# Architecture Design

# Data Path

The Data Path determines how data is transferred inside the CPU during instruction execution (run time).

Even though the data path varies for different instruction types such as Arithmetic Operations, Branches, Jumps, and Memory Operations each instruction phase must start by fetching the corresponding instruction from the memory.

At the end of each instruction the PC shall be updated so that the next phase can begin.

This means either incrementing the PC by 4 or adding an offset to it (branch/jump).

At start of the operation the PC is passed to the PMP & PMA checker (with integrated register), at the same time the control signals read/write/instruction/enable are set to read/instruction/enable. After one clock cycle, the PMP & PMA checker finishes checking the address, if an exception is generated, the trap is taken. Otherwise the data as well as the read/write/enable bits are passed to the AXi4-Lite master.

The AXI4-Lite transfer takes at least two clock cycles. The memOp_finished signal is passed to the control unit as soon as the transfer is finished. The Instruction Register is written by the AXI master if the store_systemData signal is high and the Control Unit is in the Fetch Instruction cycle. The LoadBufferRegister is written at the end of each read cycle, independent if it is used or not.

Instruction decoding and execution will follow on the next clock cycle.

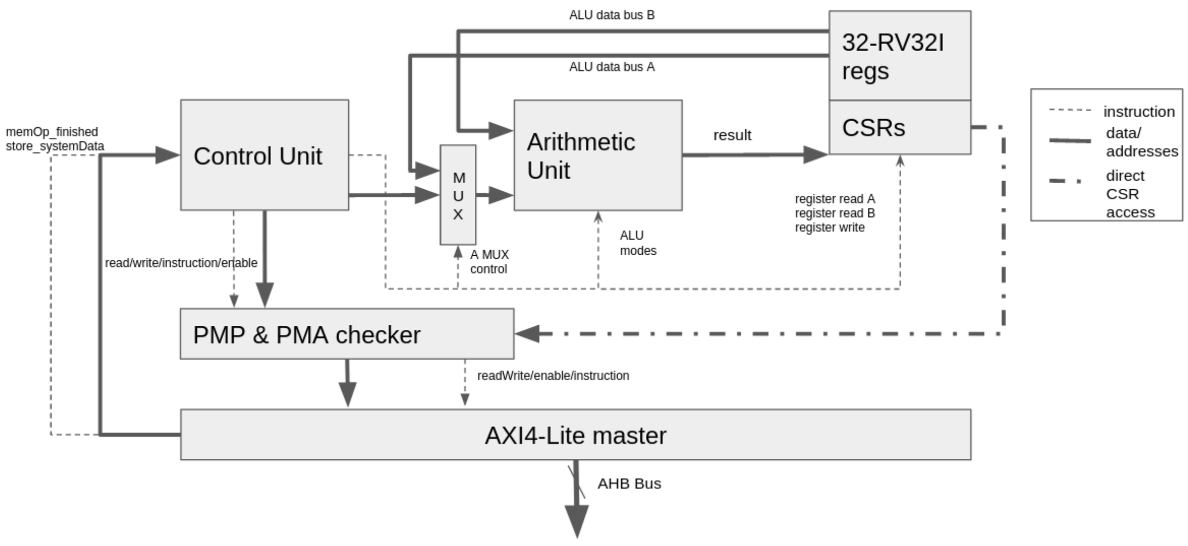
The updated PC value (assumed) is calculated inside a special unit during the second clock cycle, but not yet stored in the PC.

