

***UNIVERSITY OF***

***ENGINEERING AND TECHNOLOGY, ATD CAMPUS*.**

**Microarchitecture Of Subservient**

**NAME:**

1. **HASHAAM QAZI**
2. **FAHAD HUSSAIN**

**REG NO:**

1. **21ABELT0906**
2. **21ABELT0902**

**SUBMITTED TO: Dr Aneesullah**

**DATE: 6/12/2024**

Contents

[Subservient\_debug\_switch 3](#_Toc184335392)

[inputs and outputs 3](#_Toc184335393)

[serving\_arbiter 6](#_Toc184335394)

[Inputs and Outputs 6](#_Toc184335395)

[Behavior of the Arbiter 7](#_Toc184335396)

[Memory Inputs: 8](#_Toc184335397)

[Subservient\_rf\_ram\_if 9](#_Toc184335398)

[Ports 9](#_Toc184335399)

[Serving\_mux 12](#_Toc184335400)

[Inputs and Outputs: 12](#_Toc184335401)

[Functionality 13](#_Toc184335402)

[Subservient\_ram 15](#_Toc184335403)

[Inputs and Outputs 15](#_Toc184335404)

[Internal Logic 16](#_Toc184335405)

[Key Observations 17](#_Toc184335406)

[SERV MICROARCHITECTURE 18](#_Toc184335407)

[Serv\_decode 18](#_Toc184335408)

[Overview of serv\_decode: 19](#_Toc184335409)

[Key Components: 21](#_Toc184335410)

[Applications: 22](#_Toc184335411)

[serv\_immdec 24](#_Toc184335412)

[Immediate Extraction 25](#_Toc184335413)

[Control Signals 25](#_Toc184335414)

[Workflow 26](#_Toc184335415)

[ALU 27](#_Toc184335416)

[Inputs 28](#_Toc184335417)

[Key Components 29](#_Toc184335418)

[2. Logical Operations 29](#_Toc184335419)

[3. Comparison Result (o\_cmp) 30](#_Toc184335420)

[4. Result Selection 30](#_Toc184335421)

[5. State Registers 30](#_Toc184335422)

[Serv\_bufreg 31](#_Toc184335423)

[Key Functions 32](#_Toc184335424)

[2. Alignment Checking 32](#_Toc184335425)

[3. Support for Shift Operations 32](#_Toc184335426)

[4. Memory Addressing 32](#_Toc184335427)

[5. Efficient Serial Data Processing 33](#_Toc184335428)

[serv\_bufreg2 33](#_Toc184335429)

[Purpose 34](#_Toc184335430)

[Key Components 34](#_Toc184335431)

[Behavior 35](#_Toc184335432)

[Summary of Key Operations 36](#_Toc184335433)

[serv\_state 37](#_Toc184335434)

[Key Concepts and Behaviors 39](#_Toc184335435)

[serv\_csr 43](#_Toc184335436)

[1. Input and Output Signals 44](#_Toc184335437)

[2. Key Components 45](#_Toc184335438)

[3. Detailed Behavioral Breakdown 47](#_Toc184335439)

[Serv\_mem\_if 48](#_Toc184335440)

[Parameters 49](#_Toc184335441)

[Inputs and Outputs 49](#_Toc184335442)

[Functionality 50](#_Toc184335443)

[Key Points 51](#_Toc184335444)

[Serv\_rf\_if 52](#_Toc184335445)

[Inputs and Outputs 53](#_Toc184335446)

[Functionality 54](#_Toc184335447)

MICRO ARCHITECTURE OF SUBSERVIENT

## Subservient\_debug\_switch



### inputs and outputs

**Debug selector**

input wire i\_debug\_mode

**Debug interface**

input wire [31:0] i\_wb\_dbg\_adr,

input wire [31:0] i\_wb\_dbg\_dat,

input wire [3:0] i\_wb\_dbg\_sel,

input wire i\_wb\_dbg\_we,

input wire i\_wb\_dbg\_stb,

output wire [31:0] o\_wb\_dbg\_rdt,

output wire o\_wb\_dbg\_ack,

**Data bus interface towards CPU**

input wire [31:0] i\_wb\_dbus\_adr,

input wire [31:0] i\_wb\_dbus\_dat,

input wire [3:0] i\_wb\_dbus\_sel,

input wire i\_wb\_dbus\_we,

input wire i\_wb\_dbus\_stb,

output wire [31:0] o\_wb\_dbus\_rdt,

output wire o\_wb\_dbus\_ack,

**Data bus interface towards memory/peripherals**

output wire [31:0] o\_wb\_mux\_adr,

output wire [31:0] o\_wb\_mux\_dat,

output wire [3:0] o\_wb\_mux\_sel,

output wire o\_wb\_mux\_we,

output wire o\_wb\_mux\_stb,

input wire [31:0] i\_wb\_mux\_rdt,

input wire i\_wb\_mux\_ack);

**Purpose of the Module**

This module routes signals between:

1. **Debug interface** (wb\_dbg): For debugging the system.
2. **CPU data bus interface** (wb\_dbus): Regular communication between the CPU and peripherals/memory.
3. **Shared bus interface** (wb\_mux): The shared interface to memory or peripherals.

It decides whether the debug interface (i\_wb\_dbg) or CPU data bus interface (i\_wb\_dbus) has control of the shared bus (o\_wb\_mux), based on the i\_debug\_mode signal.

**Signal Routing Logic**

1. **Debug Responses**:
   * **Data (o\_wb\_dbg\_rdt)**: Directly forwarded from the shared bus data response (i\_wb\_mux\_rdt).
   * **Acknowledge (o\_wb\_dbg\_ack)**: Enabled only when the system is in debug mode (i\_debug\_mode == 1).
2. **CPU Responses**:
   * **Data (o\_wb\_dbus\_rdt)**: Also forwarded from the shared bus response (i\_wb\_mux\_rdt).
   * **Acknowledge (o\_wb\_dbus\_ack)**: Enabled only when the system is NOT in debug mode (i\_debug\_mode == 0).
3. **Shared Bus Requests**:
   * **Address (o\_wb\_mux\_adr)**:
     + Routed from debug (i\_wb\_dbg\_adr) if in debug mode.
     + Routed from CPU (i\_wb\_dbus\_adr) otherwise.
   * **Data (o\_wb\_mux\_dat)**:
     + Routed from debug (i\_wb\_dbg\_dat) if in debug mode.
     + Routed from CPU (i\_wb\_dbus\_dat) otherwise.
   * **Select (o\_wb\_mux\_sel)**, **Write Enable (o\_wb\_mux\_we)**, and **Strobe (o\_wb\_mux\_stb)**:
     + Follow the same logic as the address and data signals.

## serving\_arbiter



### Inputs and Outputs

**Inputs:**

* **Instruction Bus (I-Bus) Signals:**
  + **i\_wb\_cpu\_ibus\_adr:** Address from the CPU's instruction bus.
  + **i\_wb\_cpu\_ibus\_stb:** Strobe signal indicating a valid instruction request.
* **Data Bus (D-Bus) Signals:**
  + **i\_wb\_cpu\_dbus\_adr:** Address from the CPU's data bus.
  + **i\_wb\_cpu\_dbus\_dat:** Data from the CPU to write.
  + **i\_wb\_cpu\_dbus\_sel:** Byte select signals for finer control over data writes.
  + **i\_wb\_cpu\_dbus\_we:** Write enable signal for the data bus.
  + **i\_wb\_cpu\_dbus\_stb:** Strobe signal indicating a valid data request.
* **Memory Bus Signals (from memory to CPU):**
  + **i\_wb\_mem\_rdt:** Data read from memory.
  + **i\_wb\_mem\_ack:** Acknowledgment signal from memory indicating a completed operation.

**Outputs:**

* **To CPU:**
  + **D-Bus:**
    - **o\_wb\_cpu\_dbus\_rdt:** Data read from memory for the data bus.
    - **o\_wb\_cpu\_dbus\_ack:** Acknowledgment signal for the data bus.
  + **I-Bus:**
    - **o\_wb\_cpu\_ibus\_rdt:** Data read from memory for the instruction bus.
    - **o\_wb\_cpu\_ibus\_ack:** Acknowledgment signal for the instruction bus.
* **To Memory:**
  + **o\_wb\_mem\_adr:** Address to the memory.
  + **o\_wb\_mem\_dat:** Data to write to the memory.
  + **o\_wb\_mem\_sel:** Byte select signals for the memory.
  + **o\_wb\_mem\_we:** Write enable signal to the memory.
  + **o\_wb\_mem\_stb:** Strobe signal indicating a valid memory request.

### Behavior of the Arbiter

The arbiter decides which bus (I-Bus or D-Bus) gets access to the shared memory bus based on the input strobe signals (i\_wb\_cpu\_ibus\_stb and i\_wb\_cpu\_dbus\_stb).

**Signal Assignments:**

1. **CPU D-Bus Outputs:**
   * **o\_wb\_cpu\_dbus\_rdt:** Memory data (i\_wb\_mem\_rdt) is directly passed to the CPU's data bus.
   * **o\_wb\_cpu\_dbus\_ack:** Acknowledgment is valid when:
     + Memory acknowledges (i\_wb\_mem\_ack).
     + The I-Bus is *not* requesting (!i\_wb\_cpu\_ibus\_stb).
2. **CPU I-Bus Outputs:**
   * **o\_wb\_cpu\_ibus\_rdt:** Memory data (i\_wb\_mem\_rdt) is passed to the instruction bus.
   * **o\_wb\_cpu\_ibus\_ack:** Acknowledgment is valid when:
     + Memory acknowledges (i\_wb\_mem\_ack).
     + The I-Bus is requesting (i\_wb\_cpu\_ibus\_stb).

