
SERV

Aug 09, 2021

Contents:

1	Modules	1
1.1	serv_rf_top	2
1.2	serv_top	2
1.3	serv_alu	2
1.4	serv_bufreg	4
1.5	serv_csr	5
1.6	serv_ctrl	7
1.7	serv_decode	8
1.8	serv_immdec	9
1.9	serv_mem_if	11
1.10	serv_rf_if	13
1.11	serv_rf_ram	14
1.12	serv_rf_ram_if	14
1.13	serv_state	14
1.14	shift_reg	14
1.15	serv_shift	15
2	Instruction life cycle	17
2.1	Fetch	19
2.2	Decode	19
2.3	Execute	19

SERV is a bit-serial CPU which means that the internal datapath is one bit wide. *SERV internal dataflow* show the internal dataflow. For each instruction, data is read from the register file or the immediate fields of the instruction word and the result of the operation is stored back into the register file. Reading and writing memory is handled through the memory interface module.

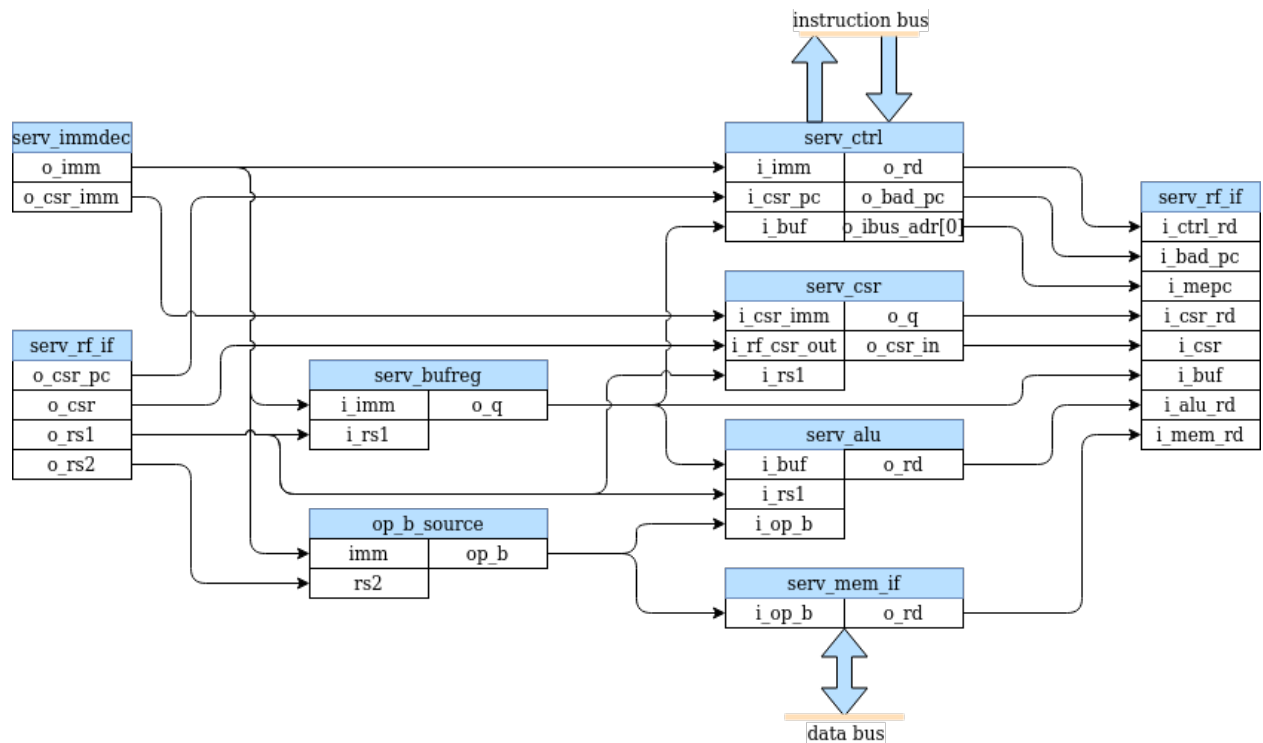
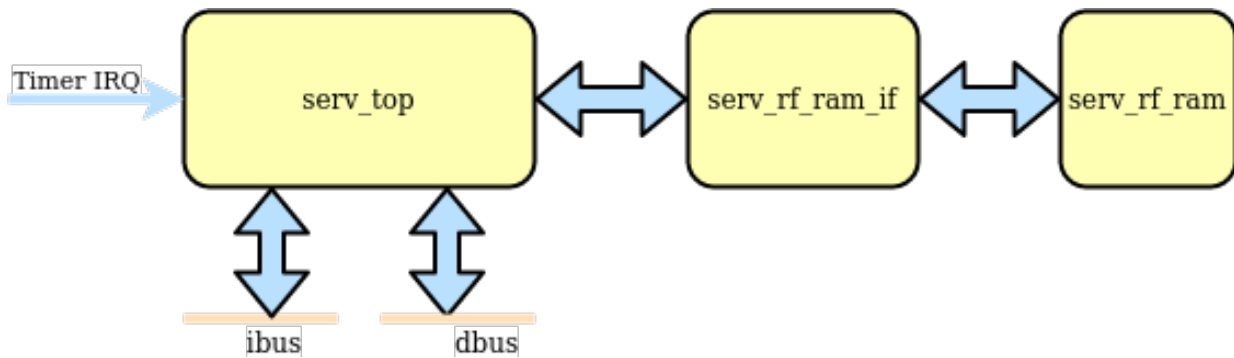


Fig. 1: SERV internal dataflow

1.1 serv_rf_top

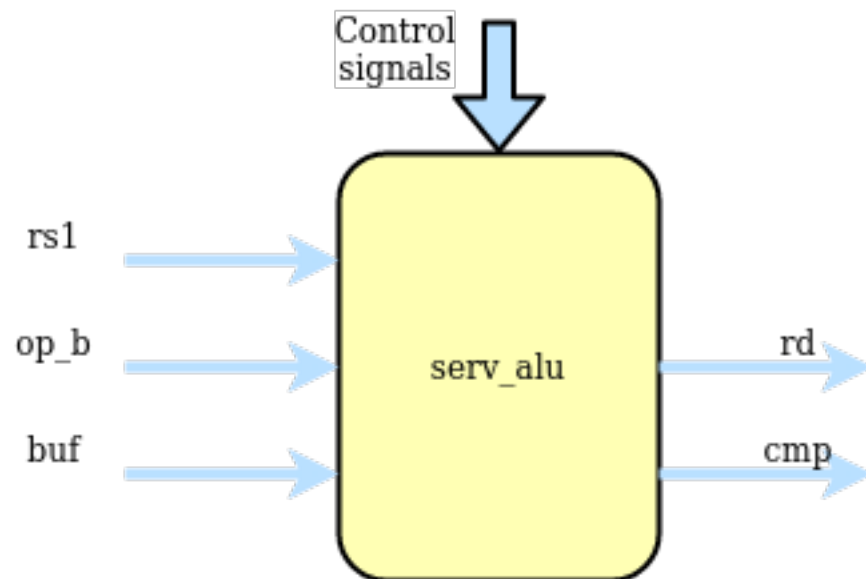


`serv_rf_top` is a top-level convenience wrapper that includes SERV and the default RF implementation and just exposes the timer IRQ and instruction/data wishbone buses.