The design incorporates mostly registers that are read on the rising edge and written on the falling. Exceptions are only the Instruction Register inside the Control Unit as well as the LoadBufferRegister (write on rising edge).

| Clock Cylce | Action |
|---|---|
| 1 | rising:<br>- pass PC and read/instruction/enable to PMP & PMA checker (control unit)<br>falling:<br>- load address in PMP & PMA checker output register |
| 2-x (assumed: 3) | 2: rising:<br>- start AXI transfer<br>- calculate PC+4 (not stored yet)<br>3: rising: wait |
| 4 | rising: if store_systemData is high, the data will be read into the IR (Instruction register) and decoding can begin. Execution may begin as soon as the memOp_finished signal is applied |

## Arithmetic Operations

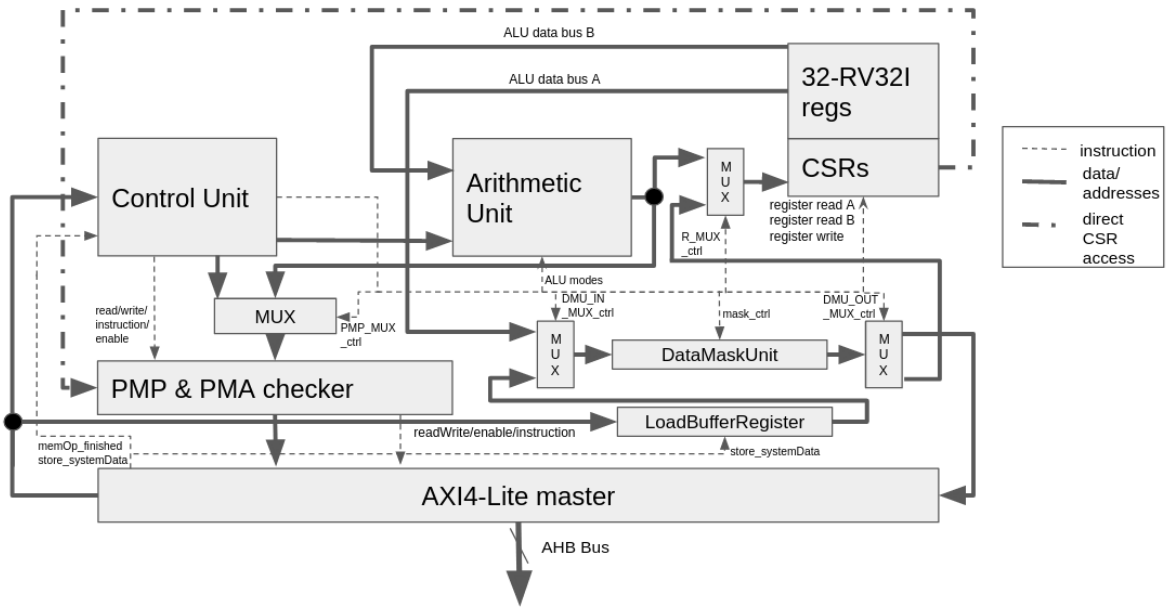
Arithmetic Operation



| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | rising:<br>- load register B onto ALU data bus B and either the immediate or register A on the data bus A (this is decided by multiplexer and the corresponding control signal)<br>- ALU calculates the result (if the instruction is to load an immediate, the value on data bus B is irrelevant)<br>falling:<br>- load data into specified register by control signal register write<br>- deassert execute signals<br>- load PC with updated value |

## Memory Operations

Memory Operations



Store:

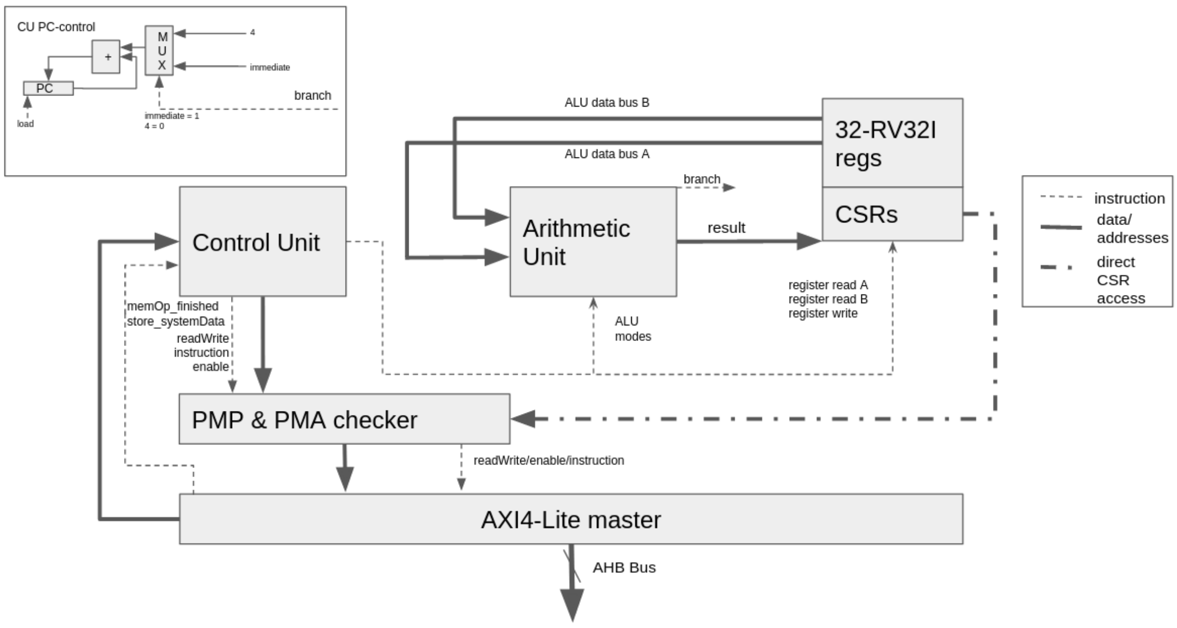
| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | rising:<br>- load register B onto ALU data bus B as well as the offset(immediate) to calculate the address<br>- ALU calculates the address<br>- perform PMP & PMA checks<br>- load DataMaskUnit via data bus A and mask data<br>falling:<br>- if address passes PMP & PMA checker, load it in buffer register, else assert exception signal and load it in buffer register<br>- provide AHB master with data to store |
| 6 | rising:<br>- start AXI4-Lite transfer |
| 7-x | AXI4-Lite transfer |
| x+1 | falling:<br>- deassert execute signals<br>- update PC |

Load:

| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | rising:<br>- load register B onto ALU data bus B as well as the offset(immediate) to calculate the address<br>- ALU calculates the address<br>- perform PMP & PMA checks<br>falling:<br>- if address passes PMP & PMA checker, load it in buffer register, else assert exception signal and load it in buffer register |
| 6 | rising:<br>- start AXI4-Lite transfer |
| 7-x | AXI4-Lite transfer |
| x+1 | rising:<br>- load data in load buffer register<br>- Mask Data<br>falling:<br>- store masked data in register<br>- deassert execute signals<br>- update PC |