### Memory Inputs:

* + **o\_wb\_mem\_adr:** Chooses the address based on the I-Bus strobe:
    - If the I-Bus is active (i\_wb\_cpu\_ibus\_stb), use the I-Bus address.
    - Otherwise, use the D-Bus address.
  + **o\_wb\_mem\_dat:** Forward data from the D-Bus.
  + **o\_wb\_mem\_sel:** Forward byte select signals from the D-Bus.
  + **o\_wb\_mem\_we:** Enable memory writes only if:
    - The D-Bus is requesting (**i\_wb\_cpu\_dbus\_we**).
    - The I-Bus is *not* active (**!i\_wb\_cpu\_ibus\_stb**).
  + **o\_wb\_mem\_stb:** Set when either the I-Bus or the D-Bus is requesting.

**Functionality Summary**

* **Read Operations:**
  + Both I-Bus and D-Bus can read from memory. The arbiter ensures that the correct acknowledgment and data are forwarded back to the CPU.
* **Write Operations:**
  + Only the D-Bus can write to memory, and only when the I-Bus is not active.
* **Arbitration Logic:**
  + Priority is given to the I-Bus (**i\_wb\_cpu\_ibus\_stb**). If it is active, its address and strobe take precedence.

## Subservient\_rf\_ram\_if



**Parameters**

* **width**: The data width of the memory interface (default is 8 bits).
* **reset\_strategy**: Defines the reset behavior, e.g., "MINI" or "NONE".
* **csr\_regs**: Number of Control and Status Registers (CSRs) used.
* **depth**: The SRAM depth, calculated based on the number of registers and CSRs.
* **l2w**: Log base 2 of the width, used for addressing alignment.

### Ports

**SERV Side**

* **Inputs**:
  + i\_clk, i\_rst: Clock and reset signals.
  + i\_wreq, i\_rreq: Write and read request signals.
  + i\_wreg0, i\_wreg1: Write register addresses (low and high).
  + i\_rreg0, i\_rreg1: Read register addresses (low and high).
  + i\_wen0, i\_wen1: Write enable signals for low and high writes.
  + i\_wdata0, i\_wdata1: Write data for low and high writes.
* **Outputs**:
  + o\_ready: Indicates when the adapter is ready for a new request.
  + o\_rdata0, o\_rdata1: Read data outputs.

**RAM Side**

* **Outputs**:
  + o\_waddr: Write address to SRAM.
  + o\_wdata: Data to be written to SRAM.
  + o\_wen: Write enable for SRAM.
  + o\_raddr: Read address from SRAM.
  + o\_ren: Read enable for SRAM.
* **Inputs**:
  + i\_rdata: Read data from SRAM.

**Core Features**

1. **Ready Signal (o\_ready)**:
   * The adapter is ready if either a request is pending (rgnt) or a write request (i\_wreq) is active.
2. **Write Logic**:
   * **Data Alignment**:
     + Manages partial word writes by using wtrig0 and wtrig1 signals to determine when to trigger low (i\_wdata0) or high (i\_wdata1) writes.
   * **Addressing**:
     + Combines the register index (i\_wreg0 or i\_wreg1) and an internal write counter (wcnt) to form the write address.
   * **Write Enable (o\_wen)**:
     + Ensures writes are only performed when data is valid and conditions are met.
3. **Read Logic**:
   * **Data Alignment**:
     + Handles partial word reads using triggers (rtrig0, rtrig1) and the read counter (rcnt).
   * **Addressing**:
     + Combines the register index (i\_rreg0 or i\_rreg1) with the read counter to form the read address.
   * **Read Enable (o\_ren)**:
     + Ensures SRAM is read when valid read conditions are met.
   * **Output Data (o\_rdata0, o\_rdata1)**:
     + Captures the read data in rdata0 and rdata1, aligning it as needed for the register file interface.
4. **Reset Logic**:
   * Controlled by the reset\_strategy parameter.
   * Initializes control signals like rgate, rcnt, and rgnt during reset.
5. **Counters**:
   * **Write Counter (wcnt)**:
     + Tracks write alignment within a word.
   * **Read Counter (rcnt)**:
     + Tracks read alignment and manages read triggers.

**Operation**

**Write Process:**

1. Write requests (i\_wreq) initiate the write cycle.
2. The write address and data are determined based on triggers (wtrig0, wtrig1) and counters (wcnt).
3. Data is written to SRAM when the write enable signal (o\_wen) is active.

**Read Process:**

1. Read requests (i\_rreq) initiate the read cycle.
2. The read address is formed from the register index and the read counter.
3. Data is captured from SRAM and aligned for output to the register file interface.

## Serving\_mux



**Key Components**

### Inputs and Outputs:

* + **Inputs**:
    - **From the CPU**:
      * i\_wb\_cpu\_adr: 32-bit address for the transaction.
      * i\_wb\_cpu\_dat: 32-bit data to be written to the bus.
      * i\_wb\_cpu\_sel: 4-bit select signals for byte-level control.
      * i\_wb\_cpu\_we: Write enable signal (1 for write, 0 for read).
      * i\_wb\_cpu\_stb: Strobe signal indicating a valid transaction request.
    - **From Memory (mem) and External (ext) Devices**:
      * i\_wb\_mem\_rdt, i\_wb\_ext\_rdt: Read data from memory or external devices.
      * i\_wb\_mem\_ack, i\_wb\_ext\_ack: Acknowledge signals for transaction completion.
  + **Outputs**:
    - **To the CPU**:
      * o\_wb\_cpu\_rdt: 32-bit read data from memory or external devices.
      * o\_wb\_cpu\_ack: Acknowledge signal routed back to the CPU.
    - **To Memory**:
      * o\_wb\_mem\_\*: Address, data, select, write-enable, and strobe signals for memory transactions.
    - **To External Device**:
      * o\_wb\_ext\_\*: Address, data, select, write-enable, and strobe signals for external device transactions.

1. **Address Decoding**:
   * ext = (i\_wb\_cpu\_adr[31:30] != 2'b00);
     + Determines whether the transaction is targeted at an **external device** (ext = 1) or **internal memory** (ext = 0).
     + Addresses starting with 00 (in the most significant 2 bits) are routed to **memory**; other addresses are routed to **external devices**.

### Functionality

1. **Routing Read Data (o\_wb\_cpu\_rdt)**:
   * If ext = 1: The read data is routed from the external device (i\_wb\_ext\_rdt).
   * If ext = 0: The read data is routed from memory (i\_wb\_mem\_rdt).
2. **Routing Acknowledge Signal (o\_wb\_cpu\_ack)**:
   * If ext = 1: The acknowledge signal comes from the external device (i\_wb\_ext\_ack).
   * If ext = 0: The acknowledge signal comes from memory (i\_wb\_mem\_ack).
3. **Memory Signals (o\_wb\_mem\_\*)**:
   * Address, data, select, and write-enable signals are directly passed from the CPU.
   * The strobe signal (o\_wb\_mem\_stb) is active only when ext = 0.
4. **External Device Signals (o\_wb\_ext\_\*)**:
   * Address, data, select, and write-enable signals are also passed directly from the CPU.
   * The strobe signal (o\_wb\_ext\_stb) is active only when ext = 1.

**How It Works**

* The multiplexer acts as a **single point of access** for the CPU to communicate with both memory and external devices.
* Based on the address, it determines whether the transaction is directed to:
  1. **Internal Memory** (addresses starting with 00).
  2. **External Device** (addresses starting with 01, 10, or 11).
* Only the appropriate set of signals (mem or ext) is activated at a time, ensuring that the CPU communicates with the correct device.

**Use Case**

This module is ideal for systems using the **Wishbone interconnect**, where a CPU needs to interact with multiple peripherals or memory regions. It simplifies the design by abstracting the routing logic for memory and peripheral accesses into a single module.

## Subservient\_ram



**Key Features**

1. **Wishbone Bus Interface**:
   * Handles interactions with a CPU or master device via a Wishbone protocol.
   * Converts 32-bit Wishbone transactions into byte-level memory operations.
2. **Register File (RF) Operations**:
   * Provides dedicated memory access for non-Wishbone transactions.
   * Supports independent read (i\_raddr, o\_rdata) and write (i\_waddr, i\_wdata, i\_wen) operations.
3. **Shared SRAM Access**:
   * Manages memory read and write requests from both the Wishbone interface and the RF, ensuring correct arbitration.

**Parameters**

* **depth**: Total memory depth in bytes.
* **aw**: Address width, calculated using $clog2(depth).

