog reset logic
- **Interrupt Handler** – Coordinates pipeline flushes on interrupt

Processor Pipeline Block Diagram:



This pipeline architecture enables overlapped execution of instructions and efficient utilization.

1.1 SoC Control and Reset Signals

- **reset:** Power-On Reset (External Reset)

  Source: Comes from external.

  Use: Passed to SPI flash controller to reset it.

  Behavior: Likely active-high. Resets the system during initial boot.

  Purpose: Ensures SPI flash and programming logic are reset when power is first applied.

- **rst:** Controlled Reset for Processor and APB Peripherals

  Source: Internal signal, generated inside the SoC.

  Generated As: Set high on `posedge prg_mode_top`, cleared on `negedge clk_top`.

  Scope: Connected to processor, SPI slave, I2C slave, and PWM.

  Purpose: Programmable reset after programming is done.

1.2 Clocks and Reset

- **clk:** To processor — 1-bit gated clock signal (`clk_top && prg_mode_top`) that runs processor only after memory programming is done.
- **rst:** To processor — 1-bit reset signal triggered on `posedge prg_mode_top`.
- **ext_clk:** To processor — 1-bit ungated external clock (`clk_top`) for writing into instruction memory from flash.

1.3 Instruction Flash Memory Processor

- **instruction:** To processor — 32-bit instruction fetched from SPI flash.
- **dummy_pc:** To processor — 32-bit PC value used as read address in SPI flash.
- **write_en:** To processor — 1-bit signal indicating instruction is valid for loading into instruction memory.
- **prg_mode:** To processor — 1-bit programming mode flag. When low, processor stalls and waits for programming.

1.4 Processor APB Interface (Master)

Write Request Signals

- **proc_addr:** From processor — 32-bit address of memory-mapped peripheral.
- **proc_write:** From processor — 1-bit. High = Write, Low = Read.
- **proc_wdata:** From processor — 32-bit data to be sent to APB slave.

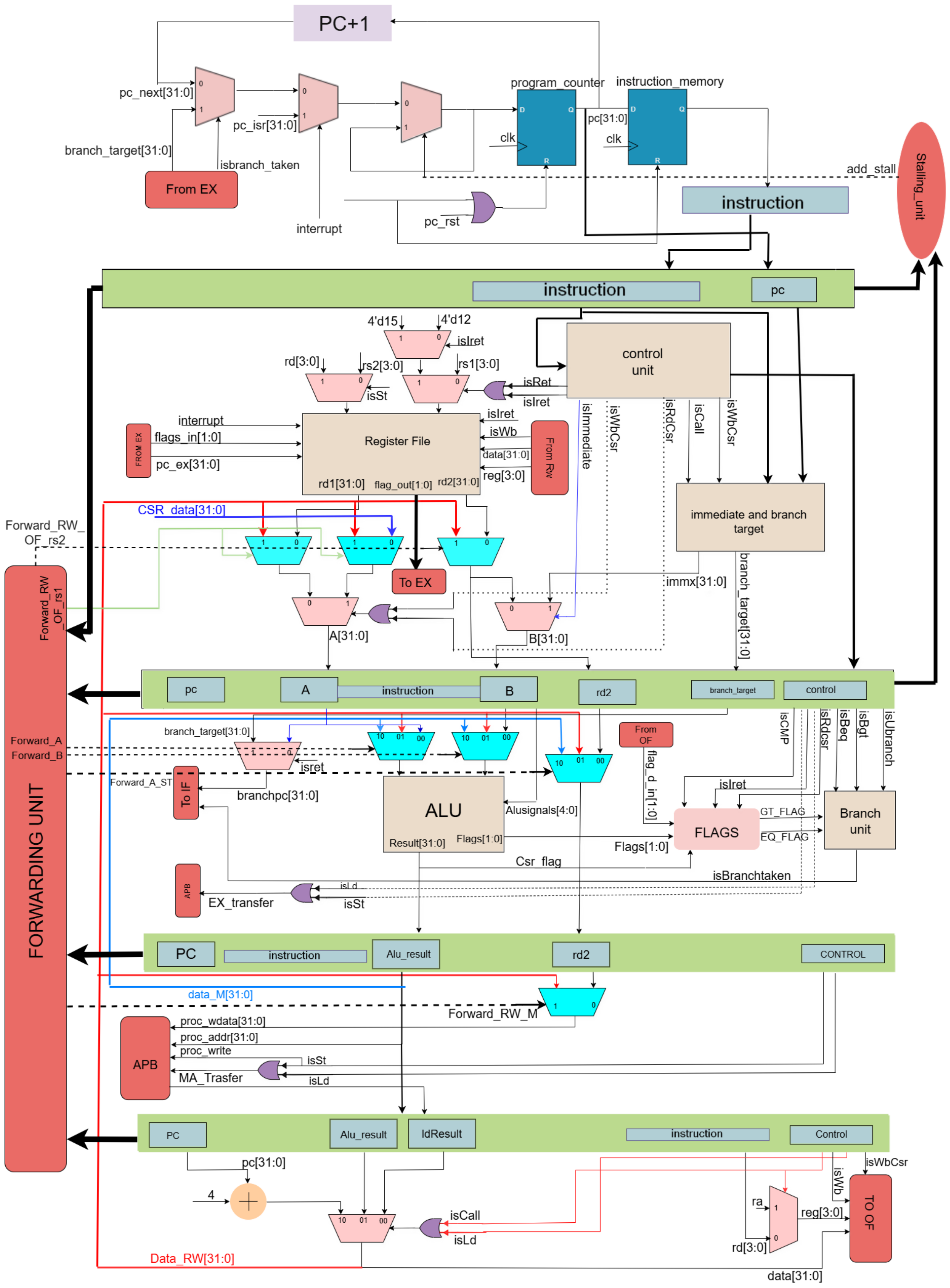
Read Response Signals

- **proc_rdata:** To processor — 32-bit data read from APB slave.
- **APB_ack:** To processor — 1-bit signal indicating transaction completion.

1.5 Processor APB Interface (Controller)

- **EX_transfer:** From processor — 1-bit signal indicating Execute stage APB request.
- **MA_transfer:** From processor — 1-bit signal indicating Memory Access stage APB request.

