Arithmetic logic unit (ALU)

Description

The `hsv_core_alu` module is a two-stage Arithmetic Logic Unit (ALU) designed as part of the execute and memory stage within the `hsv_core` architecture. Its primary function is to perform the arithmetic and logical operations of a RV32I architecture on input data from the pipeline stages and generate results to be used in subsequent stages.

General Desing

The ALU module consists of two stages for computation and a pipelining buffer, as described not available in demo.

- **First Stage - Bitwise Operations and Setup**: This stage (`hsv_core_alu_bitwise_setup`) handles the initial processing of input operands, preparing them for further arithmetic and bitwise operations. Key functions include:
  - **Logical Operations**: Implements AND, OR, XOR, and PASS operations based on control signals (PASS = no logical operation needed).
  - **Shift Operations**: Prepares data for left and right shifts, including sign-extended shifts and zero extensions.
  - **Setup for Arithmetic Operations**: Prepares operands for addition, extending them to accommodate sign bits or applying negation as required.
- **Second Stage - Shift and Add**: This stage (`hsv_core_alu_shift_add`) performs the main arithmetic and shift operations:
  - **Addition**: Implements a 33-bit addition, including handling for signed comparisons using a sign-extended representation.
  - **Shifting**: Handles logical and arithmetic shifts, both left and right, using a unified shift logic.
  - **Comparison Operations**: Evaluates less-than comparisons using a simple arithmetic subtraction approach, adjusting signs when necessary. This is a particular case of addittion.
- **Piping**: The results from the ALU are buffered using an `hs_skid_buffer`, which manages pipeline stalls, flush requests and the correct data flow to the next stages. The buffer is parameterized to match the data width of `commit_data`.

Design Considerations and Features

- **Sequential and Flush Control**: The ALU is clocked using `clk_core` and reset with `rst_core_n`. It includes flush signals (`flush_req` and `flush_ack`) to manage pipeline flushing, allowing for the termination of in-progress operations if necessary, such as when a branch is mispredicted or an exception occurs.

- **Data Interface**: The module uses a ready-valid handshake protocol for data communication, with input (`valid_i`, `ready_i`) and output (`valid_o`, `ready_o`) signals facilitating the exchange of data between the ALU and other stages of the pipeline. The ALU receives operand data through the `alu_data` structure and outputs the results through a `commit_data` structure.
- **Pipelining**: The ALU design is fully pipelined to enhance throughput, allowing different operations to be processed concurrently in separate stages. This means there can be up three instructions at the same time in the ALU (one in bitwise setup, one in shift and add and one buffered at the pipe).
- **Shift and Add Integration**: Every instruction produces both a shifted and an added result. Depending on the operation, one of the outputs is chosen. Only one is ever selected for an instruction.
- **Support for Signed and Unsigned Operations**: The module supports both signed and unsigned arithmetic and logical operations as needed for a compliant RV32I implementation.
- **Modular Design**: The ALU is designed with clear modular boundaries (`bitwise_setup` and `shift_add`), allowing for easy maintenance, testing, and potential future expansions or optimizations.
- **Error Handling**: When illegal instructions are detected by the decode stage, the issue logic routes the instruction through the ALU. This is because ALU results can be easily discarded and never produce errors on their own.

I/O

Input Table

Input Name	Direction	Type	Description
`clk_core`	Input	`logic`	Core clock signal for sequential operations.
`rst_core_n`	Input	`logic`	Active-low reset signal for core operations.
`flush_req`	Input	`logic`	Request signal to flush the ALU operations.
`alu_data`	Input	`alu_data_t`	Input data structure containing ALU operands and control signals.
`valid_i`	Input	`logic`	Valid signal indicating the input data is ready to be processed.
`ready_i`	Input	`logic`	Ready signal from the next stage indicating it can accept data.

Output Table

Output Name	Direction	Type	Description
`flush_ack`	Output	`logic`	Acknowledge signal indicating flush has been processed.
`ready_o`	Output	`logic`	Ready signal indicating the ALU can accept new input data.
`commit_data`	Output	`commit_data_t`	Output data structure containing the results of ALU operations.
`valid_o`	Output	`logic`	Valid signal indicating that the output data is ready to be consumed.

Localparams and Structs

`alu_data_t` Struct Table

This is the data that directly tells the ALU what operation to do.

Field Name	Type	Description
`illegal`	`logic`	Indicates if the instruction is illegal; set to 1 for illegal instructions and 0 for valid ALU instructions.
`negate`	`logic`	Control signal to negate the operand(s).
`flip_signs`	`logic`	Control signal to flip the signs of the operand(s).
`bitwise_select`	`alu_bitwise_t`	Specifies the bitwise operation to be performed by the ALU.
`sign_extend`	`logic`	Control signal to extend the sign of the operands.
`is_immediate`	`logic`	Indicates whether the second operand is an immediate value.
`compare`	`logic`	Control signal to perform a comparison operation.
`out_select`	`alu_out_t`	Selects the output result generated by the ALU .
`pc_relative`	`logic`	Control signal indicating that the operation is relative to the program counter (PC).

Field Name	Type	Description
`common`	`exec_mem_commo n_t`	Contains common execution and memory-related control signals shared across multiple execution units.

`exec_mem_common_t` Struct Table

This is a shared structure present in all execute and memory stage modules. ALU extracts the operands from here.