### Inputs and Outputs

1. **Inputs**:
   * **Register File (RF) Inputs**:
     + i\_clk, i\_rst: Clock and reset signals.
     + i\_waddr, i\_wdata, i\_wen: Write address, data, and enable signals.
     + i\_raddr, i\_ren: Read address and enable signals.
   * **Wishbone Interface Inputs**:
     + i\_wb\_adr, i\_wb\_dat, i\_wb\_sel, i\_wb\_we, i\_wb\_stb: Wishbone address, data, byte select, write enable, and strobe signals.
   * **SRAM Inputs**:
     + i\_sram\_rdata: Data read from SRAM.
2. **Outputs**:
   * **Register File Output**:
     + o\_rdata: Data read from memory or default value.
   * **SRAM Outputs**:
     + o\_sram\_\*: Signals to control SRAM (write/read address, data, enable, etc.).
   * **Wishbone Interface Outputs**:
     + o\_wb\_rdt: 32-bit read data.
     + o\_wb\_ack: Acknowledge signal for transaction completion.

### Internal Logic

1. **Wishbone Enable Signal (wb\_en)**:
   * Indicates an active Wishbone transaction when:
     + Strobe (i\_wb\_stb) is active.
     + No RF write (rf\_wen\_r is low).
     + Wishbone acknowledge signal (o\_wb\_ack) is low.
2. **Wishbone Write Enable (wb\_we)**:
   * Active if Wishbone write enable (i\_wb\_we) and byte select (i\_wb\_sel[bsel]) are high.
3. **Write and Read Address/Data Multiplexing**:
   * Writes:
     + If Wishbone transaction is active (wb\_en), address/data comes from the Wishbone interface.
     + Otherwise, address/data comes from the RF interface.
   * Reads:
     + Wishbone transactions use the Wishbone address.
     + Non-Wishbone reads use the RF address.
4. **Byte Selection (bsel)**:
   * Selects which byte (0–3) of a 32-bit Wishbone word is currently being accessed.
   * Incremented on each clock cycle during a Wishbone transaction.
5. **Read Data (o\_wb\_rdt)**:
   * Concatenates bytes read from SRAM (i\_sram\_rdata) into the full 32-bit Wishbone word.
6. **Acknowledge Signal (o\_wb\_ack)**:
   * Asserted when all bytes of a Wishbone word have been accessed (bsel = 2'b11).
7. **Reset Behavior**:
   * Clears bsel and o\_wb\_ack.
   * Resets regzero, which controls whether reads to the last address return zero.

### Key Observations

1. **Shared Access Arbitration**:
   * Both RF and Wishbone interfaces can write to the SRAM, but only one can be active at a time.
   * Priority is implicitly given to Wishbone transactions when wb\_en is high.
2. **Zeroing Mechanism**:
   * If the RF read address (i\_raddr) is the last possible value (regzero is true), the read data (o\_rdata) is set to zero.
3. **Byte-Level Transactions**:
   * The module operates at an 8-bit level, enabling efficient memory access for systems requiring byte granularity.

**Functionality Overview**

1. **Wishbone Read Transaction**:
   * wb\_en is asserted.
   * SRAM read address is set from the Wishbone interface.
   * Data bytes are accumulated into o\_wb\_rdt as bsel increments.
   * o\_wb\_ack signals transaction completion after all bytes are read.
2. **Wishbone Write Transaction**:
   * wb\_en and wb\_we are asserted.
   * SRAM write address and data are set from the Wishbone interface.
3. **Register File Access**:
   * Independent of the Wishbone interface.
   * Uses separate RF-specific signals for read/write operations.

# SERV MICROARCHITECTURE

## Serv\_decode



### Overview of serv\_decode:

1. **Purpose**:
   * Decode RISC-V instructions fetched from memory (i\_wb\_rdt) into control signals for other components of the processor.
   * It identifies instruction types (e.g., arithmetic, branch, memory access) and determines the corresponding control outputs.
2. **Parameters**:
   * PRE\_REGISTER: Controls whether pre-decoded signals are registered (pipelined).
   * MDU: Indicates if the Multiply-Divide Unit (MDU) is enabled.
3. **Inputs**:
   * **clk:** Clock signal for synchronous updates.
   * **i\_wb\_rdt[31:2]**: Instruction read from the Wishbone bus.
   * **i\_wb\_en:** Enable signal to latch new instruction data.
4. **Outputs**:

**To state**

* o\_sh\_right,
* o\_bne\_or\_bge,
* o\_cond\_branch,
* o\_e\_op,
* o\_ebreak,
* o\_branch\_op,
* o\_shift\_op,
* o\_slt\_or\_branch,
* o\_rd\_op,
* o\_two\_stage\_op,
* o\_dbus\_en,

**MDU**

* o\_mdu\_op,

**Extension**

* output reg [2:0] o\_ext\_funct3,

**To bufreg**

* o\_bufreg\_rs1\_en,
* o\_bufreg\_imm\_en,
* o\_bufreg\_clr\_lsb,
* o\_bufreg\_sh\_signed,

**To ctrl**

* o\_ctrl\_jal\_or\_jalr,
* o\_ctrl\_utype,
* o\_ctrl\_pc\_rel,
* o\_ctrl\_mret,

**To alu**

* o\_alu\_sub,
* output reg [1:0] o\_alu\_bool\_op,
* o\_alu\_cmp\_eq,
* o\_alu\_cmp\_sig,
* output reg [2:0] o\_alu\_rd\_sel,

**To mem IF**

* o\_mem\_signed,
* o\_mem\_word,
* o\_mem\_half,
* o\_mem\_cmd,

**To CSR**

* o\_csr\_en,
* output reg [1:0] o\_csr\_addr,
* o\_csr\_mstatus\_en,
* o\_csr\_mie\_en,
* o\_csr\_mcause\_en,
* output reg [1:0] o\_csr\_source,
* o\_csr\_d\_sel,
* o\_csr\_imm\_en,
* o\_mtval\_pc,

**To top**

* output reg [3:0] o\_immdec\_ctrl,
* output reg [3:0] o\_immdec\_en,
* o\_op\_b\_source,

**To RF IF**

* o\_rd\_mem\_en,
* o\_rd\_csr\_en,
* o\_rd\_alu\_en);

1. **Pre-register Option**:
   * When PRE\_REGISTER is enabled, instruction fields (opcode, funct3, etc.) are latched on the clock's rising edge, reducing combinational delay.

### Key Components:

**1. Opcode and Function Decoding:**

* Extracts and interprets fields (opcode, funct3, etc.) from the instruction word (i\_wb\_rdt).
* Generates control signals based on these fields.

**2. Control Signals:**

* **Branch and Jump**:
  + **o\_cond\_branch**: Identifies conditional branches (e.g., BEQ, BNE).
  + **o\_ctrl\_jal\_or\_jalr:** Differentiates between JAL and JALR.
* **ALU Operations**:
  + **o\_alu\_sub:** Indicates subtraction operations or comparisons.
  + **o\_alu\_bool\_op:** Specifies boolean operations for the ALU.
* **Memory Access**:
  + **o\_mem\_cmd:** Differentiates between load and store instructions.
  + **o\_mem\_signed:** Indicates whether a memory access is signed or unsigned.
* **CSR Handling**:
  + **o\_csr\_en:** Enables access to Control and Status Registers.
  + **o\_csr\_mstatus\_en:** Enables updates to the mstatus CSR.

**3. Immediate Decoding:**

* **co\_immdec\_ctrl** and **co\_immdec\_en:** Generate control signals for immediate handling based on instruction type.

**4. Pipeline Registering:**

* If PRE\_REGISTER is enabled, instruction fields are latched into registers to improve timing

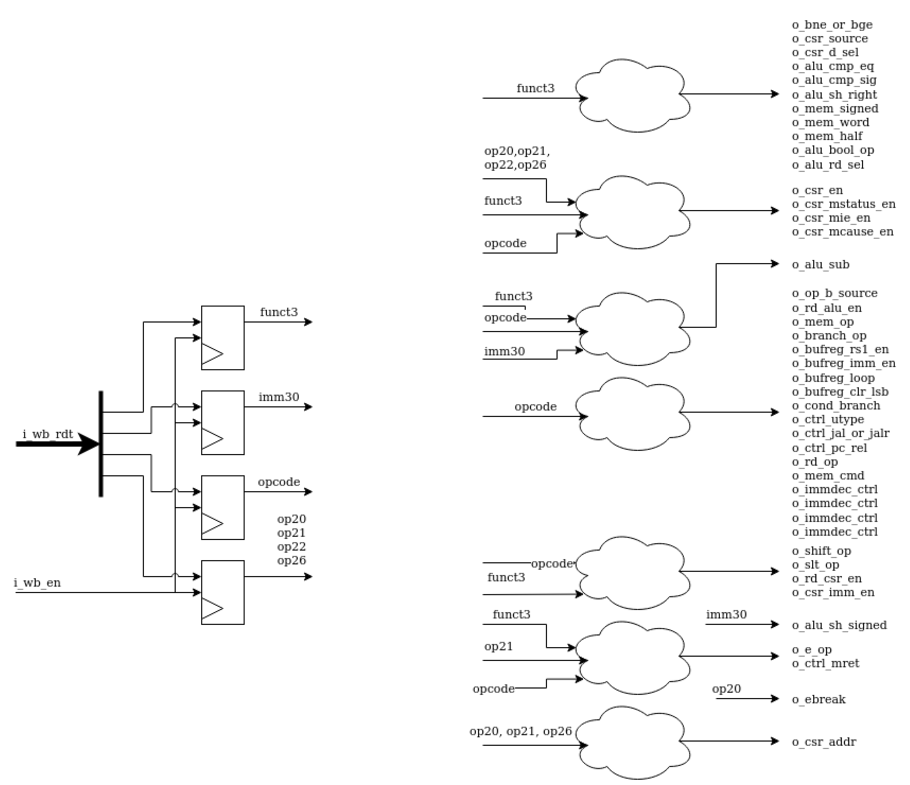
**Pipeline Optimization (Timing vs. Throughput)**

* **When PRE\_REGISTER is enabled:**
  + Decoded signals like funct3, imm30, and opcode are captured into **pipeline registers** (flip-flops) at the rising edge of the clock before further processing.
  + This helps manage timing issues in long combinational paths by "registering" intermediate values earlier in the pipeline.
  + Improves maximum clock frequency (fmax) by breaking up the critical path into smaller chunks.
* **When PRE\_REGISTER is disabled:**
  + Signals are decoded directly from i\_wb\_rdt without intermediate pipeline registers.
  + This saves hardware (flip-flops), but may result in a longer critical path and reduced fmax.