Processor Pipeline Block Diagram:


PC+1
pc_next[31:0]
pc_isr[31:0]
branch_target[31:0]
isbranch_taken
From EX
interrupt
program_counter
instruction_memory
pc[31:0]
clk
pc_rst
add_stall
instruction
Stalling_unit
instruction
pc
4'd15
4'd12
isIret
rd[3:0]
rs2[3:0]
rs1[3:0]
isSt
isRet
control unit
interrupt
flags_in[1:0]
pc_ex[31:0]
FROM EX
Register File
rd1[31:0]
flag_out[1:0]
rd2[31:0]
isWb
data[31:0]
reg[3:0]
From RW
isImmediate
isWbCsr
isRdCsr
isCall
immediate and branch target
CSR_data[31:0]
Forward_RW_OF_rs2
Forward_RW_OF_rs1
To EX
immx[31:0]
branch_target[31:0]
A[31:0]
B[31:0]
pc
A
instruction
B
rd2
branch_target
control
Forward_A
Forward_B
Forward_A_ST
isret
To IF
branchpc[31:0]
From OF
flag_d_in[1:0]
isCMP
isRdcsr
isBeq
isBgt
isUbranch
ALU
Alusignals[4:0]
Result[31:0]
Flags[1:0]
FLAGS
GT_FLAG
EQ_FLAG
Branch unit
Csr_flag
isBranchtaken
APB
isLd
EX_transfer
isSt
FORWARDING UNIT
PC
instruction
Alu_result
rd2
CONTROL
data_M[31:0]
Forward_RW_M
proc_wdata[31:0]
proc_addr[31:0]
proc_write
MA_Trasfer
PC
Alu_result
ldResult
instruction
Control
pc[31:0]
4
isCall
isLd
Data_RW[31:0]
ra
rd[3:0]
reg[3:0]
isWb
isWbCsr
TO OF
data[31:0]

2 SPI Flash Controller

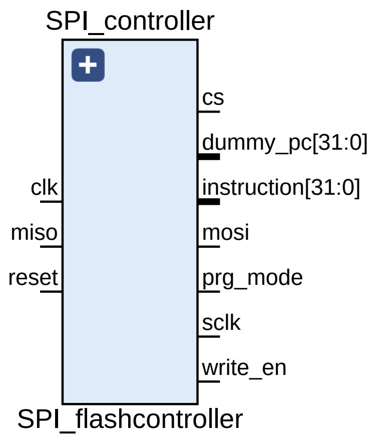


Figure 1: Block Diagram flash controller module

2.1 IDLE State

The FSM enters the **IDLE** state after a rst, which is triggered on the negative edge of the reset signal (power on reset or global reset). In this state, all control signals are initialized. The chip select (CS) is set high to disable the SPI chip. Additionally, *mosi_en*, *sclk_en*, and *buffer_en* are all deasserted. Internal variables like *Dummy_Pc* and the *timer* are reset to zero, and the *prg_mode* signal is deasserted.

The *shift_reg* register is initialized with the 32-bit value *0x03000000*, which represents a standard SPI Read command followed by the target address (24-bit 0). After completing these initializations, the FSM proceeds to the **START** state.

2.2 START State

In the **START** state, CS is driven low to enable communication with the SPI device, and the SPI clock enable signal (*sclk_en*) is asserted to begin clocking data. The contents of *shift_reg* are then serially transmitted over the MOSI line.

The most significant bit (MSB) of *shift_reg* is assigned to the MOSI output. On each negative edge of the SPI clock, the register is left-shifted by one bit to prepare the next bit for transmission. This shifting process continues until the *timer* reaches 32, indicating that the entire command sequence has been sent. At this point, the FSM transitions to the **READ** state.

2.3 READ State

When the FSM enters the **READ** state, it begins receiving data from the SPI flash memory through the MISO line. Incoming bits are shifted into *shift_reg*. When the *timer* reaches 1 and the *start_hap* signal is deasserted (used to ensure that memory is not written during the first received word), the current content of *shift_reg* is copied to *data_buffer*, and *write_en* is cleared to prevent premature data writing.

When the *timer* reaches 2 and *start_hap* is still deasserted, the contents of *data_buffer* are considered valid. At this point, the value is assigned to the *instruction* output, the *Dummy_Pc* counter is updated, and *write_en* is asserted to indicate valid data. If *start_hap* is still asserted at $timer = 2$ (so that writing starts after the first received word), it is cleared to avoid incorrect writes during the initial cycle. The write signal is kept high for 31 cycles, but a latch is used on the instruction memory side to write to memory at the first negative edge after data is received. Then the latch is turned off and re-enabled at the next positive edge of *write_en*.

The FSM continues to collect data in this state until *data_buffer* holds the value *0xFFFFFFFF* (the End Of File instruction), signaling the end of valid data. Once this condition is met, the FSM transitions to the **END** state.

2.4 END State

The **END** state is entered when the SPI flash returns the value *0xFFFFFFFF*, indicating that there is no more valid data to be read. In this final state, *CS* is pulled high to disable the SPI device, and *sclk_en* is deasserted to stop the SPI clock. The *write_en* signal is also cleared.

Additionally, the *prg_mode* signal is asserted at the negedge of the clk to indicate that the system may now enter a program mode or start the processor. The FSM halts in this state and does not transition to any other state thereafter.

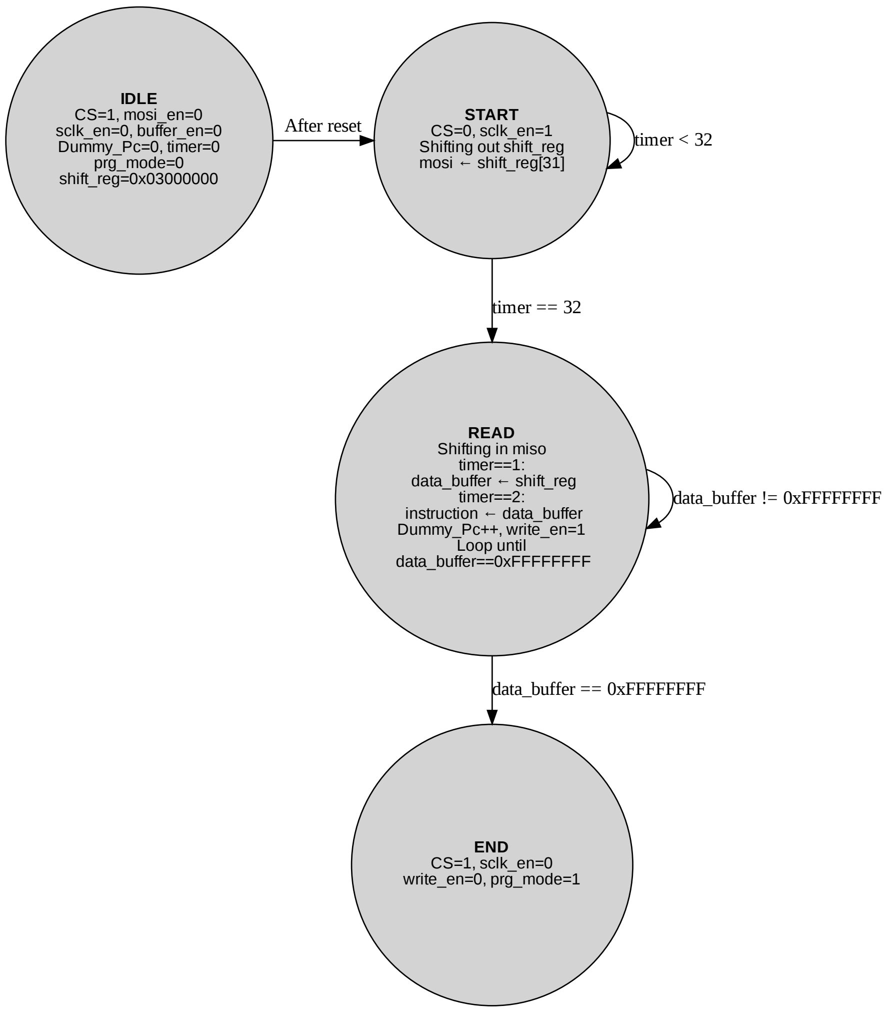


Figure 2: FSM of Flash Controller

3 Fetch Unit

The fetch stage is the first stage in a classic instruction pipeline, responsible for retrieving the next instruction from memory and preparing the processor for subsequent execution steps. In a typical five-stage pipeline (IF, ID, EX, MEM, WB), the fetch stage (IF) ensures a continuous flow of instructions, maximizing instruction-level parallelism and processor throughput.