1.2 serv_top

`serv_top` is the top-level of the SERV core without an RF

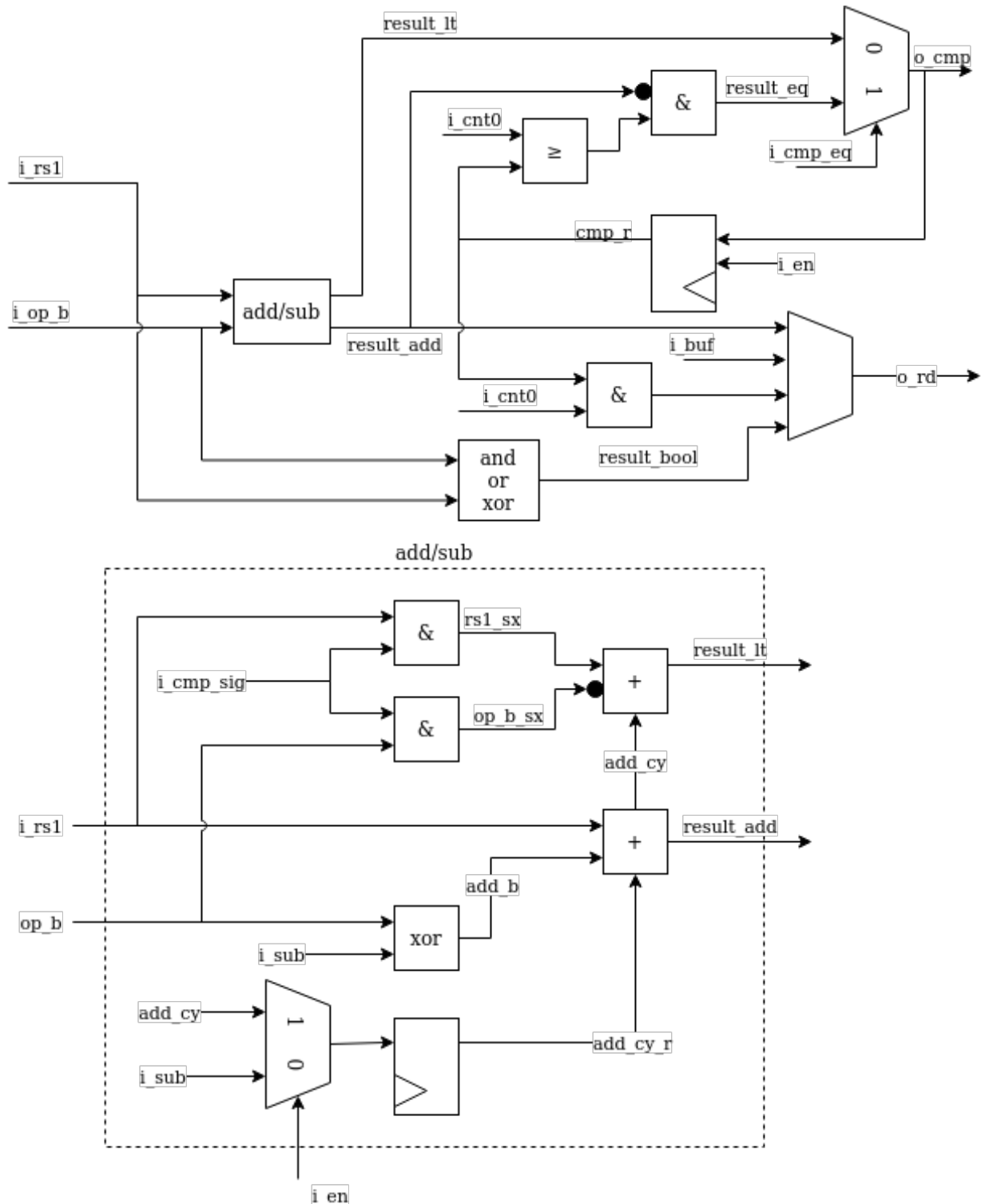
1.3 serv_alu



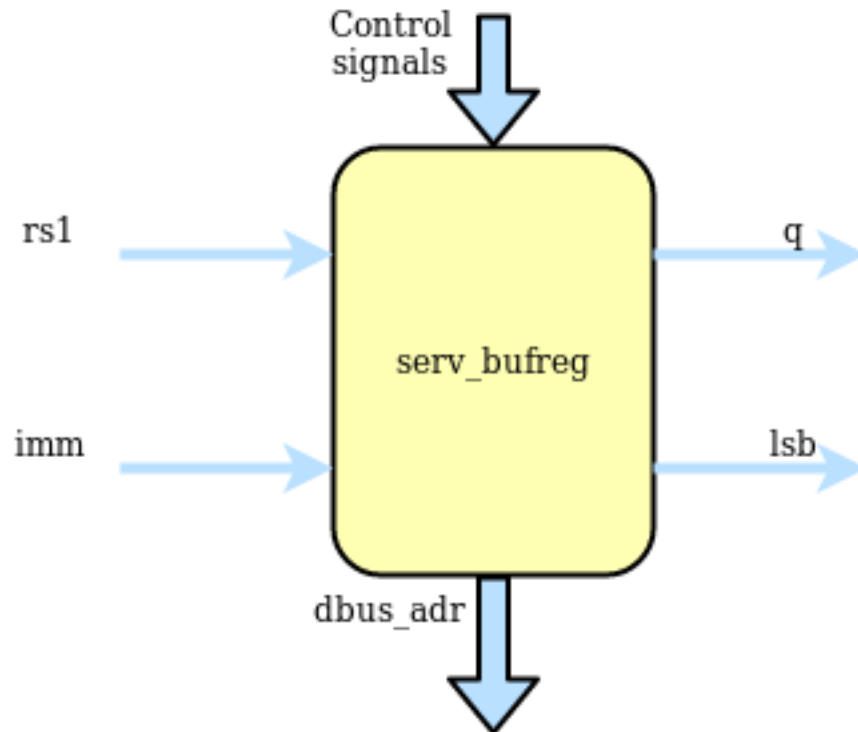
`serv_alu` handles alu operations. The first input operand (A) comes from `i_rs1` and the second operand (B) comes from `i_rs2` or `i_imm` depending on the type of operation. The data passes through the add/sub or bool logic unit and finally ends up in `o_rd` to be written to the destination register. The output `o_cmp` is used for conditional branches to decide whether or not to take the branch.