**Design Parameterization**

* The PRE\_REGISTER parameter provides flexibility to the design, allowing the designer to:
  + Choose between higher clock speed (PRE\_REGISTER = 1) or fewer hardware resources (PRE\_REGISTER = 0).
  + Reuse the same codebase for different configurations or design requirements (e.g., different target FPGA/ASICs or use cases).

**Separate Responsibilities**

* **Pre-registering (gen\_pre\_register)** separates the responsibility of instruction decoding and signal generation across multiple clock cycles.
* **Post-registering (gen\_post\_register)** does everything in a single cycle, which is simpler but potentially slower for critical paths. 

The two missing bits in the input wire i\_wb\_rdt are intentional and related to the structure of the RISC-V instruction format and memory alignment.

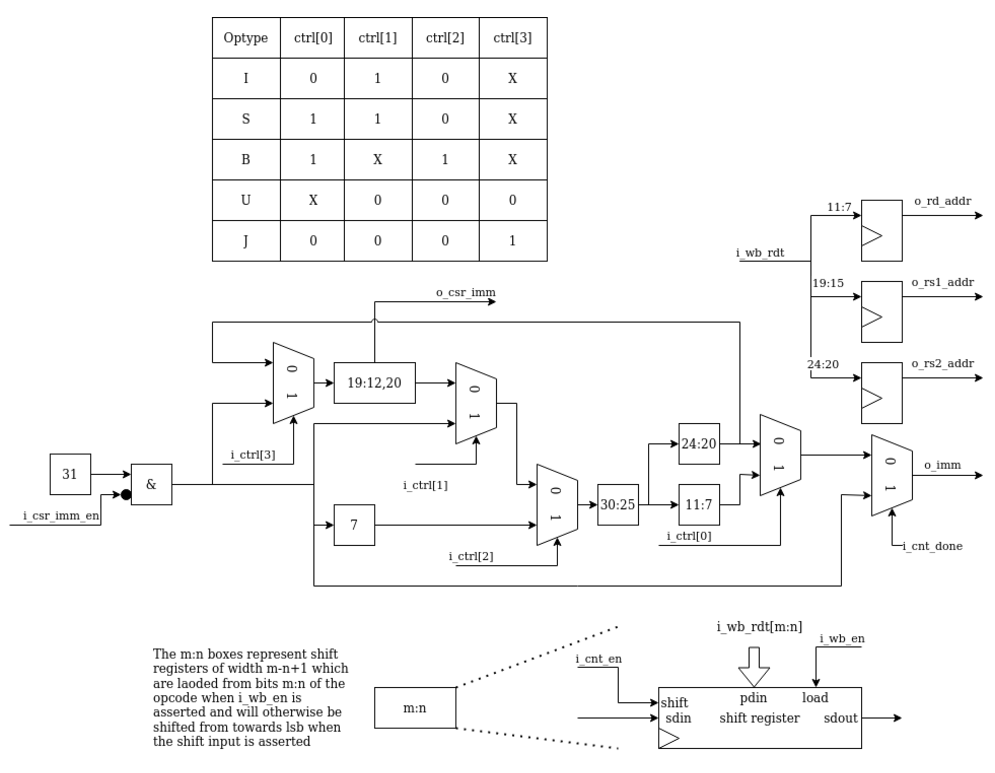
**Explanation:**

1. **RISC-V Instruction Alignment**:
   * RISC-V instructions are 32 bits long and must be aligned on a 4-byte boundary in memory.
   * This means the lowest two bits of the program counter (PC) are always 00. These bits are redundant because they are constant for valid instructions and do not carry meaningful data.
2. **Optimization**:
   * To save hardware resources and simplify design, these two constant bits ([1:0]) are omitted in the instruction data bus (**i\_wb\_rdt).**
   * The bus **i\_wb\_rdt[31:2]** effectively contains the instruction data without these unused lower bits.
3. **Reconstruction**:
   * If needed, the lower two bits can always be assumed to be 00 for instruction fetching and processing, ensuring alignment.

This is a common optimization in designs where memory bandwidth or hardware resources are critical. By eliminating the redundant bits, the design becomes more efficient without losing functionality.

## serv\_immdec





### Inputs

* i\_clk,

**State**

* + i\_cnt\_en,
  + i\_cnt\_done,

**Control**

* + [3:0] i\_immdec\_en,
  + i\_csr\_imm\_en,
  + [3:0] i\_ctrl,
  + i\_wb\_en,
  + [31:7] i\_wb\_rdt

### outputs

* [4:0] o\_rd\_addr,
* [4:0] o\_rs1\_addr,
* [4:0] o\_rs2\_addr,

### Data

* o\_csr\_imm,
* o\_imm

### Immediate Extraction

* Immediate values in RISC-V instructions are spread across specific bitfields of the 32-bit instruction (i\_wb\_rdt). For example:
  + **I-type**: imm[11:0] = bits [31:20].
  + **S-type**: imm[11:5] = bits [31:25], imm[4:0] = bits [11:7].
  + **B-type**: imm[12] = bit 31, imm[10:5] = bits [30:25], imm[4:1] = bits [11:8], imm[11] = bit 7.
* The registers imm31, imm19\_12\_20, imm7, imm30\_25, imm24\_20, and imm11\_7 store these parts temporarily. The diagram shows how each part is connected to a shift register.

### Control Signals

* **i\_ctrl[3:0]**:
  + These signals dictate how parts of the immediate are shifted or manipulated.
  + The truth table in the diagram specifies which control bit is active for each type of immediate (I, S, B).
* **i\_immdec\_en[3:0]**:
  + Enables specific immediate decoding stages.
  + Example: i\_immdec\_en[0] controls the update of imm11\_7.

**3. Shift Registers**

* Instead of directly storing all immediate bits in one go, the module shifts partial results into registers over clock cycles.
* The shift operation is controlled by i\_cnt\_en, i\_ctrl, and i\_immdec\_en.

**4. Sign Bit (signbit)**

* **Purpose**: Extends the sign bit (imm31) for signed instructions.
* The sign bit is cleared for CSR instructions (i\_csr\_imm\_en), as these use zero-extended immediates.

**5. Register Address Extraction**

* o\_rd\_addr: Extracted from bits [11:7] (imm11\_7).
* o\_rs1\_addr: Extracted from bits [19:15] (imm19\_12\_20).
* o\_rs2\_addr: Extracted from bits [24:20] (imm24\_20).

**6. Immediate Output (o\_imm)**

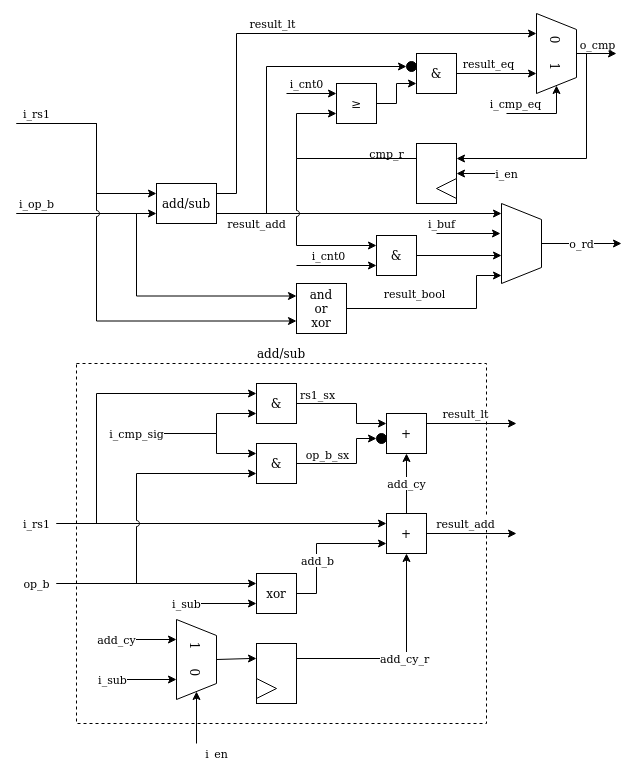
* Combines the intermediate registers to output the decoded immediate.
* Depends on whether the count is done (i\_cnt\_done) or the least significant bit of imm11\_7 or imm24\_20.