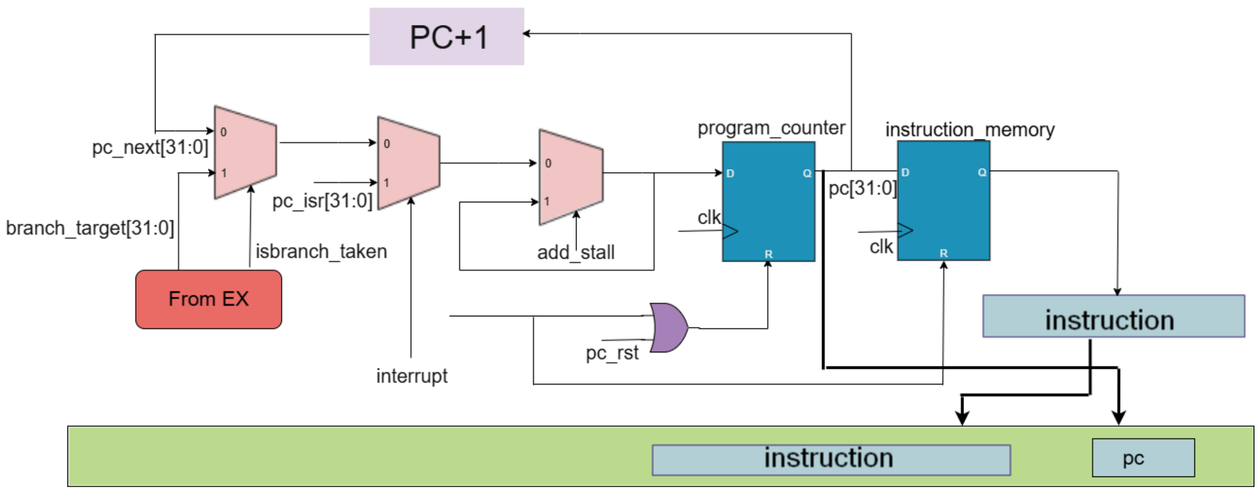


Figure 3: Block Diagram of the Fetch Unit

3.1 Program Counter Module

The `program_counter` module holds the address of the next instruction to be fetched and executed. This ensures that the program counter (PC) is updated correctly in response to various control signals such as resets, stalls, interrupts, and normal sequential execution.

1. Reset Logic

- If any of the reset signals (`rst`, `pc_rst`) are asserted (high), the PC is set to `0x40`.
- `pc_rst` comes from the watchdog module and is used to reset the processor whenever the PC is stuck.

2. Stall Handling

- If a pipeline stall is requested (`add_stall` is high), the PC holds its current value.
- This prevents the pipeline from fetching the next instruction.
- The `add_stall` signal comes from `data_hazard_stall` whenever a load-use hazard is detected.

3. Interrupt Handling

- If an interrupt is detected, the PC is set to the ISR address (`pc_isr`).
- This allows the processor to immediately jump to the address of the interrupt service routine (ISR).

4. Normal Operation

- In the absence of resets, stalls, or interrupts, the PC is updated to `pcnext`.

3.2 PC for Branch or Call

When `isbranchtaken_E` = 1, the multiplexer redirects PC to the branch target address; otherwise, it selects the next sequential address.

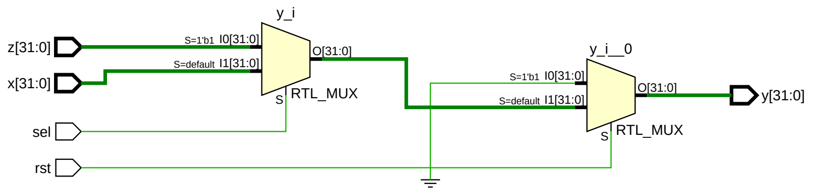


Figure 4: Block Diagram of the Unit to select pc according to branching

Multiplexer Signal Mapping

The multiplexer implements the following logic:

Input	Description
`x`	`pc_next` — Next sequential PC (e.g., PC + 1)
`z`	`pc_branchTarget` — Target address for taken branch or call
`sel`	`isBranchTaken_E` — Select signal indicating branch is taken

3.3 Instruction Memory

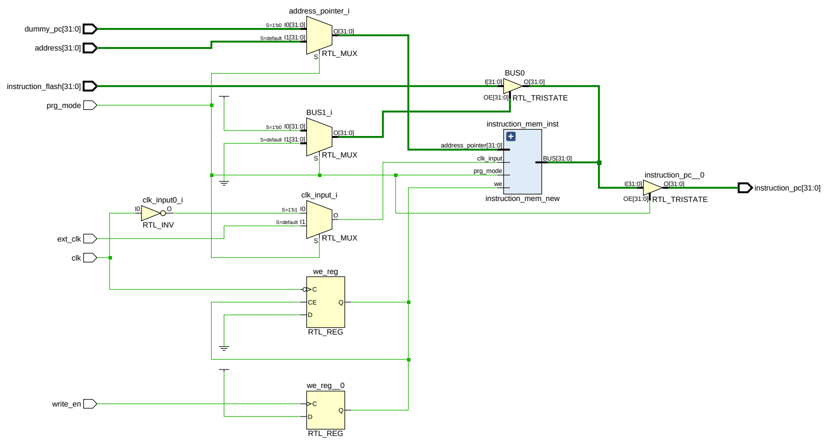


Figure 5: Block Diagram Instruction memory and interface with flash

1. instruction_mem_flash Module

- **Dual-Mode Operation:**
  - `prg_mode = 0`: Programming Mode
    - Writes instructions from SPI flash to memory
    - Uses `dummy_pc` as write address
    - Clocks with inverted processor clock (˜clk)
    - Reading from flash at negedge of clock
  - `prg_mode = 1`: Execution Mode
    - Reads instructions for processor execution
    - Uses address from program counter
    - Clocks with external clock (`ext_clk`)
    - Forwards instructions to processor at posedge of clock

2. Bus Direction Control
Bus is a tristate buffer driven by `prg_mode`.

- **Programming Mode** (`prg_mode = 0`):
  - Bus is driven by `instruction_flash` (external flash data)
  - Functions as input bus for memory writes
  - Transfers instructions from SPI flash to memory
- **Execution Mode** (`prg_mode = 1`):
  - Functions as output bus for memory reads
  - Allows `instruction_mem_new` to drive instructions onto the bus

3.4 Pipeline Registers

The pipeline register between the Instruction Fetch (IF) and Operand Fetch (OF) stages stores essential data and control signals required for decoding and later execution. It enables smooth data flow between stages and supports control mechanisms like reset, interrupt handling, and branch resolution.

IF–OF Pipeline Register Signals:

1. **instruction [31:0]** – Holds the 32-bit instruction fetched from memory.
2. **pc [31:0]** – Holds the program counter value corresponding to the fetched instruction.
3. **rst** – When asserted, clears the pipeline register (flushes contents), typically inserting a NOP.
4. **isBranchtaken_E** – If asserted, flushes the IF–OF register due to a branch being taken in the Execute stage.
5. **interrupt** – When high, causes the IF–OF register to flush and a NOP to be inserted, allowing the pipeline to redirect to the interrupt service routine (ISR).