The add/sub unit can do additions $A+B$ or subtractions $A-B$ by converting it to $A+B+1$. Subtraction mode (`i_sub = 1`) is also used for the comparisons in the `slt*` and conditional branch instructions. The $+1$ used in subtraction mode is done by preloading the carry input with 1. Less-than comparisons are handled by converting the expression $A < B$ to $A - B < 0$ and checking the MSB, which will be set when the result is less than 0. This however requires sign-extending

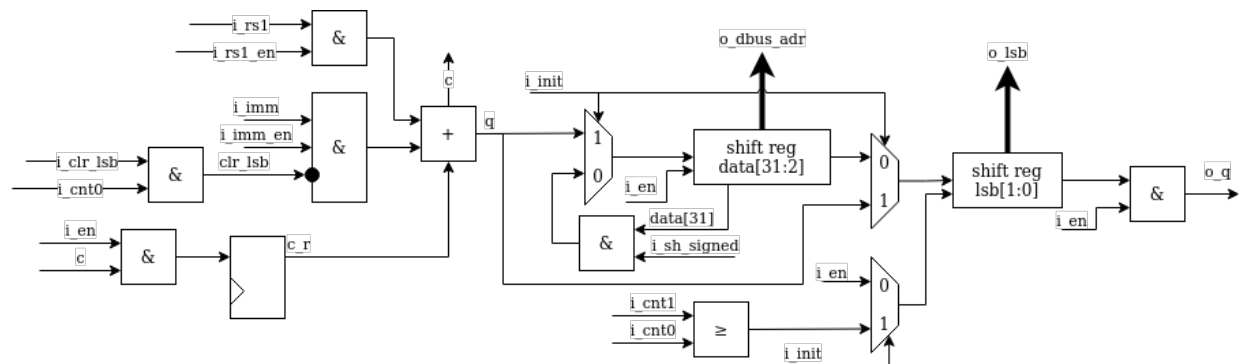
the operands to 33-bit inputs. For signed operands (when `i_cmp_sig` is set), the extra bit is the same as the MSB. For unsigned, the extra bit is always 0. Because the ALU is only active for 32 cycles, the 33rd bit must be calculated in parallel to the ordinary addition. The result from this operations is available in `result_lt`. For equality checks, `result_eq` checks that all bits are 0 from the subtraction.



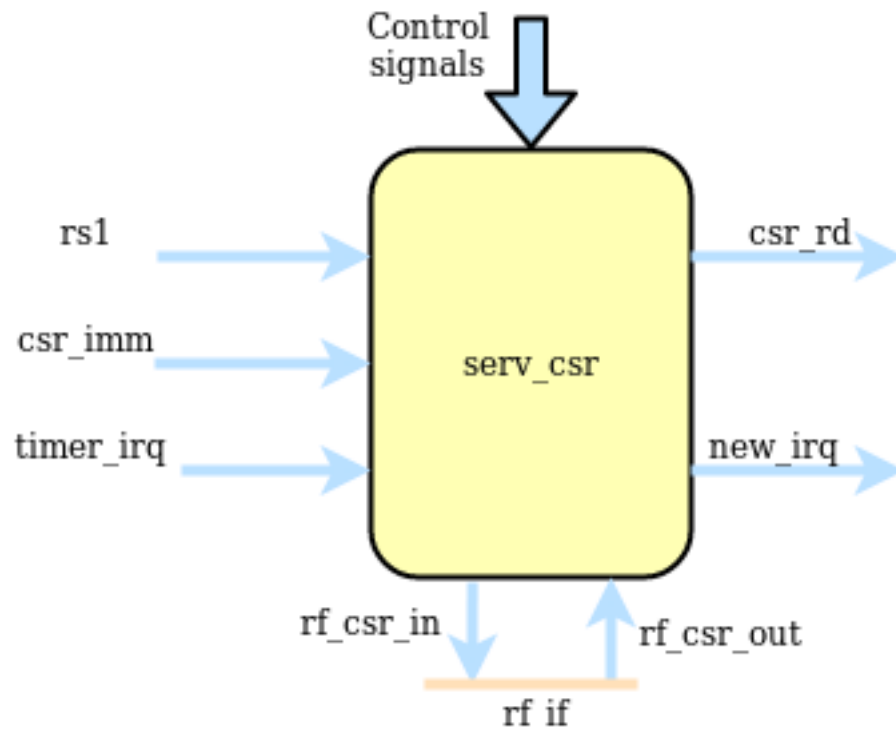
1.4 serv_bufreg



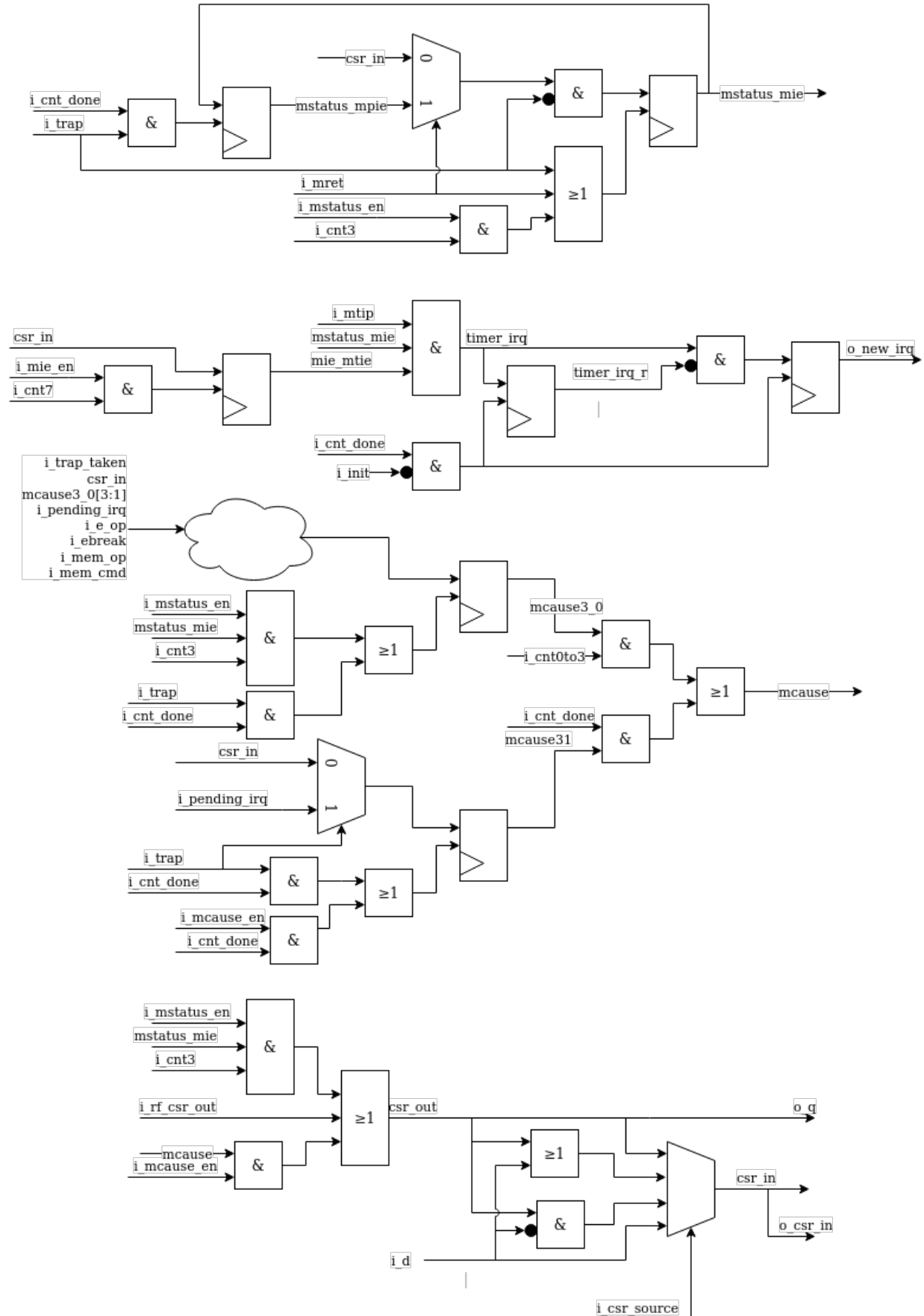
For two-stage operations, **serv_bufreg** holds data between stages. This data can be the effective address for branches or load/stores or data to be shifted for shift ops. It has a serial output for streaming out results during stage two and a parallel output that forms the dbus address. **serv_bufreg** also keeps track of the two lsb when calculating addresses. This is used to check for alignment errors. In order to support these different modes, the input to the shift register can come from **rs1**, the immediate (**imm**), **rs1+imm** or looped back from the shift register output. The latter is used for shift operations. For some operations, the LSB of the immediate is cleared before written to the shift register. The two LSB of the shift register are special. When the shift register is loaded, these two get written first before the rest of the register is filled up. This allows the memory interface to check data/address alignment early.