### Workflow

1. **Instruction Load (i\_wb\_en)**:
   * When a new instruction is loaded, the module extracts all immediate fields directly from i\_wb\_rdt.
   * This happens in parallel for all fields.
2. **Shifting and Decoding (i\_cnt\_en)**:
   * For complex instructions, the relevant immediate parts are shifted into place over several clock cycles.
   * Control signals (i\_ctrl, i\_immdec\_en) determine how the bits are shifted and combined.
3. **Immediate Generation (o\_imm)**:
   * At the end of the decoding process (i\_cnt\_done), the full immediate value is formed.
   * It is either the sign-extended form (signbit) or the least significant bit from the decoded fields, based on the type of operation.

## ALU





### Inputs

clk,

**State**

* i\_en,
* i\_cnt0,

**Control**

* i\_sub,
* [1:0] i\_bool\_op,
* i\_cmp\_eq,
* i\_cmp\_sig,
* [2:0] i\_rd\_sel,

**Data**

* [B:0] i\_rs1,
* [B:0] i\_op\_b,
* B:0] i\_buf,

**Output**

* [B:0] o\_rd
* o\_cmp,

### Key Components

**1. Arithmetic Operations**

* **Addition/Subtraction:**
  + result\_add computes the sum or difference of i\_rs1 and i\_op\_b.
  + The subtraction is implemented by inverting i\_op\_b (using i\_op\_b ^ {W{i\_sub}}) and adding 1 (if i\_sub is set).
  + The carry-out (add\_cy) is tracked using a flip-flop (add\_cy\_r) to support multi-bit operations over clock cycles.
* **Sign-Extension for Comparisons:**
  + For signed comparisons (i\_cmp\_sig), the most significant bits (rs1\_sx, op\_b\_sx) of i\_rs1 and i\_op\_b are sign-extended.
* **Comparison Operations:**
  + result\_lt: Computes "less than" by comparing the signed operands.
  + result\_eq: Checks equality by ensuring all bits in result\_add are zero.

### 2. Logical Operations

* Logical results (result\_bool) depend on the i\_bool\_op signal:
  + 00: XOR.
  + 01: Zero (used during shift operations).
  + 10: OR.
  + 11: AND.
* The operation is implemented efficiently using a combination of masking and bitwise logic.

### 3. Comparison Result (o\_cmp)

* The o\_cmp output is determined based on the type of comparison:
  + If i\_cmp\_eq is set, o\_cmp is the result of equality check (result\_eq).
  + Otherwise, it is the result of the "less than" comparison (result\_lt).

### 4. Result Selection

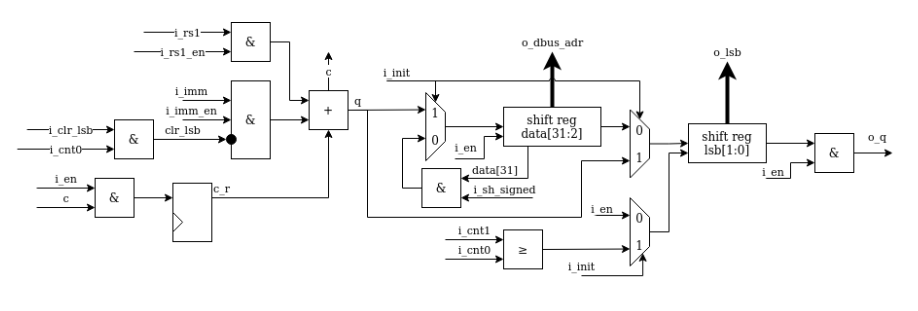
* The final ALU result (o\_rd) is selected based on i\_rd\_sel:
  + i\_rd\_sel[0]: Selects the addition result (result\_add).
  + i\_rd\_sel[1]: Selects the "less than" result (result\_slt).
  + i\_rd\_sel[2]: Selects the logical operation result (result\_bool).
* The result is ORed with the buffer (i\_buf), which is used to preserve previous states.

### 5. State Registers

* **Carry Register (add\_cy\_r):**
  + Tracks the carry-out of addition for multi-cycle operations.
* **Comparison Register (cmp\_r):**
  + Stores the comparison result across clock cycles.

## Serv\_bufreg





**Inputs:**

* rs1, imm, and their sum (rs1 + imm) are potential inputs to the shift register.
* Control signals decide which source is selected.

**Shift Register:**

* A 32-bit shift register holds data and streams it out serially.
* The two LSBs (shift reg lsb[1:0]) are given special treatment to ensure alignment checks and proper serial loading.

**Outputs:**

* o\_dbus\_adr: Parallel output used for memory addressing.
* o\_lsb: Tracks the two LSBs for alignment checking.
* o\_q: Serial output for streaming results during the second stage.

**Control Signals:**

* i\_en, i\_init, i\_cnt0, etc., manage the behavior of the buffer, including loading data, shifting, and controlling the serial output.

### Key Functions

**1. Data Buffering for Multi-Stage Operations**

* The buffer can store:
  + rs1 (register source 1).
  + An immediate value (imm).
  + The sum of rs1 + imm (e.g., effective address for branches or memory access).
  + Data looped back from the shift register output for shift instructions.
* These modes are selected based on control signals, ensuring that appropriate data is stored and processed during the operation.

### 2. Alignment Checking

* The two least significant bits (lsb) of the shift register are monitored to check for alignment errors.
* For certain instructions (e.g., load/store), misaligned addresses can cause errors. The lsb output ensures that the alignment of addresses is verified early in the pipeline.

### 3. Support for Shift Operations

* For shift instructions:
  + Data is loaded into the shift register.
  + The result is streamed out serially during the operation (bit by bit).
* Special handling is done for the two LSBs of the immediate value:
  + These bits are cleared for shift operations to maintain proper alignment.

### 4. Memory Addressing

* The buffer outputs the dbus\_adr signal, which represents the data bus address.
* This parallel output forms the address used in load/store instructions.

### 5. Efficient Serial Data Processing

* The shift register's design allows for efficient serial streaming of data, which is a hallmark of the SERV core's minimal resource usage.
* Data is shifted into and out of the register serially, with the two LSBs being written first before the rest of the register is filled.

## serv\_bufreg2



### Purpose

The module acts as a multi-purpose register (dat) that supports:

1. **Store Operations:** Shifts data to the correct position and presents it on the data bus.
2. **Load Operations:** Latches data from the bus and shifts it out at the appropriate time.
3. **Shift Operations:** Shifts data serially while counting down to determine when the operation is complete.

### Key Components

**Inputs:**

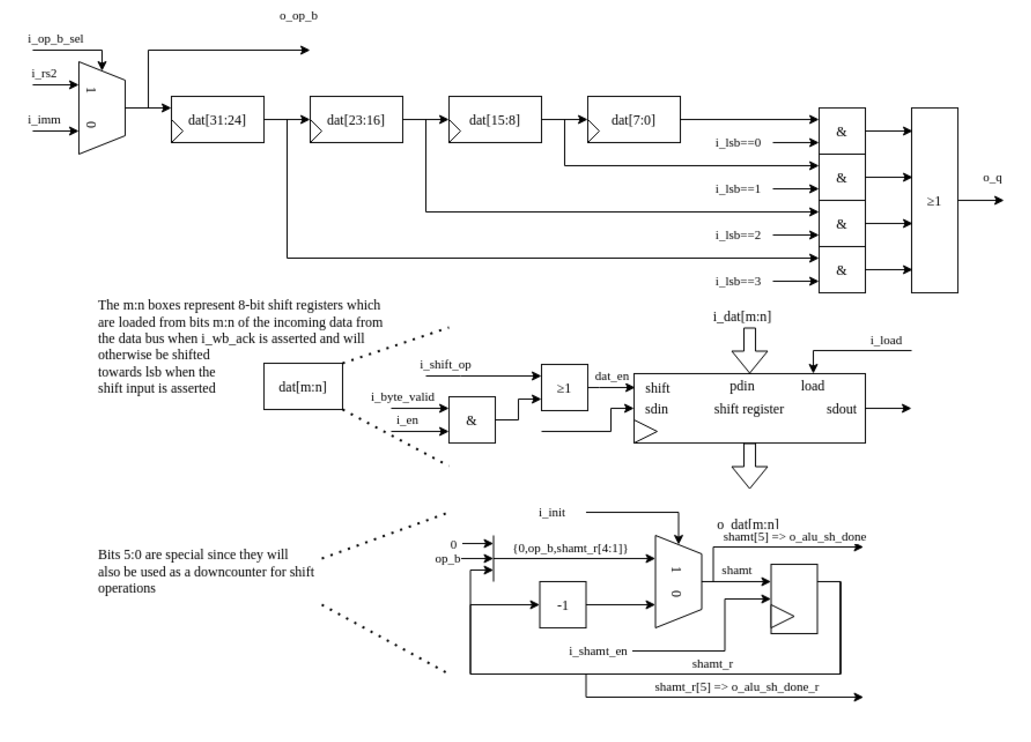
* **State Signals:**
  + i\_clk: Clock signal.
  + i\_en: Enable signal for operation.
  + i\_init: Initialization signal for starting an operation.
  + i\_cnt\_done: Indicates the end of a count cycle.
  + i\_lsb: The 2 least significant bits to control output logic.
  + i\_byte\_valid: Indicates valid byte operations.
* **Control Signals:**
  + i\_op\_b\_sel: Selects between i\_rs2 and i\_imm for the operation.
  + i\_shift\_op: Indicates a shift operation.
* **Data Signals:**
  + i\_rs2 and i\_imm: Data inputs.
  + i\_load: Load enable signal.
  + i\_dat: Data input for load operations.