Explanation:
The IF–OF register temporarily stores the instruction and its corresponding PC between the fetch and decode phases. It includes control logic to handle situations where the current instruction should be discarded or stalled:

- On `rst`, `interrupt`, or `isBranchtaken_E`, the contents are cleared to prevent incorrect execution.
- Under normal operation, the register captures the instruction and PC on every clock cycle.

These values are then passed to the Operand Fetch (OF) stage as `instruction_D` and `pc_D`.

4 Decode Cycle

This module implements the decode stage of a pipelined processor, handling instruction decoding, operand preparation, hazard detection and forwarding, and control signal generation.

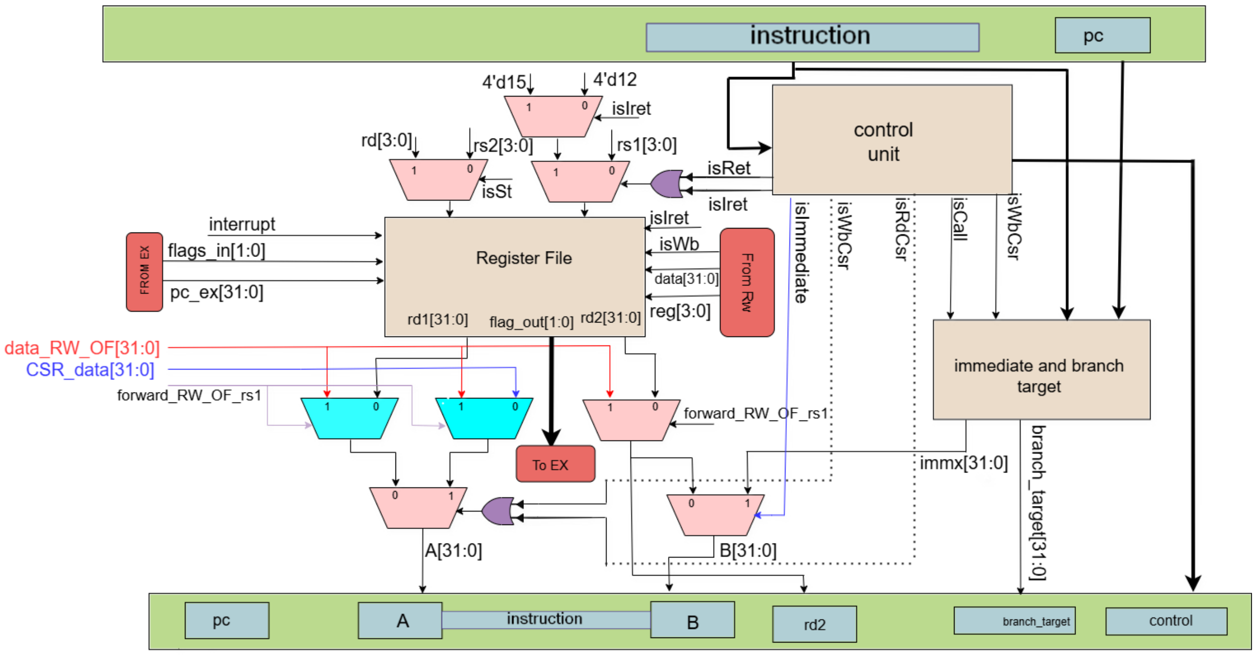


Figure 6: Decode stage and its pipeline

4.1 Register File Module

This 16-register, 32-bit wide register file supports the following features:

- **Dual read ports (rd1 and rd2):** Simultaneously reads data from `rs1` and `rs2`.
- **Single write port:** Writes data to `rd_ra` on the clock edge when the `writeback` signal is high.
- **Specialized registers:**
  - **R12 (Program Counter - PC):** Stores the current value of the program counter when the `interrupt` signal is high.
    `register[12] <= pc_EX`
  - **R13 (Flags Register):** Continuously stores the flags value.
    `register[13] <= {30'b0, flags}`
  - **R14 (Stack Pointer - SP):** Used as the stack pointer.
  - **R15 (Return Address - RA):** Updated whenever there is a call instruction.
- **Flag Output:** When the `iret` signal is high, the saved flags value from `register[13]` is sent via `flag_out`.

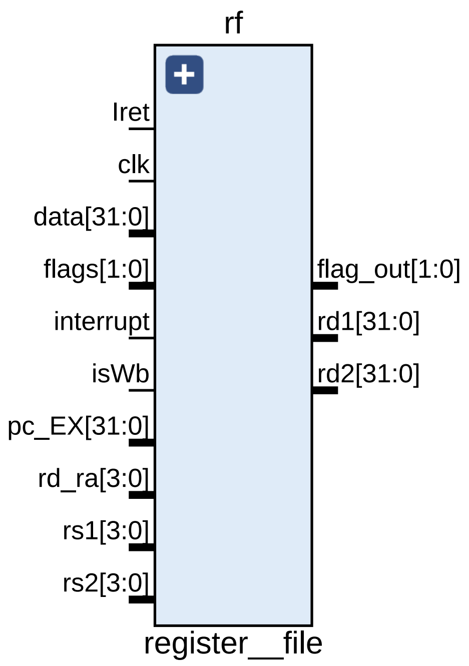


Figure 7: Block Diagram of Register File

4.2 Immediate & Branch PC Generator Module

Branch Target Calculation:

- **Call instructions:** Uses an absolute target derived from a sign-extended 27-bit immediate.
- **Other branches:** Computes a PC-relative target as: `PC + sign-extended 27-bit offset`.

Immediate Generation:
Uses instruction bits `[17:16]` to determine the type of immediate:

- `01` – Zero-extension: `immx = {16'b0, instruction[15:0]}`
- `10` – Upper-half load: `immx = instruction[15:0] << 16`
- `00` – Sign-extension: `immx = {{16{instruction[15]}}, instruction[15:0]}`

Special CSR Handling:
When clearing the CSR, the following immediate formats are used:

- `10` – Upper-half load: `immx = {instruction[15:0], {16{1'b1}}}`
- `00` – Lowerr-half load: `immx = {{16{1'b1}}, instruction[15:0]}`