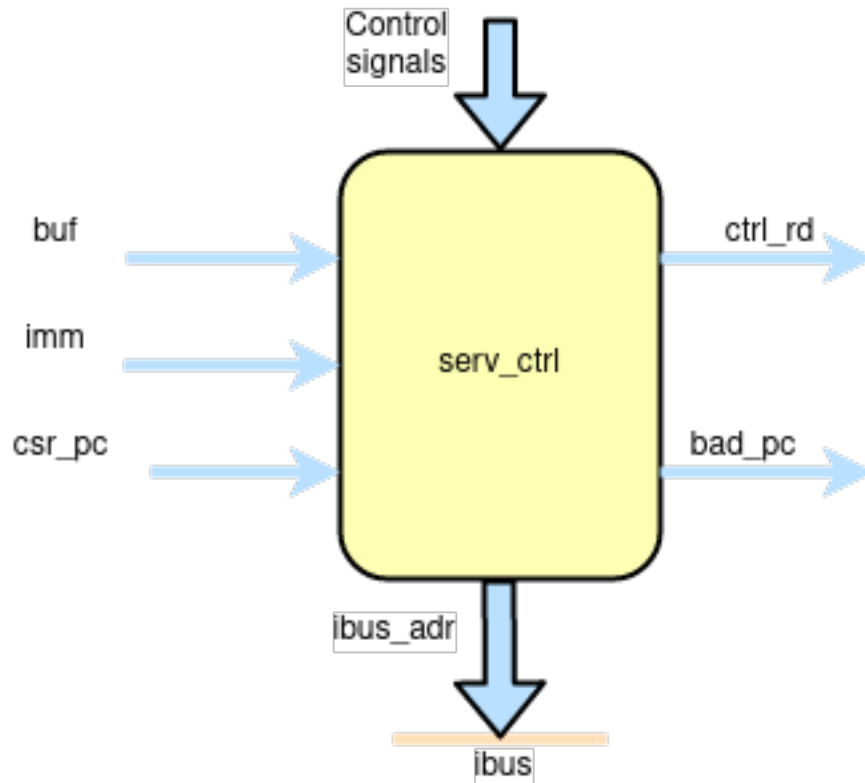
1.5 serv_csr



`serv_csr` handles CSR accesses and all status related to (timer) interrupts. Out of the eight CSRs supported by SERV, only four resides in `serv_csr` (`mstatus`, `mie`, `mcause` and `mip`) and for those registers, SERV only implement the bits required for `ecall`, `ebreak`, misalignment and timer interrupts. The four remaining CSRs are commonly stored in the RF



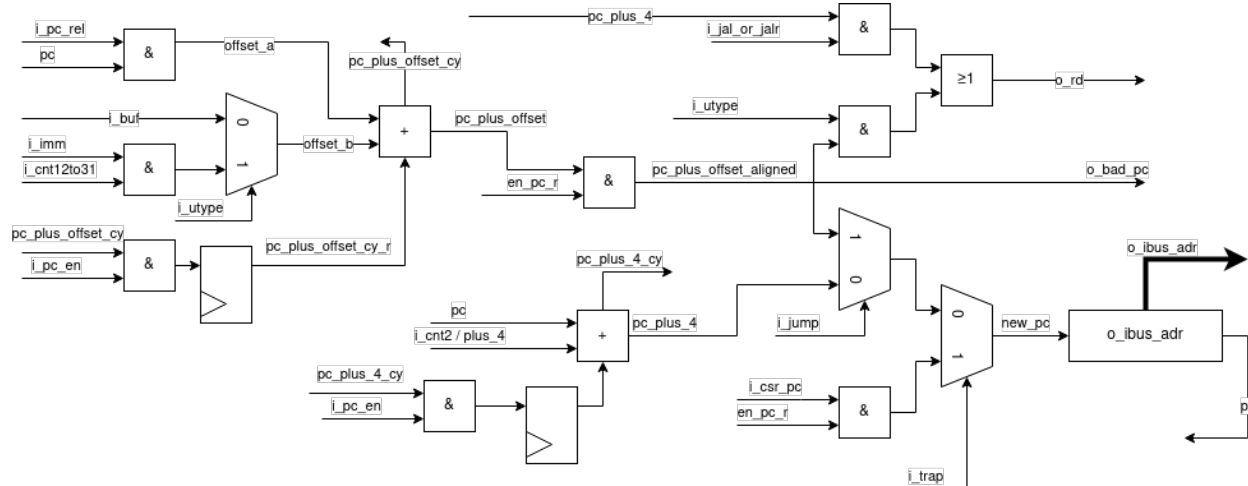
1.6 serv_ctrl



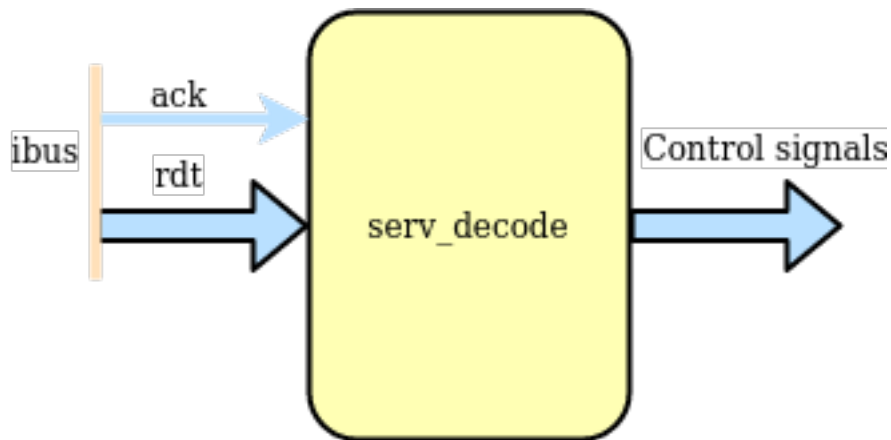
serv_ctrl keeps track of the current PC and contains the logic needed to calculate the next PC. The PC is stored in shift register with a parallel output connected to the instruction bus.

The new PC can come from three sources. For normal instructions, it is incremented by four, which is the next 32-bit address. Jumps can be absolute or relative to the current PC. Absolute jumps are precalculated in serv_bufreg and written directly to the PC. PC relative jumps have the offset part precalculated in serv_bufreg which gets added to the current PC before storing as the new PC. The third source for the new PC comes from the CSR registers when entering or returning traps.