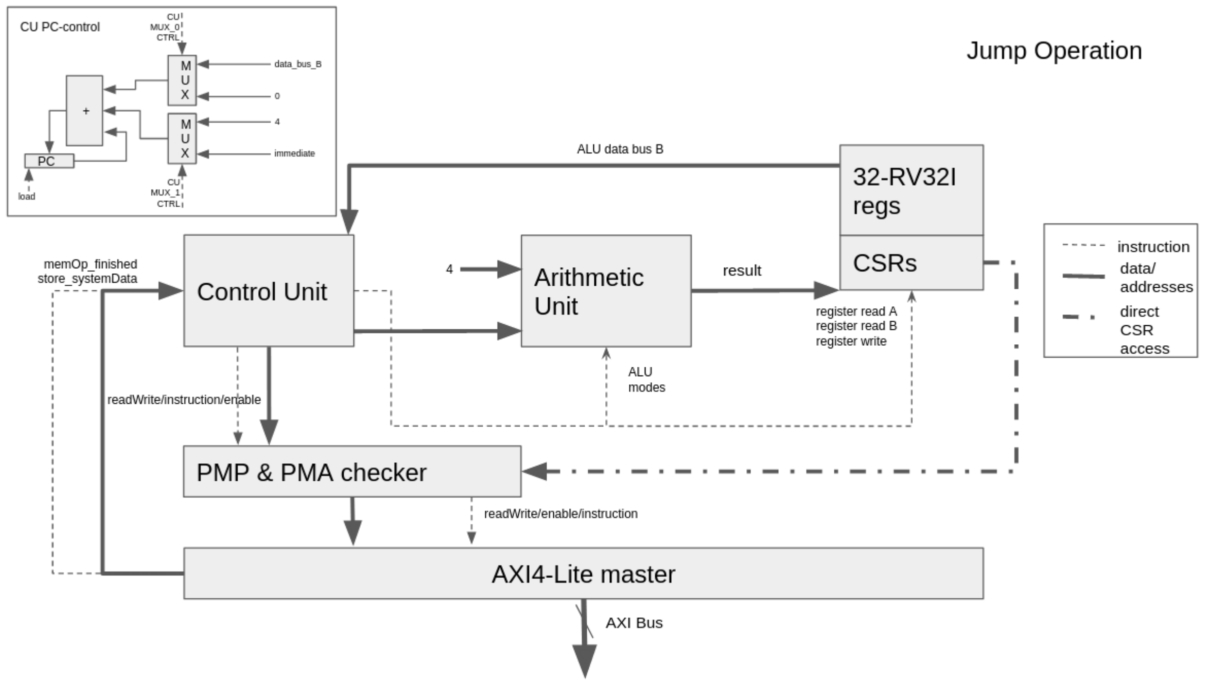
## Conditional Branches

Jump Operation



| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | rising:<br>- Load Value of src1 & src2 register over data bus A & B<br>- ALU performs compare<br>- branch signal is either set or not<br>- branch signal either applies 4 or the immediat to the operand input of the adder inside CU PC-control<br>- add chosen value to the PC (CU<br>falling:<br>- deassert execute signals<br>- update PC |

## Jump Operations

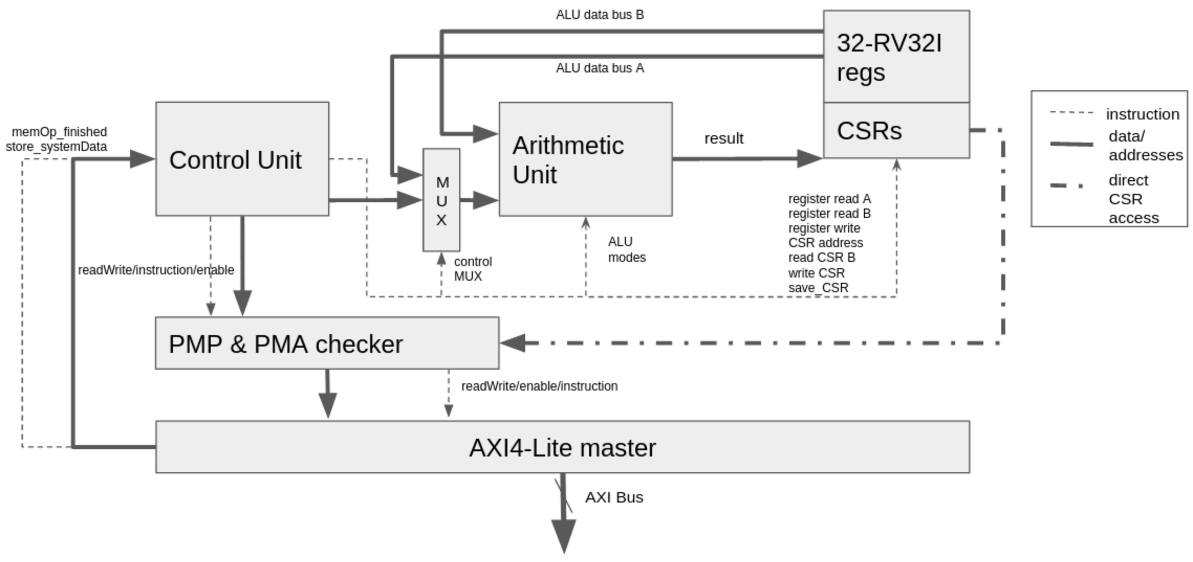


JAL/JALR:

| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | rising:<br>- PC must be available at ALU input A<br>- pass register B to MUX1 in CU PC-Control<br>- add immediate, PC and register B (optional)<br>- increment PC +4 using the ALU<br>falling:<br>- store PC at specified register<br>- deassert execute signals<br>- update PC |

# Zicrs Instructions

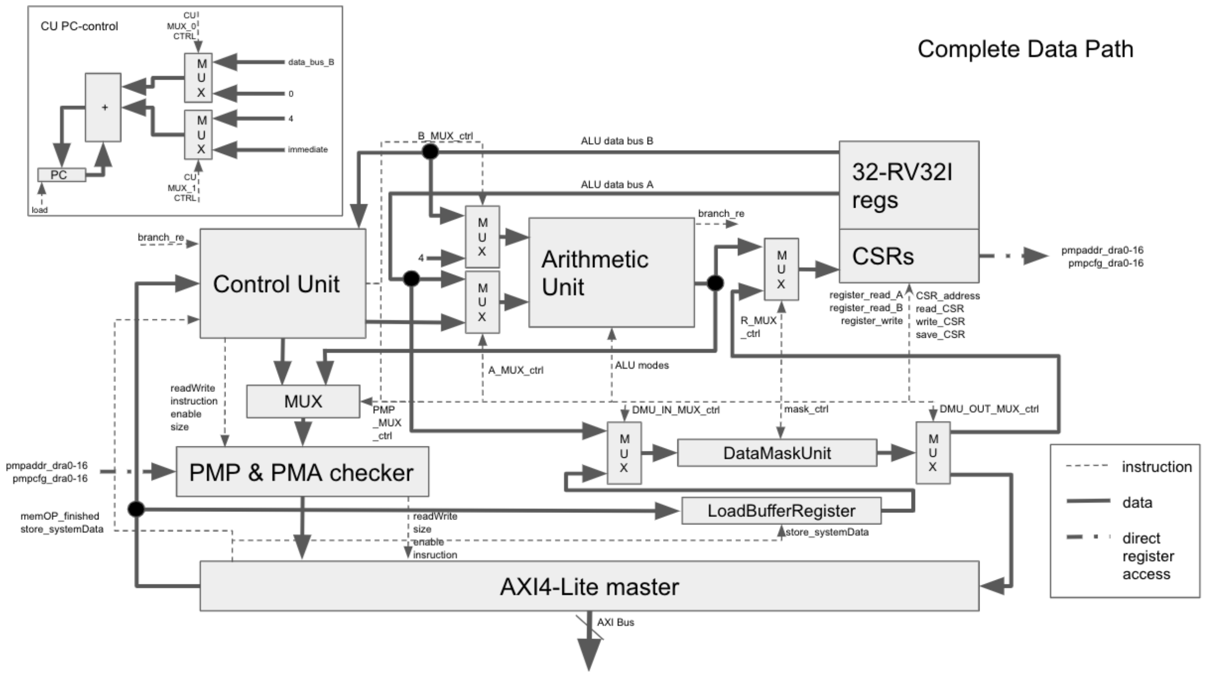
Zicsr Operation



| Clock Cycle | Action |
|---|---|
| 4 | control signals must be stable at the end of clock cycle 4 & immediates generated<br>falling:<br>- read control signals into ExecuteBuffer |
| 5 | CSRRW(I):<br>rising:<br>- load value from rs1 (pass through alu) (or Immediate)<br>falling:<br>- store value from ALU in CSR<br>CSRRS(I) & CSRRC(I):<br>if rs1 != x0 or immediate != 0<br>rising:<br>- load data from CSR onto ALU data bus A<br>- load data from rs1 onto ALU data bus B (or immediate)<br>- mask data<br>falling:<br>- if rd!=x0 & instruction != CSRRW(I) (save_CSR = 1): Store Value from CSR in rd (inside register file)<br>- deassert execute signals<br>- update PC<br>- store in CSR |