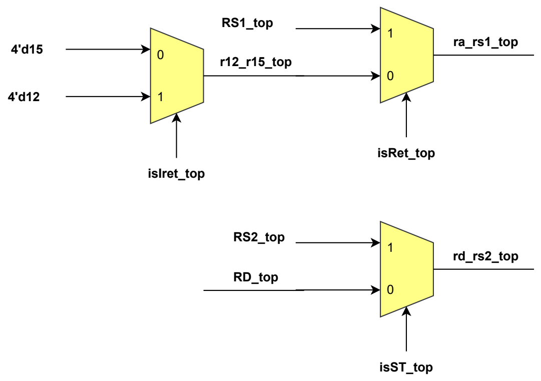


Figure 8: Input the reg file read ports

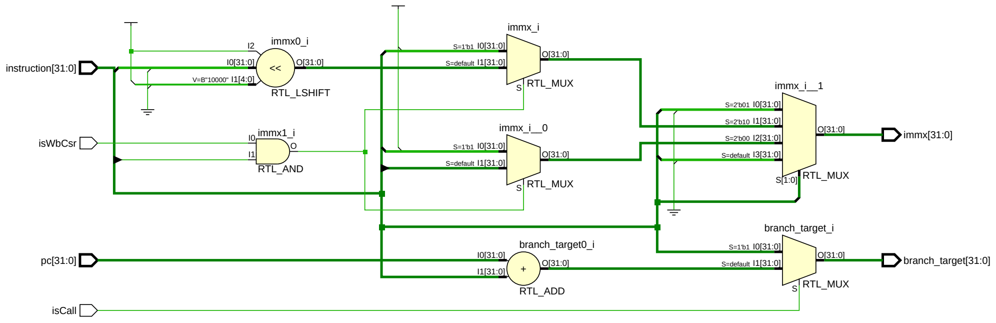


Figure 9: Block Diagram of Immediate and Branch Target generator Unit

4.3 Control Unit Module

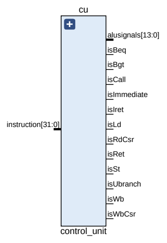


Figure 10: Block Diagram of Control Unit

This module decodes 32-bit instructions to generate 13 critical control signals required for pipeline execution and ALU operations.

Control_signals:

Signal	Description
`isRet`	High when the opcode is `ret` (10100). The PC selects the value popped from the return-address (register `r15`).
`isSt`	High when the instruction is a `store (st)`. Indicates a memory write operation to data memory.
`isWb`	High when the result of the operation must be written back to the general-purpose register file. Active for instructions like `add, sub, and, or, not, mov, lsl, lsr`, and `load`.
`isImmediate`	High when `instruction[26]` is 1, indicating that operand 2 comes from the immediate field instead of `rs2`.
`isBeq`	High when Branch-if-equal (`beq`, opcode 10000). The branch unit compares `rs1` and `rs2`. If equal, it signals the PC logic to jump to the branch target.
`isBgt`	High when Branch-if-greater-than (`bgt`, opcode 10001). Similar to `isBeq` but uses a signed-greater-than comparator.
`isUbranch`	High when Unconditional branch (`b`, opcode 10010). The PC directly jumps to the branch target.
`isLd`	High when Load (`ld`, opcode 01110). Indicates a memory read operation from data memory.
`isCall`	High when Function call (`call`, opcode 10011). Pushes the return address (`PC+1`) onto register `r15`. Forces the PC to the call target.
`isIret`	High when Interrupt-return (opcode 10101). Similar to `isRet`, but also restores processor status and sets the PC to the instruction following the one that was interrupted.
`isWbCsr`	High when the current instruction writes data to control/status registers (CSR).
`isRdCsr`	High when the current instruction reads data from CSR.
`alusignals`	ALU control signals that determine which ALU operation is performed.

4.4 Data Forwarding Muxes in the Decode Stage

1. **Forwarding Mux for Source Register 1 (`rd1_top`) (MUX-M1)**
   - **Inputs:**
     - `rd1_RW_OF` – Data read from the register file (normal path)
     - `data_RW_OF` – Data forwarded from the writeback stage (bypass path)
     - `forward_RW_OF_rs1` – Select signal (1 = forward from writeback, 0 = use register file)
   - **Output:** `rd1_top` (operand for ALU or subsequent logic)
   - **Description:** If source register 1 is awaiting a value to be written back in the current cycle, this mux selects the forwarded value instead of the stale register file value.
2. **Forwarding Mux for Source Register 2 (`rd2_top`) (MUX-M2)**
   - **Inputs:**
     - `rd2_RW_OF` – Data read from the register file
     - `data_RW_OF` – Data forwarded from the writeback stage
     - `forward_RW_OF_rs2` – Select signal (1 = forward, 0 = use register file)

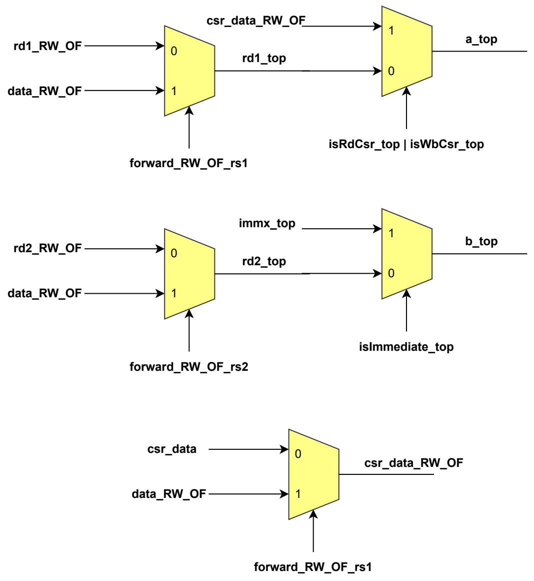


Figure 11: Data forwarding in OF Stage

- **Output:** `rd2_top`
- **Description:** Functions similarly to MUX-M1, but for the second source register.

3. **Forwarding Mux for CSR Data (`csr_data_RW_OF`) (MUX-M3)**
   - **Inputs:**
     - `csr_data` – Data from the Control and Status Register (CSR)
     - `data_RW_OF` – Forwarded data from the writeback stage
     - `forward_RW_OF_rs1` – Select signal (1 = forward, 0 = use CSR data)
   - **Output:** `csr_data_RW_OF`
   - **Description:** Handles CSR data forwarding when the CSR is being updated in the same cycle. *Example:* If `set csr5 1` is in RW stage and `clear csr5 1` is in decode stage.
4. **Immediate Operand Mux (`b_top`) (MUX-M4)**
   - **Inputs:**
     - `rd2_top` – Register value (possibly forwarded)
     - `immx_top` – Immediate value generated from instruction
     - `isImmediate_top` – Select signal (1 = use immediate, 0 = use register value)
   - **Output:** `b_top`
   - **Description:** Selects between a register operand and an immediate value based on the instruction type.
5. **CSR/Operand A Mux (`a_top`) (MUX-M7)**
   - **Inputs:**
     - `rd1_top` – Register value (possibly forwarded)
     - `csr_data_RW_OF` – CSR data (possibly forwarded)

    - `isRdCsr_top | isWbCsr_top` – Select signal (CSR operation = 1, normal = 0)
- **Output:** `a_top`
- **Description:** Selects the correct operand source (CSR or register) for the ALU or further processing during CSR instructions.

4.5 Multiplexers for RS1, RS2, Return, and IRET Selection

1. **Store Instruction MUX (MUX-M2): Selecting between RS2 and RD for Store Operations**
    - **Applicable for:** Store instructions
    - **Inputs:**
        - `RS2_top` – Normal source register 2
        - `RD_top` – Destination register (used as source for store data)
    - **Select Signal:** `isSt_top` (high for store instructions)
    - **Output:** `rd_rs2_top` — selected register index for store data operand
2. **Return Address Register Selector MUX (MUX-M22): Selecting between R15 and R12 for IRET**
    - **Applicable for:** Interrupt return (IRET) and normal return (RET) instructions
    - **Inputs:**
        - `4'd15` – Register 15 (RA) for normal return
        - `4'd12` – Register 12 (PC saved during interrupt) for IRET
    - **Select Signal:** `isIret_top` (high for IRET instructions)
    - **Output:** `r12_r15_top` — selected return address register index
3. **Return Address MUX (MUX-M77): Selecting between RS1 and Return Address for Return Instructions**
    - **Applicable for:** Return instructions
    - **Inputs:**
        - `RS1_top` – Normal source register 1
        - `r12_r15_top` – Selected between R12 or R15 depending on IRET
    - **Select Signal:** `isRet_top` (high for return instructions)
    - **Output:** `ra_rs1_top` — selected register index used as return address

4.5.1 OF–EX Pipeline Register

The Operand Fetch to Execute (OF–EX) pipeline register holds all control and data signals required for execution. It serves as the interface between the decode and execute stages of the processor pipeline, preserving instruction intent and operand values.