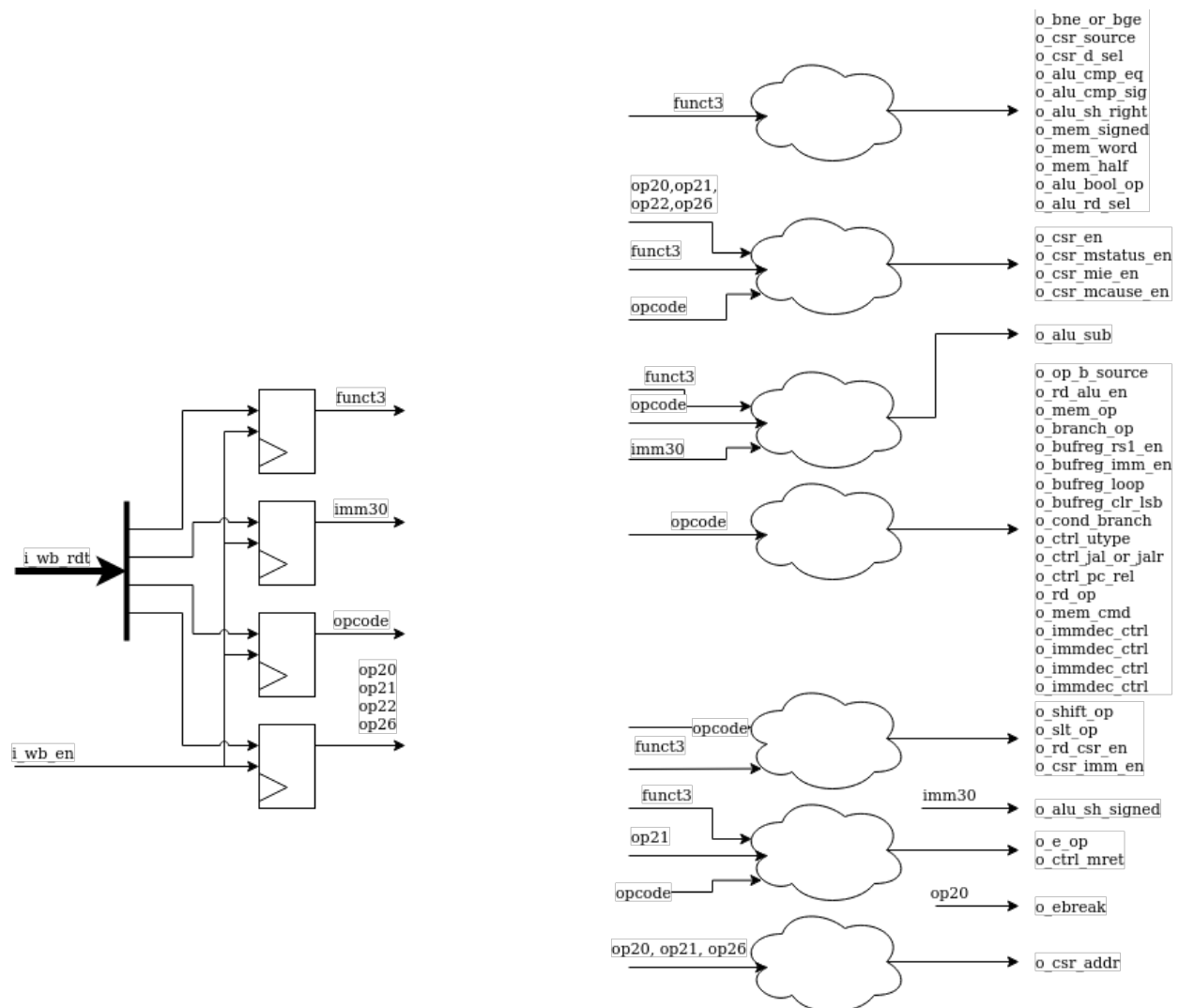
Some operations (LUI, AUIPC, jumps, entering or returning from traps) also update the destination register through o_rd. In the case of misaligned instruction traps, the invalid PC is written to o_bad_pc to be passed to mtval.



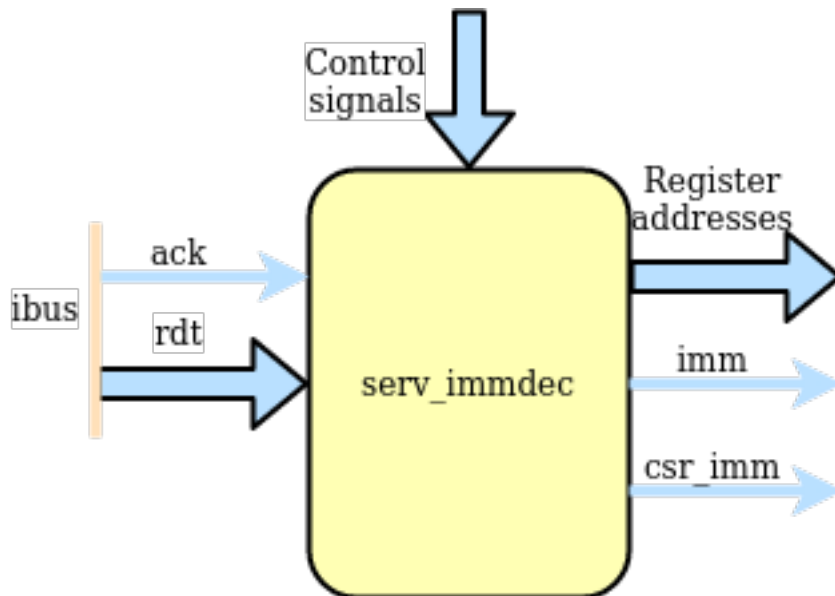
1.7 serv_decode



serv_decode is responsible for decoding the operation word coming from ibus into a set of control signals that are used internally in SERV.



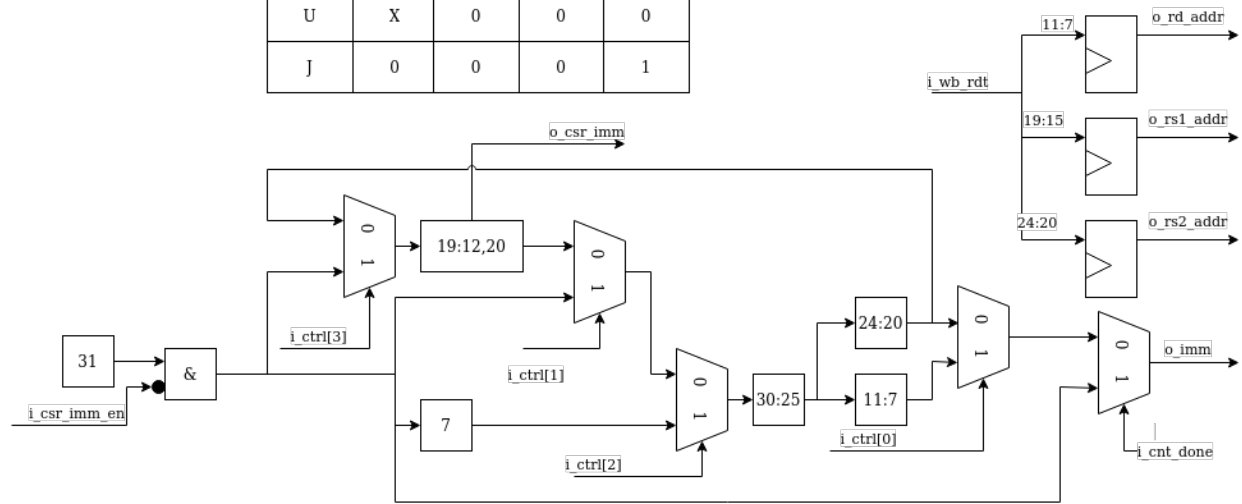
1.8 serv_immdec



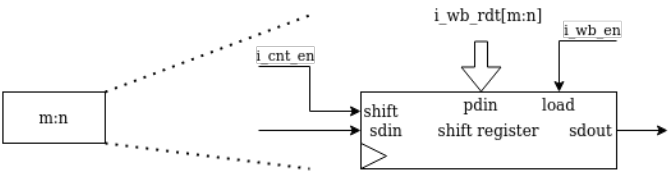
The main responsibility of serv_immdec is to stitch together the pieces of immediates from the instruction word and push it out in the correct order. When a new instruction arrives, the relevant parts are placed into a number of shift registers, and the connections between the registers are setup differently depending on the type of operation.

serv_immdec also extracts the register addresses from the operation word.