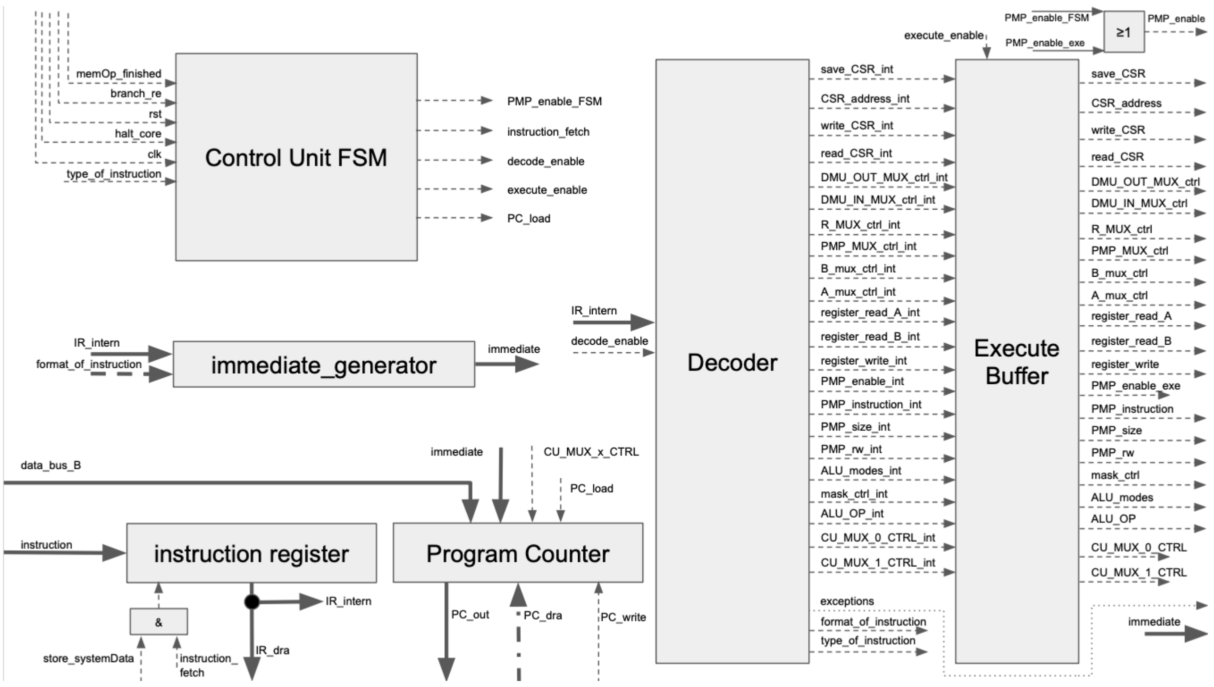
## Complete Data Path

The data path shown below is the complete data path of the RV32I IP, without the exception control unit and memory mapped CSRs. Every Instruction specified inside the RV32I ISA as well as the “Zicsr” extension should be executable, using this Data Path..



# Control Unit

## Control Unit Architecture



The Control Unit's main components are the FSM, Decoder, immediate_generator as well as various Registers.

The FSM is in charge of controlling the different phases (instruction fetch, decode and execute) in the last cycle of the execute phase, the PC_load signal must be applied on the rising edge of the clock in order to update the PC on the falling edge.

The Decoder is used to convert the instruction to control signals and provide the fsm and immediate generator with information about the instruction format and type. It is not clocked and will work as soon as the decode_enable signal is applied to it.