OF–EX Pipeline Register Fields:

1. **Control Signals:**
    - `isRet` – Indicates a return instruction.
    - `isSt` – Store instruction flag.
    - `isWb` – Write-back enable signal.
    - `isBeq` – Conditional branch if equal.
    - `isBgt` – Conditional branch if greater.

- `isUbranch` – Indicates an unconditional branch.
- `isLd` – Load instruction flag.
- `isCall` – Indicates a function call instruction.
- `alusignals[4:0]` – ALU control signals to select operation.
- `IsIret` – Indicates an interrupt return instruction.
- `IsRdCsr` – Read from CSR (Control and Status Register).
- `IsWbCsr` – Write-back to CSR.

2. **pc [31:0]** – Program counter forwarded from the OF stage.
3. **instruction [31:0]** – The decoded instruction word.
4. **branch_target [31:0]** – Calculated branch/jump target address.
5. **A [31:0]** – First operand read from register file.
6. **B [31:0]** – Second operand read from register file.
7. **rd2 [31:0]** – The value to be stored in memory during store instructions.
8. **Register Fields:**
   - `RD` – Destination register index.
   - `Ra` – Source register index (useful for CSR or jump instructions).
9. **Control Inputs:**
   - `rst` – Active-high reset. Clears all OF–EX registers.
   - `isBranchtaken_E` – If a branch is taken in EX stage, flush the register.
   - `interrupt` – Forces the pipeline to flush and redirect to ISR.
   - `add_stall` – When asserted, holds the current state (for hazard handling).

Explanation:
The OF–EX register captures all data and control information required by the Execute stage. It ensures correct propagation of:

- Register operands (`A`, `B`, `rd2`)
- PC and instruction metadata
- ALU control signals and special instruction types (CSR, branches, interrupt, etc.)

On every clock edge, the contents are updated unless reset, flushed by branch/interrupt, or stalled due to a data hazard. This stage is critical for enabling precise control flow, hazard resolution, and forwarding logic.

5 Execute Cycle

The execute stage is the third stage in a pipelined processor, responsible for arithmetic/logic operations, branch resolution, and operand forwarding.

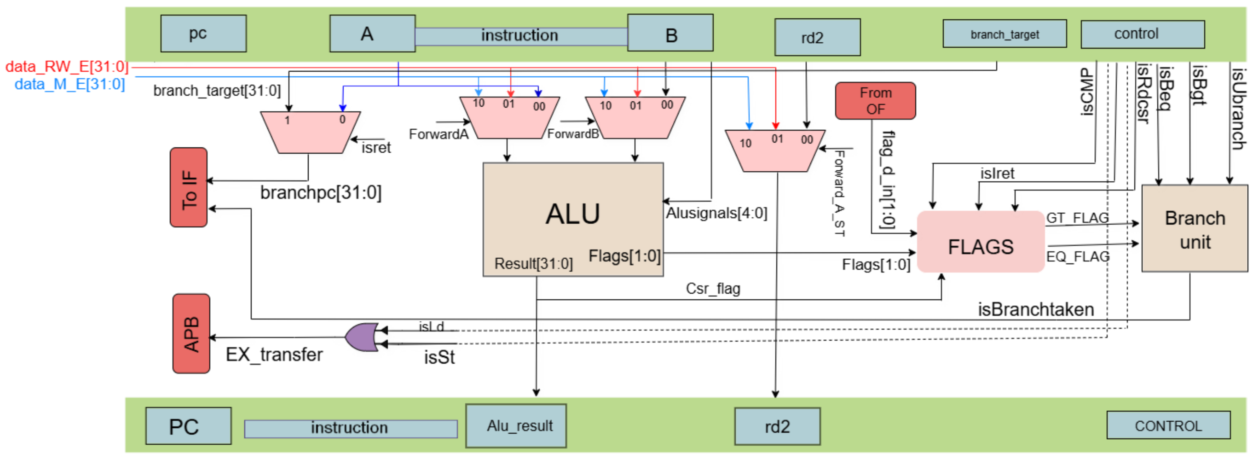


Figure 12: Block Diagram of the Execute Stage

5.1 ALU Module

This Arithmetic Logic Unit (ALU) performs 13 distinct operations on 32-bit inputs `a` and `b`, controlled by a 5-bit `alusignals` input. It produces a 32-bit `result` and 2-bit `flags` output.

Opcode	Operation	Behavior
00000	Add	`result = a + b`
01110	Load	`result = a + b` (effective address calculation)
01111	Store	`result = a + b` (effective address calculation)
00001	Subtract	`result = a - b`
00010	HLT	Halts the processor execution
00011	XOR	`result = a ^ b`
00100	ASL (Arithmetic Shift Left)	`result = a << b` (sign-included)
00101	Compare	Sets flags: 01 if equal, 10 if greater, 00 otherwise
00110	Bitwise AND	`result = a & b`
00111	Bitwise OR	`result = a | b`
01000	Bitwise NOT	`result = ~b` (operates on b only)
01001	Move/CSR	Multi-function: controlled by `alusignals[2:0]`
01010	Logical Shift Left	`result = a << b`
01011	Logical Shift Right	`result = a >> b`
01100	ASR (Arithmetic Shift Right)	`result = a >>> b` (sign-extended)

1. SET CSR Operation

- When the set signal is high for a CSR operation:
  - `a`: Current value from the CSR register
  - `b`: Mask generated by the assembler and computed by the immediate generator
  - `result = a | b`
  - Sets bits in CSR where mask has 1s

  - Preserves original CSR values where mask has 0s

2. CLEAR CSR Operation

- When the clear signal is high for a CSR operation:
  - `a`: Current value from the CSR register
  - `b`: Mask generated by the assembler and computed by the immediate generator
  - `result = a & b`
  - Clears bits in the CSR where mask has 1s
  - Preserves original CSR values where mask has 0s

3. READ CSR Operation

- When the read signal is high for a CSR operation:
  - `a`: Current value from the CSR register
  - `b`: Mask or shift amount generated by the immediate generator
  - `result = a >> b`
  - This shifts the CSR value right by the number of positions specified by `b`.
  - Example: To read the 5th bit, shift by 5. The LSB now contains the required bit and is routed to flags.