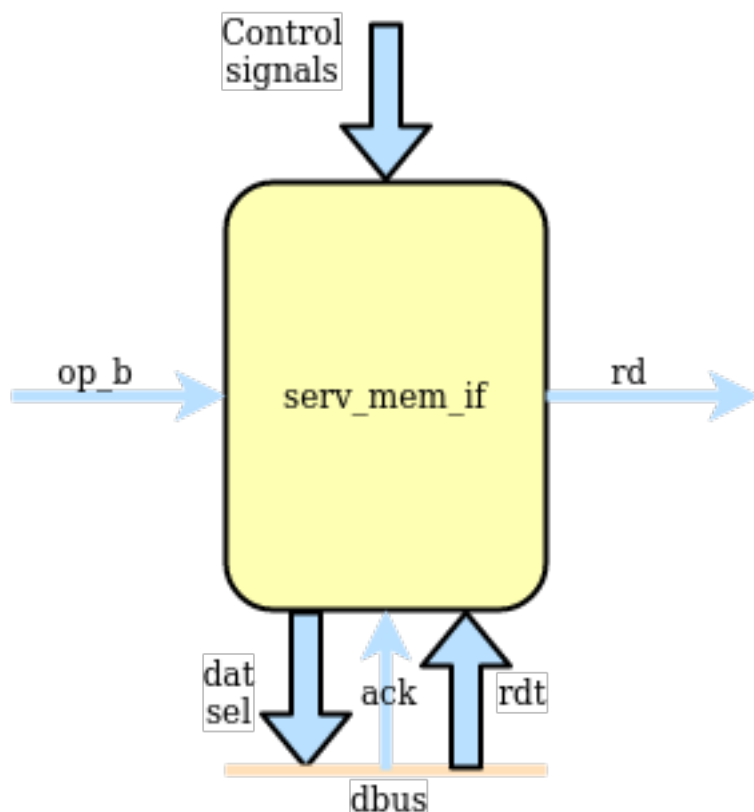
Optype	ctrl[0]	ctrl[1]	ctrl[2]	ctrl[3]
I	0	1	0	X
S	1	1	0	X
B	1	X	1	X
U	X	0	0	0
J	0	0	0	1



The m:n boxes represent shift registers of width m-n+1 which are loaded from bits m:n of the opcode when i_wb_en is asserted and will otherwise be shifted from towards lsb when the shift input is asserted



1.9 serv_mem_if



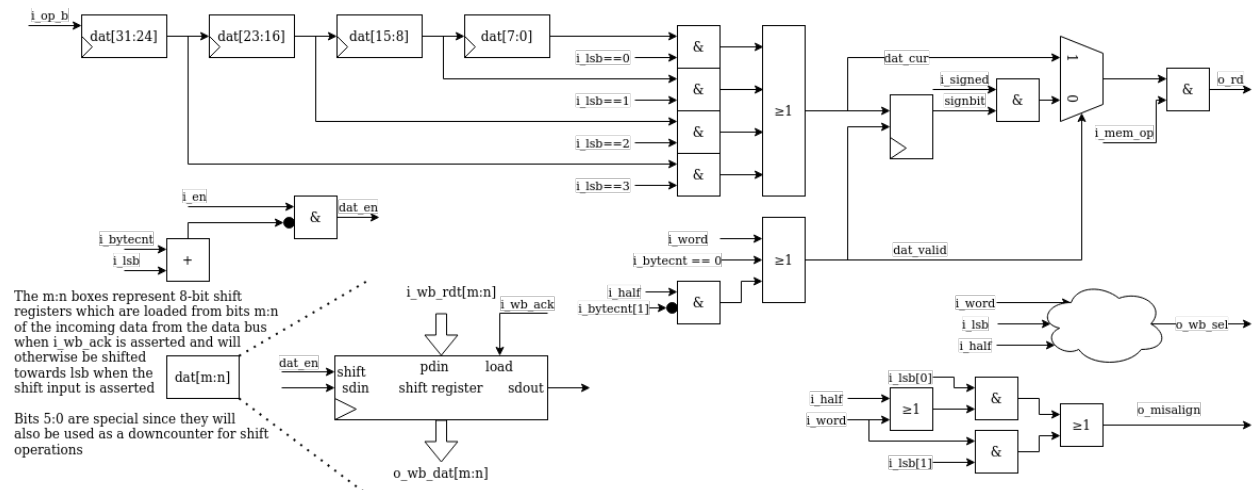
`serv_mem_if` prepares the data to be sent out on the `dbus` during store operations and serializes the incoming data during loads

The memory interface is centered around four byte-wide shift registers connected in series. During store operations, the `dat_en` signal is asserted long enough to shift in the data from `rs2` to the right place in the shift registers and the parallel output of the shift registers is then presented to the data bus as a 32-bit word together with a byte mask. The [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.

During load operations, the data from the bus is latched into the shift registers. `dat_en` is again asserted to shift out data from the registers. `i_lsb` decides from which byte stage of the shift register to tap the data, depending on the alignment of the received data. The `dat_valid` signal makes sure to only present valid data to `o_rd` and otherwise fill in with zeros or sign extension.

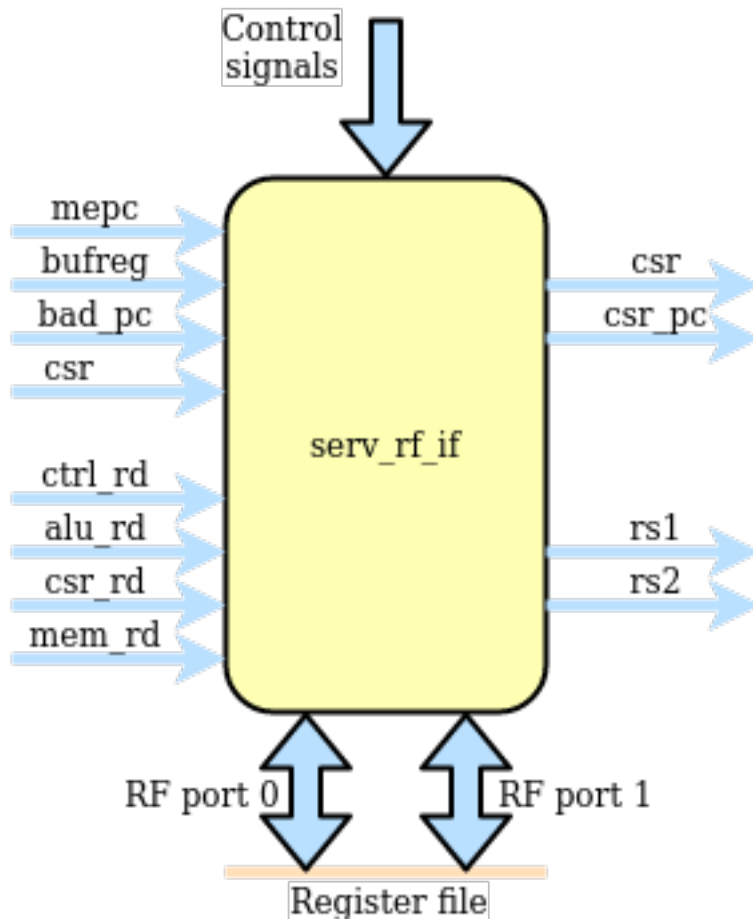
When SERV is built with `WITH_CSR`, there is also logic to detect misaligned accesses which asserts the `o_misalign` flag to the core.