**Outputs:**

* **Operational Outputs:**
  + **o\_op\_b:** Selected operand (**i\_rs2** or **i\_imm**).
  + **o\_q:** Serial data output (used during shift operations).
  + **o\_sh\_done:** Indicates that a shift operation is complete.
  + **o\_sh\_done\_r**: Indicates the current state of the shift operation.
* **External Outputs:**
  + **o\_dat:** 32-bit data register output.

**Internal Register:**

* **dat:** A 32-bit register that holds data for store, load, and shift operations.



### Behavior

1. **Operand Selection (o\_op\_b):**
   * Based **on i\_op\_b\_sel, o\_op\_b** outputs either **i\_rs2** or **i\_imm.**
2. **Enable Signal (dat\_en):**
   * Determines whether the dat register is updated. Enabled for shift operations or when valid byte operations are signaled.
3. **Data Shifting Logic (dat\_shamt):**
   * Handles the shifting behavior:
     + **Shift Operations:** In "down counter" mode, decrements the least significant 6 bits of dat to count shifts.
     + **Other Operations:** Clears bit 5 of dat during initialization, if the operation is complete (i\_cnt\_done).
4. **Shift Completion (o\_sh\_done and o\_sh\_done\_r):**
   * **o\_sh\_done:** Asserts when the 6th bit of dat wraps around to signal shift completion.
   * **o\_sh\_done\_r:** Tracks the current state of bit 5 in dat.
5. **Serial Output (o\_q):**
   * Outputs specific bits of dat serially, based on the value of i\_lsb. This allows data to be streamed out bit by bit for shift operations.
6. **Store, Load, and Shift Operations:**
   * **Store:** Data to be stored is shifted into dat during initialization (dat\_en).
   * **Load:** Latches external data (i\_dat) into dat when i\_load is asserted.
   * **Shift:** Updates dat using a combination of the selected operand (o\_op\_b) and the shifted data.
7. **Data Register Update:**
   * On every clock cycle, if dat\_en or i\_load is asserted:
     + If i\_load is enabled, dat is loaded with i\_dat.
     + Otherwise, data is shifted into dat, combining o\_op\_b and the existing contents of dat.

### Summary of Key Operations

1. **Store:**
   * Data shifted to correct positions during initialization (dat\_en).
   * Output is presented on o\_dat.
2. **Load:**
   * External data (i\_dat) is latched into dat when i\_load is high.
   * Data shifts out at the appropriate time for memory interactions.
3. **Shift:**
   * Data is shifted serially into dat.
   * Down counter (dat\_shamt) ensures the correct number of shifts and signals completion.

## serv\_state



The serv\_state module is the central state controller for the SERV processor, coordinating bit-serial execution, instruction fetch, memory access, branching, and exception handling efficiently for minimal-resource designs.

**INPUTS**

* i\_clk
* i\_rst
* i\_new\_irq
* i\_ibus\_ack
* i\_dbus\_ack
* i\_rf\_ready
* i\_take\_branch
* i\_branch\_op
* i\_mem\_op
* i\_mdu\_op
* i\_csr\_mstatus
* i\_csr\_mie
* i\_csr\_mcause
* i\_buf\_valid

**OUTPUTS**

* o\_cnt\_en
* o\_init
* o\_cnt\_done
* o\_cnt0to3
* o\_cnt0
* o\_cnt2
* o\_cnt3
* o\_cnt7
* o\_cnt15
* o\_cnt31
* o\_cnt63
* o\_rf\_rreq
* o\_rf\_wreq
* o\_ctrl\_pc\_en
* o\_ctrl\_jump
* o\_ctrl\_trap
* o\_ctrl\_misalign
* o\_ibus\_cyc
* o\_dbus\_cyc
* o\_rf\_csr\_en
* o\_mdu\_valid
* o\_sh\_done

### Key Concepts and Behaviors

**1. Parameters**

* **RESET\_STRATEGY**: Defines the reset behavior (MINI or NONE).
* **WITH\_CSR**: Determines if the Control and Status Register (CSR) functionality is included.
* **ALIGN**: Indicates if alignment checks are included for branch or memory operations.
* **MDU**: Indicates whether a Multiply-Divide Unit (MDU) is included.
* **W**: Width of the counter (1 or 4). Determines how bit-serial operations proceed.

**2. State Control**

* **i\_clk and i\_rst**: Standard clock and reset signals.
* **init\_done**: Tracks whether initialization is complete.
* **o\_init**: Indicates whether the processor is in the initialization phase.
* **o\_ctrl\_pc\_en**: Enables the program counter update.
* **o\_ctrl\_jump**: Indicates whether a jump instruction should be taken.
* **o\_ctrl\_trap**: Indicates that the processor is handling an exception (trap).

**3. Instruction Fetching**

* **o\_ibus\_cyc and i\_ibus\_ack**:
  + These signals control instruction fetching.
  + **o\_ibus\_cyc** is asserted when the processor needs to fetch a new instruction.
  + Deasserted when an instruction fetch is acknowledged (i\_ibus\_ack).

**4. Counter Mechanism (o\_cnt and cnt\_r)**

The counter tracks the bit-level progress of operations:

* **Counter Design**:
  + o\_cnt: The upper three bits of the counter.
  + cnt\_r: A 4-bit shift register representing the lower bits.
  + Together, they count from 0 to 31 (32 bits for a RISC-V instruction).
* **Benefits of this design**:
  + Reduced resource usage: Only a few bits are checked instead of all 5 bits.
  + Minimal control logic: Counting is implicitly enabled/disabled based on cnt\_r.
* **Behavior**:
  + The counter increments when i\_rf\_ready (register file ready signal) is asserted.
  + Stops when o\_cnt\_done (indicating a completed cycle) is asserted.

**5. Control Signal Generation**

* **Memory Access (o\_dbus\_cyc)**:
  + Enables the data bus cycle for memory operations if no misalignment is detected.
* **Register File Access**:
  + **o\_rf\_rreq**: Request to read the register file (triggered on instruction fetch or exceptions).
  + **o\_rf\_wreq**: Request to write back results to the register file.
* **ALU and Branch Control**:
  + **take\_branch**: Determines if a branch should be taken based on conditions:
    - Unconditional branches (opcode[0] == 1).
    - Conditional branches (opcode[0] == 0) where ALU comparison matches the branch condition.
  + **o\_bufreg\_en**: Enables the buffer register during memory, branch, and shift operations.

**6. Exceptions and Traps**

* **misalign\_trap\_sync**:
  + Handles misaligned memory or branch operations.
  + Triggers traps (o\_ctrl\_trap) when exceptions are detected.
* **trap\_pending**:
  + Captures pending exceptions during initialization.
  + Ensures proper synchronization with instruction fetches.

**7. Multiply-Divide Unit (MDU)**

If MDU is enabled:

* **o\_mdu\_valid**: Signals that MDU is ready to process.
* Works with i\_mdu\_ready to synchronize results with the rest of the pipeline.

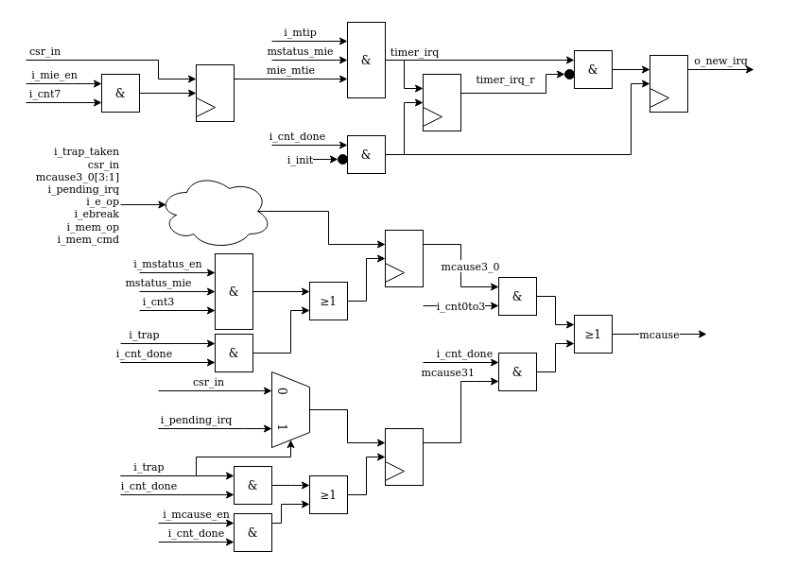
**8. Counter Variants (W == 1 or W == 4)**

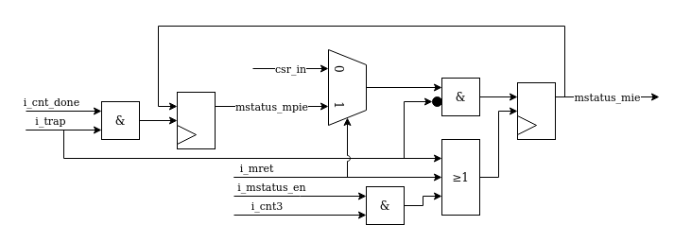
* **W == 1**:
  + A bit-serial counter using a shift register (cnt\_lsb).
  + Efficient for minimal resource usage.
* **W == 4**:
  + A simpler counter design where cnt\_r is always 1111.
  + Suitable for designs where more resources are available.