4. Flag Handling

- General Case: `flags = 2'b00` for non-compare operations
- For compare instruction only:
  - If `a == b`, `flags = 2'b01`
  - If `a > b`, `flags = 2'b10`
  - Else, `flags = 2'b00`

5.2 Flag Extraction Module

This module processes and routes condition flags for critical processor operations:

- Compare instructions: Extracts ALU-generated flags
- CSR read operations: Maps CSR status to flags
- Interrupt returns: Restores saved flags
- Default behavior: Maintains previous flag state

- **Compare Mode (`alusignal = 00101`):** Directly routes ALU comparison flags
- **CSR Read Mode (`isRdCsr = 1`):**
  - `GT_flag = CSR status bit`
  - `EQ_flag = inverse of CSR status bit`
- **Interrupt Return Mode (`isIret = 1`):** Restores flags saved before the interrupt
- **Default Mode:** Maintains previous flag values

5.3 Branch Unit Module

This combinational logic module determines whether a branch should be taken based on condition flags and branch type signals. It handles three branch types:

- Conditional Equality (`BEQ`)
- Conditional Greater-Than (`BGT`)
- Unconditional Branch (`UBranch`)

The unit outputs a high signal if:

- BEQ instruction and equal flag is set
- BGT instruction and greater-than flag is set
- Unconditional branch (e.g., call, b, ret, iret)

5.4 Multiplexers in the Execute Stage

1. **Branch Target MUX (MUX–M7)**
   - **Inputs:** `branch_target_E`, `a_E`
   - **Control:** `isRet_E`
   - **Output:** `pc_branch_E` — sent to fetch stage for PC redirection
2. **Operand A Forwarding MUX (MUX–M22)**
   - **Inputs:** `a_E`, `data_RW_E`, `data_M_E`
   - **Control:** `forwardA_E` (2-bit selector)
   - **Output:** `a_alu_top`
3. **Operand B Forwarding MUX (MUX–M23)**
   - **Inputs:** `b_E`, `data_RW_E`, `data_M_E`
   - **Control:** `forwardB_E` (2-bit selector)
   - **Output:** `b_alu_top`
4. **Store Data Forwarding MUX (MUX–M57)**
   - **Inputs:** `rd2_E`, `memory_unit_data`, `data_M_E`
   - **Control:** `forwardA` (2-bit selector)
   - **Output:** `rd2_mux` — buffered for memory stage

5.4.1 EX–MA Pipeline Register

The Execute to Memory Access (EX–MA) pipeline register holds results and control signals from the Execute stage, preparing them for use in memory operations or write-back. It transfers the evaluated ALU result, instruction metadata, and control bits into the memory stage.

EX–MA Pipeline Register Fields:

1. **pc [31:0]** – Program counter from the Execute stage, used for tracking control flow and debugging.
2. **alu_result [31:0]** – The result of the ALU operation (e.g., effective address, arithmetic computation).
3. **instruction [31:0]** – The original instruction being processed.

4. **Control Signals:**
   - `isSt` – Store instruction flag; determines if a memory write occurs.
   - `isWb` – Write-back enable flag; passed forward to determine register update.
   - `isLd` – Load instruction flag; indicates that a memory read is needed.
   - `isCall` – Indicates a function call; used for link register updates or trace/debug.
   - `isWbCsr` – CSR write-back enable; asserts when CSR value should be written to.
5. **Register Fields:**
   - `rd` – Destination register index.
   - `ra` – Source register index (used for CSR operations, call/return tracing, etc.).
6. **rst** – Active-high reset; clears the contents of the EX–MA register, effectively flushing the pipeline.

Explanation:
The EX–MA register acts as a staging buffer between the Execute and Memory stages. It ensures consistent control and data propagation across the pipeline. Key responsibilities include:

- Holding the ALU-computed address or result.
- Passing control signals to control memory access and write-back logic.
- Maintaining instruction and PC information for precise exception or debug handling.

On reset, all values in this pipeline stage are cleared to NOP-equivalent to avoid accidental execution or memory writes.

6 Memory Cycle

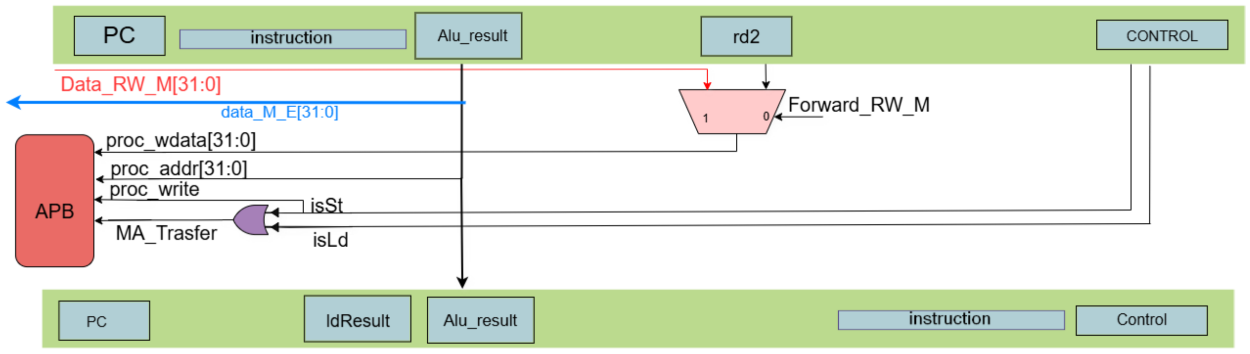


Figure 13: Block Diagram of the Memory Stage

This module implements the memory access stage in a pipelined processor, handling data memory operations, APB3 bus interfacing, and pipeline propagation to the write-back stage.

6.1 MA_TRANSFER Signal

Acts as a memory access request flag to the APB bus.

- Asserted (1) when either a load (`isLd_M`) or store (`isSt_M`) instruction is in the MEM stage.
- Tells the APB bus, "Initiate a memory transfer now."
- Active only when memory interaction is required.

6.2 APB Bus Signals

These signals connect directly to the APB bus:

1. **proc_addr (Output)**
   - Source: `alu_result_M` (from EX stage).
   - Provides the word-aligned memory address for loads/stores.
   - *Note:* Uses the ALU-computed address (e.g., base + offset).
2. **proc_write (Output)**
   - 1 $\Rightarrow$ Store (write to memory)
   - 0 $\Rightarrow$ Load (read from memory)
3. **proc_wdata (Output)**
   - Output of the forwarding MUX (`data_in_top`)
   - Carries data to write during stores
4. **proc_rdata (Input)**
   - Data received from the memory through APB during load operations
   - Forwarded to the write-back stage

6.3 Forwarding MUX (MUX–M60)

- **Inputs:**
  - `rd2_M`: Source register value for store instruction (normal path)
  - `ldresult_RW`: Result from previous load instruction (from WB stage)
- **Control:** `forward_RW_M`
- **Output:** `data_in_top` — value to send to memory for store

6.3.1 MA–RW Pipeline Register

The Memory Access to Register Write-back (MA–RW) pipeline register captures results from the memory stage and prepares them for writing back to the register file or CSR. This is the final pipeline stage in the processor's datapath, responsible for concluding instruction execution.