The shift register used for stores and loads are also used to keep track of the number of steps to shift for left/right shift operations. In this mode, the six LSB of the register is loaded with the shift amount during the init stage and the used as a down counter which raises `o_sh_done` and `o_sh_done_r` when the number of shift steps have been completed.

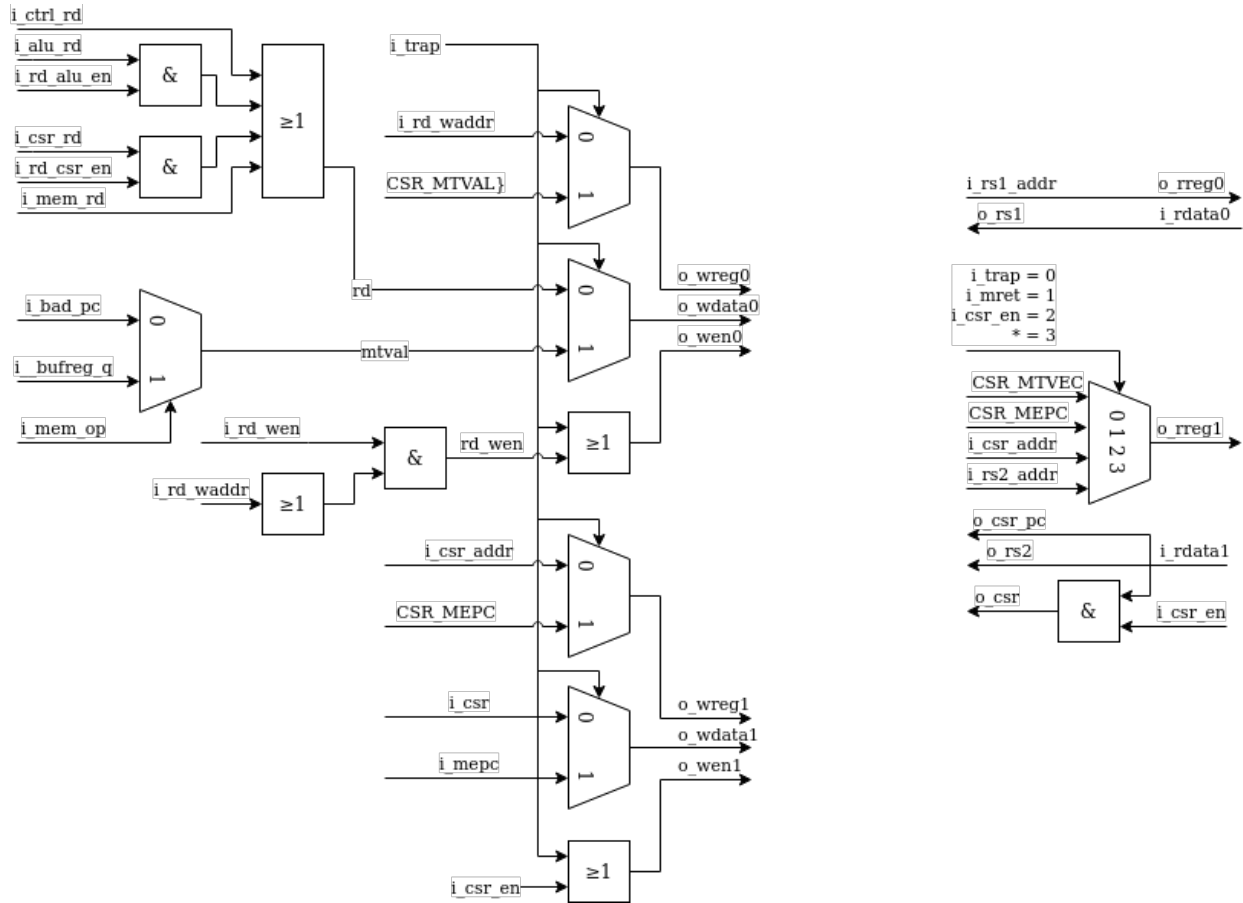


op type	lsb	3	2	1	0
sb	00	0	0	0	1
sb	01	0	0	1	0
sb	10	0	1	0	0
sb	11	1	0	0	0
sh	00	0	0	1	1
sh	10	1	1	0	0
sw	00	1	1	1	1
Logic expression		$(i_lsb == 11) \vee i_word \vee (i_half \& i_lsb[1])$	$(i_lsb == 10 \vee i_word$	$(i_lsb == 01) \vee i_word \vee (i_half \& !i_lsb[1])$	$i_lsb == 0$

1.10 serv_rf_if



`serv_rf_if` is the gateway between the core and an RF implementation. It transforms all control signals that affect register reads or writes and exposes two read and write ports to the RF. This allows implementors to plug in an RF implementation that is best suited for the technology to be used. The general purpose registers are allocated to address 0-31. In addition, four CSR are defined at addresses 32-35.



1.11 serv_rf_ram

serv_rf_ram is the default RF implementation using an SRAM-like interface. Suitable for FPGA implementations

1.12 serv_rf_ram_if

serv_rf_ram_if converts between the SERV RF IF and the serv_rf_ram interface

1.13 serv_state

serv_state keeps track of the state for the core and contains all dynamic control signals during an operations life time. Also controls the accesses towards the RF, ibus and dbus

New instructions are fetched by asserting o_ibus_cyc until there is a response on i_ibus_ack. Instruction fetches occur when the reset signal is deasserted, which is what gets SERV started, or when the PC has finished updating its value.