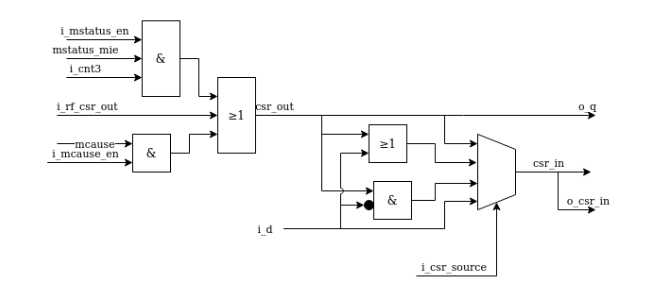
**Overall Behavior**

1. **Initialization**:
   * On reset, init\_done is cleared, and o\_init is asserted.
   * The module prepares to fetch the first instruction (o\_ibus\_cyc).
2. **Instruction Fetch**:
   * o\_ibus\_cyc asserts, and the processor waits for i\_ibus\_ack.
   * Once acknowledged, the counter starts processing the instruction bit by bit.
3. **Instruction Execution**:
   * The counter (o\_cnt and cnt\_r) steps through the instruction.
   * Relevant signals (o\_bufreg\_en, o\_rf\_rreq, etc.) control execution stages.
4. **Branching and Exceptions**:
   * Branches are taken based on take\_branch.
   * Misalignments or exceptions trigger traps (o\_ctrl\_trap).
5. **Completion**:
   * The counter signals completion (o\_cnt\_done), and the processor fetches the next instruction.

## serv\_csr





### 1. Input and Output Signals

**Inputs**

* i\_clk,
* i\_rst,

**State**

* + i\_trig\_irq,
  + i\_en,
  + i\_cnt0to3,
  + i\_cnt3,
  + i\_cnt7,
  + i\_cnt11,
  + i\_cnt12,
  + i\_cnt\_done,
  + i\_mem\_op,
  + i\_mtip,
  + i\_trap,

**Control**

* i\_e\_op,
* i\_ebreak,
* i\_mem\_cmd,
* i\_mstatus\_en,
* i\_mie\_en,
* i\_mcause\_en,
* [1:0] i\_csr\_source,
* i\_mret,
* i\_csr\_d\_sel,

**Data**

* [B:0] i\_rf\_csr\_out,
* [B:0] i\_csr\_imm,
* [B:0] i\_rs1,

**Outputs**

* + [B:0] o\_q
  + [B:0] o\_csr\_in,
  + o\_new\_irq,

### 2. Key Components

**CSR Data Sources**

The CSR input (csr\_in) is selected based on the source indicated by i\_csr\_source:

* CSR\_SOURCE\_CSR: Current CSR value is passed.
* CSR\_SOURCE\_EXT: Uses external data (i\_rs1 or i\_csr\_imm).
* CSR\_SOURCE\_SET: Performs a bitwise OR between the CSR and the source data.
* CSR\_SOURCE\_CLR: Performs a bitwise AND with the complement of the source data.

**MSTATUS (Machine Status Register)**

The mstatus register tracks:

* **MIE (Machine Interrupt Enable):** Enables global interrupt handling.
* **MPIE (Machine Previous Interrupt Enable):** Stores the value of MIE during a trap.

The behavior of mstatus:

* When a trap occurs, MIE is cleared, and its value is stored in MPIE.
* During an mret instruction, MIE is restored from MPIE.
* MIE can also be directly modified by CSR instructions.

**MCAUSE (Machine Cause Register)**

The mcause register records the cause of a trap or interrupt. It includes:

* **Interrupt Indicator (bit 31):** Differentiates between interrupts and exceptions.
* **Exception Code (bits 0-3):** Encodes the reason for the trap.

Key behaviors:

* On an external interrupt, the exception code is set to 7 (timer interrupt).
* For exceptions:
  + **Ebreak:** Code = 3.
  + **Ecall:** Code = 11.
  + **Misaligned memory access:** Code = 4 (load), 6 (store), 0 (jump).

**Interrupt Handling**

* **Timer Interrupts:**
  + timer\_irq: Asserted when a timer interrupt is pending (i\_mtip) and interrupts are enabled (mstatus\_mie & mie\_mtie).
  + o\_new\_irq: Raised when a new interrupt is detected (edge-triggered).

**CSR Output (o\_q)**

* The CSR output depends on the selected CSR (mstatus, mie, mcause) and updates are gated by the enable signals (i\_mstatus\_en, i\_mie\_en, i\_mcause\_en).

### 3. Detailed Behavioral Breakdown

**Interrupt Handling**

1. **Detecting Timer Interrupts:**
   * A timer interrupt (timer\_irq) is detected based on the state of mstatus\_mie, mie\_mtie, and i\_mtip.
   * The rising edge of timer\_irq is captured using timer\_irq\_r to generate o\_new\_irq.
2. **Raising o\_new\_irq:**
   * If a timer interrupt is detected (timer\_irq = 1), and it wasn’t previously active (timer\_irq\_r = 0), o\_new\_irq is set to 1.

**Trap Handling**

1. When a trap occurs (i\_trap & i\_cnt\_done):
   * MIE is cleared, and its value is saved in MPIE.
   * MCAUSE is updated with the appropriate exception code.
2. During mret:
   * MIE is restored from MPIE.

**CSR Updates**

1. The CSR being modified is determined by the enable signals (i\_mstatus\_en, i\_mie\_en, i\_mcause\_en).
2. Updates are gated by i\_en and counters (i\_cnt\*) to ensure correct timing for serial operations.

**Default States**

On reset (i\_rst):

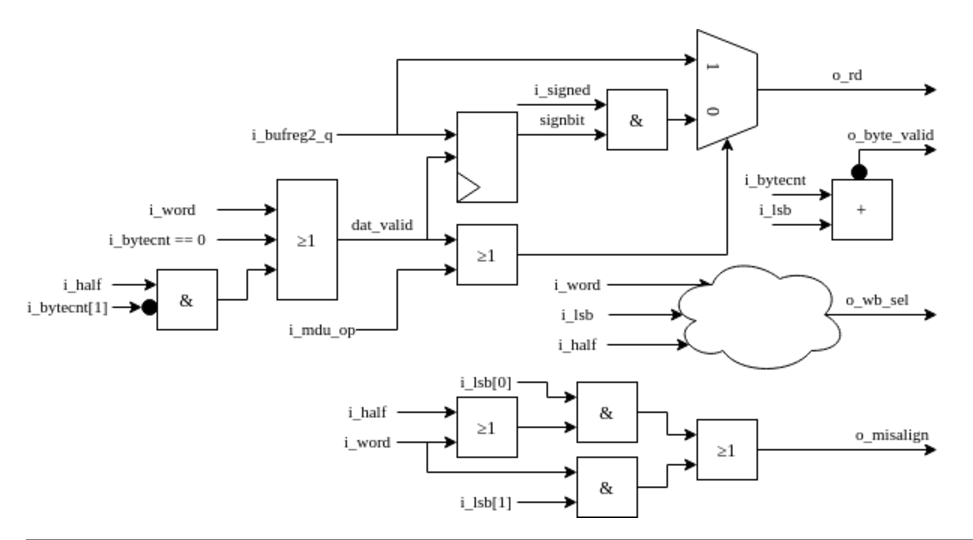
* o\_new\_irq, mie\_mtie are cleared.
* If the reset strategy is "NONE", the CSRs retain their previous values.

**4. Summary**

* **Interrupts**: Timer interrupts and edge-triggered detection (o\_new\_irq).
* **Traps**: Saving/restoring status (mstatus), recording causes (mcause).
* **CSR Operations**: Supporting various operations (read/write/set/clear) using configurable data sources.

## Serv\_mem\_if





### Parameters

1. **WITH\_CSR**: Enables or disables misalignment checks (o\_misalign) when the SERV core is built with Control and Status Registers (CSR).
2. **W**: Determines the bit width of the data bus.
3. **B**: Represents the most significant bit of the bus, calculated as W-1.

### Inputs and Outputs

**Inputs:**

* **i\_clk**: The clock signal.
* **i\_bytecnt**: Byte counter indicating the current byte of operation (0 to 3).
* **i\_lsb**: Least significant bits of the address; used to determine alignment.
* **i\_signed**: Indicates if the operation requires sign extension for signed data.
* **i\_word**: Signals a 4-byte word operation.
* **i\_half**: Signals a 2-byte half-word operation.
* **i\_mdu\_op**: Indicates if the Multiply-Divide Unit (MDU) is operating.
* **i\_bufreg2\_q**: Data from the buffer register.