MA–RW Pipeline Register Fields:

1. **pc [31:0]** – Program counter forwarded from the memory stage, used for debug, exception handling, or return operations.
2. **aluResult [31:0]** – Result computed by the ALU in the Execute stage. Used in arithmetic instructions or as an address for memory access.
3. **ldResult [31:0]** – Data loaded from memory during a load instruction. This value will be written to a general-purpose register if `isLd` is asserted.
4. **instruction [31:0]** – Original instruction propagated for reference, tracing, or CSR operations.
5. **Control Signals:**
   - `isWb` – Write-back enable; indicates that a result should be written to the register file.
   - `isLd` – Load instruction indicator; selects `ldResult` as the source for write-back.
   - `isCall` – Identifies call instructions; may be used for storing return addresses.
   - `isWbCsr` – CSR write-back enable; when high, writes data to a control/status register.
6. **Registers:**
   - `rd` – Destination register index for write-back.
   - `ra` – Source register (used for tracking or special instruction behavior).
7. **rst** – Reset signal; clears the MA–RW pipeline register contents when asserted.

Explanation:
The MA–RW register ensures that correct data is selected and forwarded to the register file. It supports different data sources (ALU vs. memory) and tracks control signals for conditional write-back logic.

- If `isLd` is high, `ldResult` is written to `rd`.
- If `isWb` is high and `isLd` is low, `aluResult` is written to `rd`.
- If `isWbCsr` is set, the corresponding CSR write path is enabled.

This stage finalizes the instruction's effect on the architectural state (register file or CSR) and completes the pipeline's execution cycle.

7 Writeback Cycle

This module handles the final data selection and register file update in a pipelined processor.

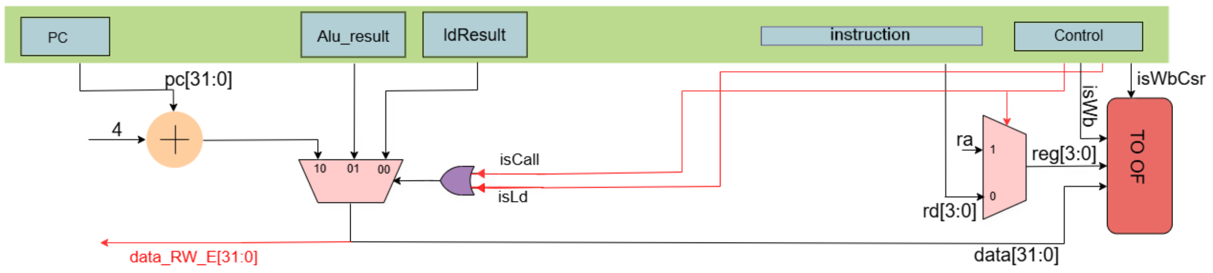


Figure 14: Block Diagram of the Writeback Stage

7.1 Next Address Calculator (pc_plus1)

- Computes PC + 1 (next sequential instruction address)
- Return address for CALL instructions
- Writes the next address to `register[15]` of the register file as the return address when `isCall` signal is high

7.2 Data Source MUX (3:1 MUX) (MUX–M10)

- **Inputs:**
  - `alu_result_RW`: Result from the ALU
  - `ldresult_RW`: Data loaded from memory
  - `pcplus4_top1`: PC + 4 value (used as return address)
- **Select Signal:** {`isCall_RW`, `isLd_RW`} (2 bits)
  - 00: ALU result (normal operations)
  - 01: Load data (memory read)
  - 10: PC+4 (call/return address)
  - 11: Undefined (defaults to ALU)

7.3 Destination Register MUX (2:1 MUX) (MUX–M15)

- **Inputs:**
  - `RD_RW`: Destination register from instruction
  - `ra_RW`: Return address register (e.g., `r15`)
- **Control:** `isCall_RW` — High for call instructions
- **Output:** `reg_RW` — register where data is written

8 Data Hazards and Stalling

8.1 Data Hazards

8.1.1 1. Forwarding from RW to OF

Signal: `forward_RW_OF_rs1`
The forwarding signal is enabled (set to 1) when:

1. The instruction in the Write-Back (WB) stage is a general-purpose register write-back instruction (i.e., not a CSR instruction), and one of the following is true:
    - It writes to the same register as the RS1 source register of the Decode (OF) stage instruction,
    - It writes to the same register as RS1 and the OF-stage instruction is a store (`opcode = 01111`), which uses RS1 as the base address,
    - Neither the WB-stage nor the OF-stage instruction involves CSR registers (to prevent incorrect forwarding between CSR and general-purpose registers),
    - The instruction in the WB stage should not be a call instruction.
2. **OR** both the WB-stage and OF-stage instructions access the same CSR register, and CSR forwarding is needed.

Signal: `forward_RW_OF_rs2`
Enabled (set to 1) when:

1. The WB stage writes to a general-purpose register, and:
    - It matches RS2 in the Decode (OF) stage,
    - Or matches a register used in OF-stage as destination where value is expected from WB,
    - It is not a call instruction,
    - No CSR access by either stage.

8.1.2 2. Forwarding from MEM to EX

Signal: `forwardA_E`
Enabled (set to 10) when:

1. MEM-stage instruction writes to a general-purpose register, and:
    - It matches RS1 in EX-stage,
    - It is not a load (since data won't be available in time — handled via stalling),
    - Neither instruction accesses CSR,
    - Neither instruction is a call.
2. **OR** both instructions access same CSR and CSR forwarding is needed.

Signal: `forwardB_E` — same conditions as `forwardA_E` but for RS2 register.
Signal: `forwardA_M_E`
Enabled (set to 10) when:

- MEM-stage instruction writes to a register,
- EX-stage is a store using the same destination register.

Example:

```
st r3, 0[r2]     <-- EX stage
add r3, r1, r2   <-- MEM stage
```

8.1.3 3. Forwarding from RW to EX

Signal: `forwardA_E`
Enabled (set to 01) when:

- RW-stage writes to the same register as RS1 in EX-stage,
- Neither involves CSR,
- Not a call instruction.

Signal: `forwardB_E` — same logic as `forwardA_E`, for RS2.
Signal: `forwardA_M_E`
Enabled (set to 01) when:

- RW-stage writes to a register,
- EX-stage is a store using the same register.

Example:

```
st r3, 0[r2]     <-- EX stage
nop
add r3, r1, r2   <-- RW stage
```

8.1.4 4. Forwarding from RW to MEM

Signal: `Forward_RW_M`
Enabled (set to 1) when:

- RW-stage instruction is a load,
- MEM-stage is a store using same destination register.

Example:

```
st r3, 0[r2]     <-- MEM stage
ld r3, 0[r4]     <-- RW stage
```

8.2 Data Hazard Stalling

Stalling means inserting a bubble (NOP) into the pipeline to pause the dependent instruction until data becomes available. It occurs when a load instruction is followed immediately by an instruction using the loaded data.
Behavior:

- ALU instruction is stalled in OF for one cycle.
- A NOP is inserted behind the load.

Signal: `add_stall` is asserted (1'b1) under these conditions:

1. **Standard Load-Use Hazard for ALU Operations**
   - EX stage: Load instruction (`opcode = 5'b01110`)
   - Destination register matches RS1 or RS2 in OF stage
   - OF-stage instruction is not a store
2. **Load-Use Hazard for Return Instructions**
   - EX stage loads into `r15` (e.g., `POP r15`)

- OF-stage instruction is `RET` (`opcode = 5'b10100`)

3. **Load-Use Hazard for Interrupt Return (IRET)**
   - EX stage loads into `r12` (e.g., `POP r12`)
   - OF-stage instruction is `IRET` (`opcode = 5'b10101`)
   - `r12` stores the PC during interrupts