1.14 shift_reg

shift_reg is a shift register implementation used in various places in SERV

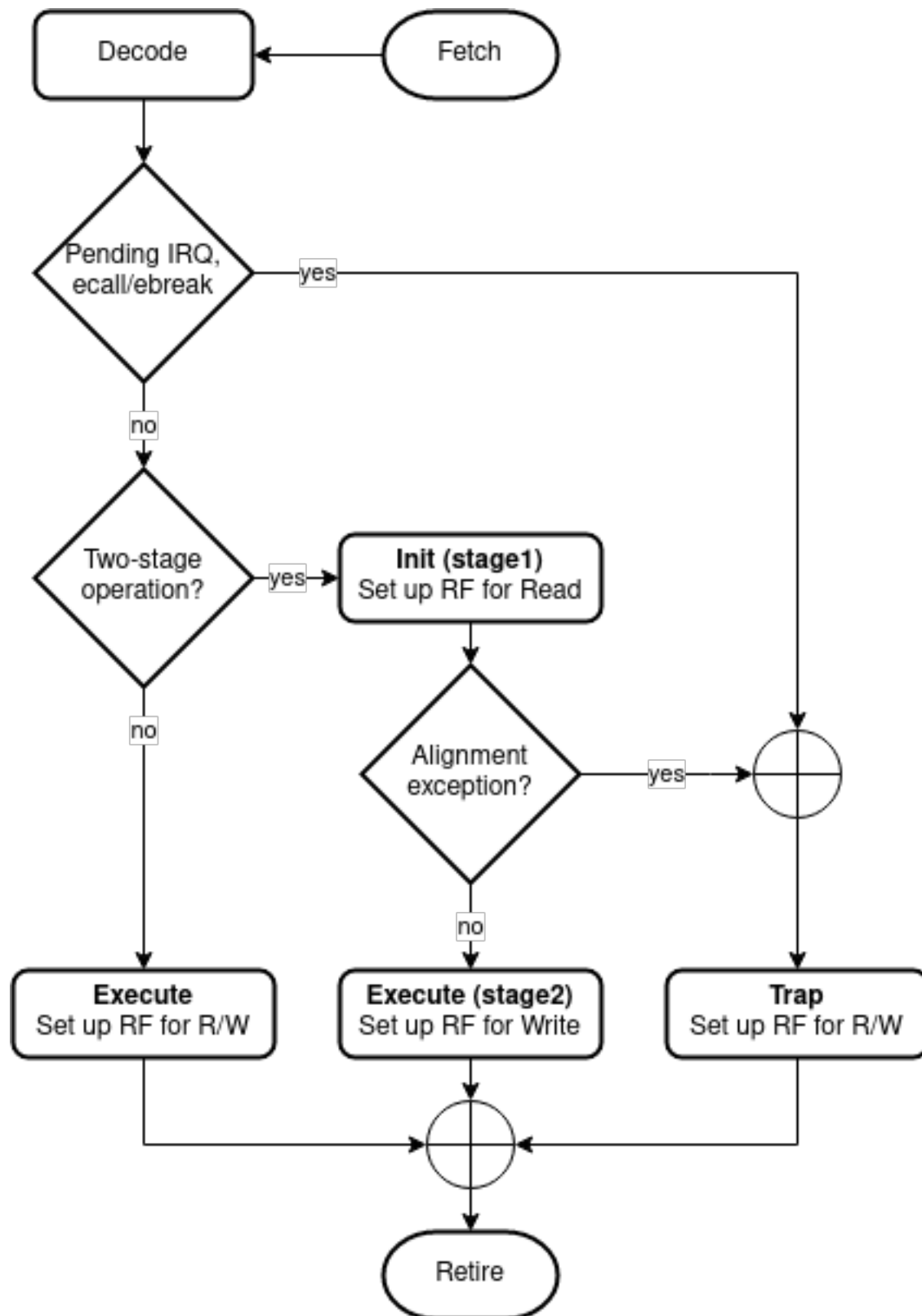
1.15 serv_shift

serv_shift lives inside the ALU and contains the control logic for shift operations

CHAPTER 2

Instruction life cycle

The life cycle of an instruction starts by the core issuing a request for a new instruction on the ibus and ends when the PC has been updated with the address of the next instruction. This section goes through what happens between those points for the various types of instructions. SERV distinguishes between two-stage and one-stage operations with the former category being all jump (branch), shift, slt and load/store instructions and the latter all other operations. In addition to this, exceptions are a special case. Only two-stage operations (jump, load/store) can cause an exception. Regardless of instruction type, they all start out the same way.



2.1 Fetch

The bus requests begin by SERV raising `o_ibus_cyc` until the memory responds with an `i_ibus_ack` and presents the instruction on `i_ibus_rdt`. Upon seeing the ack, SERV will lower `cyc` to indicate the end of the bus cycle.

2.2 Decode

When the ack appears, two things happen in SERV. The relevant portions of the instruction such as opcode, funct3 and immediate value are saved in `serv_decode` and `serv_immdec`. The saved bits of the instruction is then decoded to create the internal control signals that corresponds to the current instruction. The decoded control signals remain static throughout the instruction life cycle.

The other thing to happen is that a request to start accessing the register file is sent by strobing `rf_rreq` which prepares the register file for both read and write access.

The interface between the core and the register file is described in a protocol where the core strobes `rreq` and present the registers to read on the following cycle. The register file will prepare to stream out data bit from the two requested registers. The cycle before it sends out the first bit (LSB) it will strobe `rf_ready`. Writes work in a similar way in that the registers to write has to be presented the cycle after `rf_wreq` is strobed and that the register file will start accepting data the cycle after it has strobed `rf_ready`. Note that the delay between `rf_wreq` and `rf_ready` does not have to be the same as from `rf_rreq` to `rf_ready`. Also note that register data will only be written to a register if the corresponding write enable signal is asserted. In the diagram below, only register `r0` will be written to.

2.3 Execute

After the instruction has been decoded and the register file prepared for reads (and possibly writes) the core knows whether it is a one-stage or two-stage instruction. These are handled differently and we will begin by looking at one-stage instructions. A stage in SERV is 32 consecutive cycles during which the core is active and processes inputs and creates results one bit at a time, starting with the LSB.

2.3.1 One-stage instructions

Most operations are one-stage operations which finish in 32 cycles + fetch overhead. During a one-stage operation, the RF is read and written simultaneously as well as the PC which is increased by four to point to the next instruction. trap and init signals are low to distinguish from other stages.

2.3.2 Interrupts and `ecall/ebreak`

External timer interrupts and `ecall/ebreak` are also one-stage operations with some notable differences. The new PC is fetched from the `MTVEC` CSR and instead of writing to `rd`, the `MEPC` and `MTVAL` CSR registers are written. All this is handled by `serv_state` raising the trap signal during the instruction's execution.

2.3.3 Two-stage operations

Some operations need to be executed in two stages. In the first stage the operands are read out from the immediate and the `rs1/rs2` registers and potential results are written to `PC` and `rd` in the second stage. Various things happen between the stages depending on the type of operation. SERV has types of four two-stage operations; memory, shift, `slt` and branch operations. In all cases the first stage is distinguished by having the `init` signal raised and only performing reads from the RF.

memory

Loads and stores are memory operations. In the init stage, the data address to access is calculated, checked for alignment and stored in `serv_bufreg`. For stores, the data to write is also shifted into the data register in `serv_mem_if`.

If the address has correct alignment, the `o_dbus_cyc` signal is raised to signal an access on the data bus after the init stage has finished and waits for an incoming `i_dbus_ack`, and incoming data in case of loads. After an incoming ack, `o_dbus_cyc` is lowered and stage 2 begins. For stores, the only remaining work in stage 2 is to update the PC. For loads, the incoming data is shifted into `rd`.

If the calculated address in the init stage was misaligned, SERV will raise a exception. Instead of performing an external bus access it will set `mcause` and raise the trap signal, which causes SERV to store the current PC to `mepc`, store misaligned address to `mtval` and set the new PC from `mtvec` which will enter the exception handler.