The execute Buffer is used to hold the Decoder values and apply them to the output. If the execute_enable phase is applied to the Execute Buffer, it will read the decode values on the next falling edge. Else, the signals will be reset to dedicated values. These include PMP_MUX_ctrl to pass the PC to the PMP, PMP_size to represent 32, PMP_instruction to one and PMP_rw to represent read.

The program counter includes (as described inside the Data Path chapter) additional units to increment the program counter. These are controlled by the CU_MUX_x_CTRL signals (x=0,1).

The instruction register holds the instruction and is written by the AXI_master on a rising clock edge, if the store_systemData signal as well as the instruction_fetch is applied.

To generate immediates the immediate generator is used, it calculates the immediates dependent on format of instruction and instruction value. This is a non clocked module.

## Control Unit entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| clk | std_logic | register_read_A | std_logic_vector(31 downto 0) |
| pc_write | std_logic_vector | register_read_B | std_logic_vector(31 downto 0) |
| instruction | std_logic_vector(31 downto 0) | immediate | std_logic_vector(31 downto 0) |
| branch_re | std_logic | register_write | std_logic_vector(31 downto 0) |
| rst | std_logic | ALU_OP | std_logic_vector(5 downto 0) |
| memOp_finished | std_logic | ALU_modes | std_logic_vecotor (? downto 0) |
| halt_core | std_logic | instr_finished | std_logic |
| data_bus_B | std_logic_vector(31 downto 0) | PMP_MUX_ctrl | std_logic |
| pc_dra | std_logic_vector(31 downto 0) | A_MUX_ctrl | std_logic |
| store_systemData | std_logic | B_MUX_ctrl | std_logic |
| | | DMU_IN_MUX_ctrl | std_logic |
| | | DMU_OUT_MUX_ctrl | std_logic |
| | | R_MUX_ctrl | std_logic |
| | | mask_ctrl | std_logic_vector(2 downto 0) |
| | | iie_CU | std_logic |
| | | ece_CU | std_logic |
| | | be_CU | std_logic |
| | | CSR_address | std_logic_vector(11 downto 0) |

| | | | |
|---|---|---|---|
| | | write_CSR | std_logic |
| | | read_CSR | std_logic |
| | | save_CSR | std_logic |
| | | PMP_rw | std_logic |
| | | PMP_instruction | std_logic |
| | | PMP_enable | std_logic |
| | | PMP_size | std_logic(1 downto 0) |
| | | PC_out | std_logic_vector(31 downto 0) |
| | | IR_dra | std_logic_vector(31 downto 0) |

The control unit itself manages the current instructions, decodes them and sends control signals to all the other units in the CPU. The control unit manages the Program Counter except for the case that a corruption occurs or an exception is thrown. The control unit steadily listens to the inputs of the exception control which will stop the core on an unwanted event.

**Decoding**

To decode the instructions, a decode unit has to be implemented. This decode unit gets the instruction stream which the control unit reads from the memory. The task of the decode unit is to decode this instruction stream and create all necessary control signals for the ALU, the Register Unit and the Exception Control.

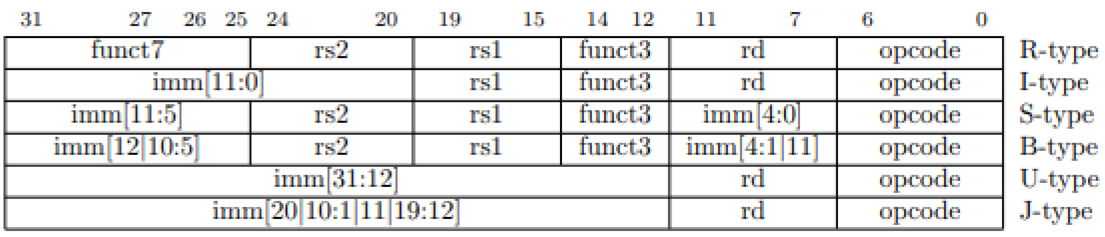
The different Instruction Formats can be found in the “System Design” File. To decode every instruction the 6 instruction formats have to be implemented and the corresponding memory and ALU operations have to be created.