Field Name	Type	Description
`token`	`insn_token`	Token representing the instruction id.
`pc`	`word`	The program counter (PC) value associated with the instruction.
`pc_increment`	`word`	Next instruction, PC+4
`rs1`	`word`	The value of the first source register (rs1).
`rs2`	`word`	The value of the second source register (rs2).
`immediate`	`word`	The immediate value associated with the instruction, if applicable.

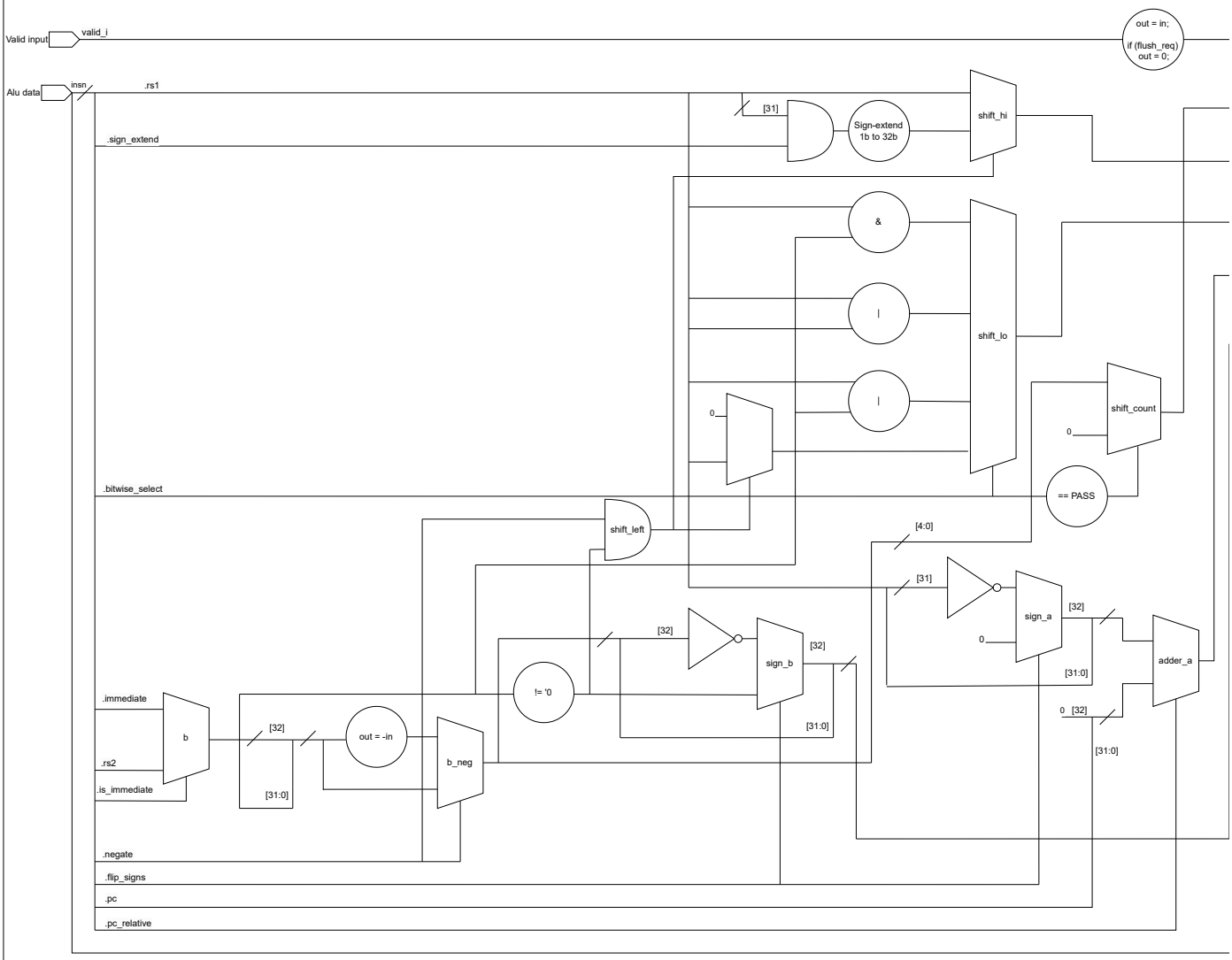
ALU Operation Table

Mnemonic	**`add`**	**`addi`**	**`sub`**	**`and`**	**`andi`**	**`or`**
Operation	`q = a + b`	`q = a + b`	`q = a - b`	`q = a & b`	`q = a & b`	`q = a |`
is_immediate	0	1	0	0	1	0
out_select	`ALU_OUT_A DDER`	`ALU_OUT_A DDER`	`ALU_OUT_A DDER`	`ALU_OUT_S HIFTER`	`ALU_OUT_S HIFTER`	`ALU_OUT_ HIFTER`
bitwise_select	`x`	`x`	`x`	`ALU_BITWI SE_AND`	`ALU_BITWI SE_AND`	`ALU_BITW SE_OR`
sign_extend	`x`	`x`	`x`	`x`	`x`	`x`

Mnemonic	`add`	`addi`	`sub`	`and`	`andi`	`or`
negate	0	0	1	`x`	`x`	`x`
flip_signs	0	0	0	`x`	`x`	`x`
compare	0	0	0	`x`	`x`	`x`
pc_relative	0	0	0	`x`	`x`	`x`
common.rs1	a	a	a	a	a	a
common.rs2	b	`x`	b	b	`x`	b
common.immediate	`x`	b	`x`	`x`	b	`x`

Submodule Diagram

ALU - First Stage
Bitwise operations
and add/shift setup



Legend