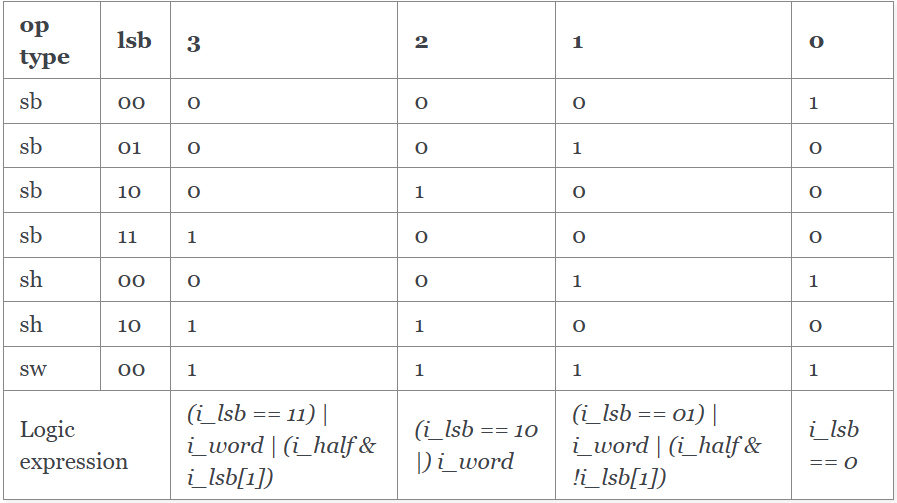
**Outputs:**

* **o\_byte\_valid**: Signals whether the current byte is valid for writing or reading.
* **o\_misalign**: Indicates misalignment during a load/store operation.
* **o\_rd**: Data to be read after alignment and sign extension.
* **o\_wb\_sel**: Write-back selection signal, indicating which byte(s) of the word to write to memory.

### Functionality

1. **Byte Validation (o\_byte\_valid)**:
   * Determines if the current byte in the transaction is valid based on i\_lsb (address alignment) and i\_bytecnt (byte position in operation).
   * Implements a complex condition optimized for synthesis tools.
2. **Data Validity (dat\_valid)**:
   * Ensures data is valid for reading or writing based on operation type:
     + **i\_mdu\_op**: MDU operations.
     + **i\_word**: Word operations (4 bytes).
     + **i\_half**: Half-word operations (2 bytes).
     + **i\_bytecnt == 2'b00**: First byte always valid.
3. **Sign Extension (o\_rd)**:
   * For valid data (dat\_valid), outputs the value of i\_bufreg2\_q.
   * For invalid data, extends the most significant bit (signbit) if i\_signed is enabled, filling the word with signbit.
4. **Write-Back Selection (o\_wb\_sel)**:
   * Determines which byte(s) of the word to write to memory based on alignment (i\_lsb):
     + Byte 0 (o\_wb\_sel[0]): Active when i\_lsb == 2'b00.
     + Byte 1 (o\_wb\_sel[1]): Active when i\_lsb == 2'b01 or in half/word operations.
     + Byte 2 (o\_wb\_sel[2]): Active when i\_lsb == 2'b10 or in word operations.
     + Byte 3 (o\_wb\_sel[3]): Active when i\_lsb == 2'b11, in word operations, or in half operations targeting bytes 2-3.
5. **Misalignment Check (o\_misalign)**:
   * When WITH\_CSR is enabled, misalignment occurs if:
     + **i\_word**: Address isn't aligned to a 4-byte boundary (i\_lsb[1:0] != 00).
     + **i\_half**: Address isn't aligned to a 2-byte boundary (i\_lsb[0] != 0).
6. **Signbit Update (signbit)**:
   * Updates signbit to the most significant bit of the buffer register (i\_bufreg2\_q[B]) for use in sign extension during invalid reads.

The [Data bus byte mask](https://serv.readthedocs.io/en/latest/internals.html#data-bus-byte-mask) table summarizes the logic for when the individual byte select signals are asserted depending on the two LSB of the data address together with the size (byte, halfword, word) of the write operation.

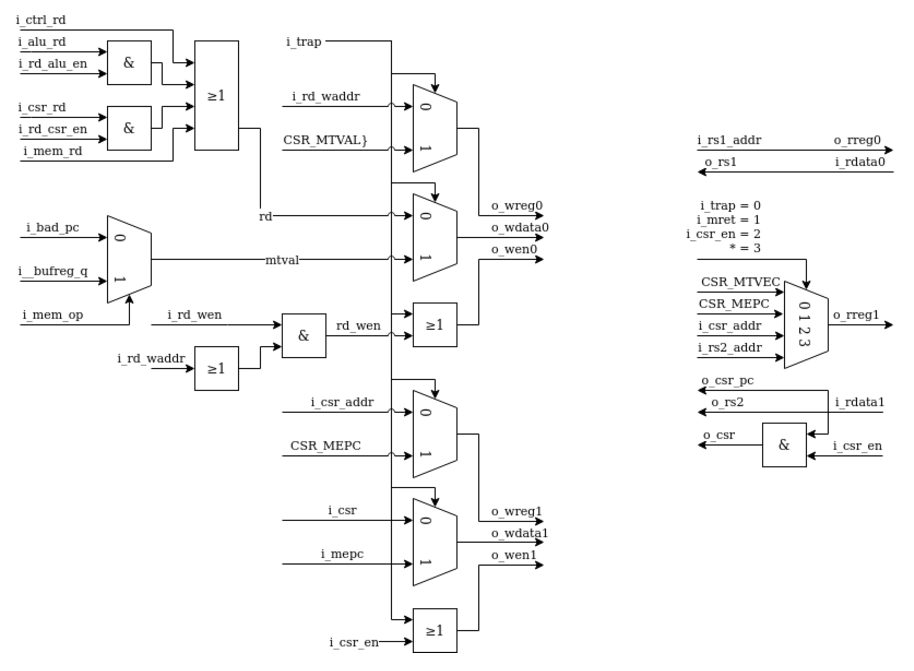


### Key Points

* The module supports **load/store alignment** and **sign extension** for operations targeting bytes, half-words, and words.
* It ensures **alignment checks** are performed when enabled (via WITH\_CSR).
* **Write-back control (o\_wb\_sel)** dynamically selects which bytes to write based on the operation type and alignment.

## Serv\_rf\_if





**Parameters**

1. **WITH\_CSR**: Enables or disables misalignment checks (o\_misalign) when the SERV core is built with Control and Status Registers (CSR).
2. **W**: Determines the bit width of the data bus.
3. **B**: Represents the most significant bit of the bus, calculated as W-1.

### Inputs and Outputs

**Inputs:**

* **i\_clk**: The clock signal.
* **i\_bytecnt**: Byte counter indicating the current byte of operation (0 to 3).
* **i\_lsb**: Least significant bits of the address; used to determine alignment.
* **i\_signed**: Indicates if the operation requires sign extension for signed data.
* **i\_word**: Signals a 4-byte word operation.
* **i\_half**: Signals a 2-byte half-word operation.
* **i\_mdu\_op**: Indicates if the Multiply-Divide Unit (MDU) is operating.
* **i\_bufreg2\_q**: Data from the buffer register.

**Outputs:**

* **o\_byte\_valid**: Signals whether the current byte is valid for writing or reading.
* **o\_misalign**: Indicates misalignment during a load/store operation.
* **o\_rd**: Data to be read after alignment and sign extension.
* **o\_wb\_sel**: Write-back selection signal, indicating which byte(s) of the word to write to memory.

### Functionality

1. **Byte Validation (o\_byte\_valid)**:
   * Determines if the current byte in the transaction is valid based on i\_lsb (address alignment) and i\_bytecnt (byte position in operation).
   * Implements a complex condition optimized for synthesis tools.
2. **Data Validity (dat\_valid)**:
   * Ensures data is valid for reading or writing based on operation type:
     + **i\_mdu\_op**: MDU operations.
     + **i\_word**: Word operations (4 bytes).
     + **i\_half**: Half-word operations (2 bytes).
     + **i\_bytecnt == 2'b00**: First byte always valid.
3. **Sign Extension (o\_rd)**:
   * For valid data (dat\_valid), outputs the value of i\_bufreg2\_q.
   * For invalid data, extends the most significant bit (signbit) if i\_signed is enabled, filling the word with signbit.
4. **Write-Back Selection (o\_wb\_sel)**:
   * Determines which byte(s) of the word to write to memory based on alignment (i\_lsb):
     + Byte 0 (o\_wb\_sel[0]): Active when i\_lsb == 2'b00.
     + Byte 1 (o\_wb\_sel[1]): Active when i\_lsb == 2'b01 or in half/word operations.
     + Byte 2 (o\_wb\_sel[2]): Active when i\_lsb == 2'b10 or in word operations.
     + Byte 3 (o\_wb\_sel[3]): Active when i\_lsb == 2'b11, in word operations, or in half operations targeting bytes 2-3.
5. **Misalignment Check (o\_misalign)**:
   * When WITH\_CSR is enabled, misalignment occurs if:
     + **i\_word**: Address isn't aligned to a 4-byte boundary (i\_lsb[1:0] != 00).
     + **i\_half**: Address isn't aligned to a 2-byte boundary (i\_lsb[0] != 0).
6. **Signbit Update (signbit)**:
   * Updates signbit to the most significant bit of the buffer register (i\_bufreg2\_q[B]) for use in sign extension during invalid reads.

**Key Points**

* The module supports **load/store alignment** and **sign extension** for operations targeting bytes, half-words, and words.
* It ensures **alignment checks** are performed when enabled (via WITH\_CSR).
* **Write-back control (o\_wb\_sel)** dynamically selects which bytes to write based on the operation type and alignment