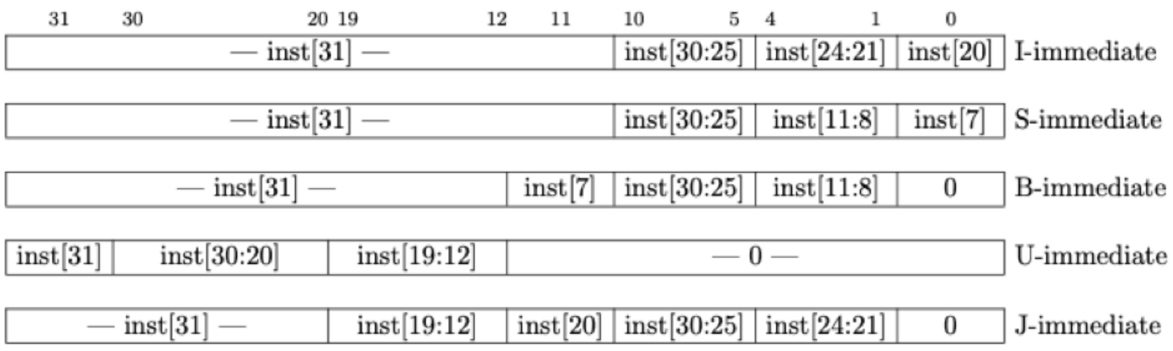
| Instruction Type | Overview | Example |
|---|---|---|
| R-Type | OpCode xD, xA, xB | ADD |
| I-Type | OpCode xD, xA | LW, ADDI |
| S-Type | OpCode xA, xB | SW |
| B-Type | OpCode xA | BREQ |
| U-Type | OpCode xA, Immediate | LUI |
| J-Type | OpCode Immediate | JAL |

The identification of the correct instruction is performed inside a separate entity which acts as a finite state-machine which detects the format and extracts the correct information from the instruction stream.

**Immediate Instructions**

Another separate entity is handling the immediate generation. Depending on the instruction type, the immediate is generated differently. Immediates are always sign extended!


31 27 26 25 24 20 19 15 14 12 11 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12|10:5] | rs2 | rs1 | funct3 | imm[4:1|11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20|10:1|11|19:12] | rd | opcode | J-type


31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

| Instruction | Description |
|---|---|
| ADDI | add 12-bit immediate (sign extended) to given register |
| SLTI | if input register is less than 12-bit immediate (sign extended) set destination register to value 1 |
| XLEN | |
| ANDI,ORI, XORI | perform bitwise AND, OR, XOR operation with input register and 12-bit immediate (sign extended) |
| SLLI, SLRI | logical shift |
| LUI | Immediate to top 20 bits of destination -> fill low 12 bits with zeros |
| AUIPC | form 32-bit offset from 20-bit immediate -> fill low 12 bits with zeros -> add this offset to AUIPC instruction address -> place result to destination |

If the current instruction includes an immediate, the mux_ctrl output has to be changed for the ALU to receive the immediate instead of a register data.

**PC control**

To control the program counter a separate entity is created which manages the PC depending on branch signals, exceptions or normal operations.

## ALU

The ALU executes the arithmetical operations required for an instruction. To perform the operation the entity ALU needs following inputs:

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| reg_A, reg_B | std_logic_vector(31 downto 0) | ALU_RESULT | std_logic_vector(31 downto 0) |
| ALU_op | std_logic_vector(3 downto 0) | branch_re | std_logic |
| ALU_branch | std_logic | | |
| clk | std_logic | | |
| immediate | std_logic_vector(31 downto 0) | | |

The ALU unit receives the inputs from the control unit which decodes the instructions.
The REG_1 and REG_2 inputs consist of the registers required for the operations.
The operation to be performed with the registers is given by ALU_OP input.

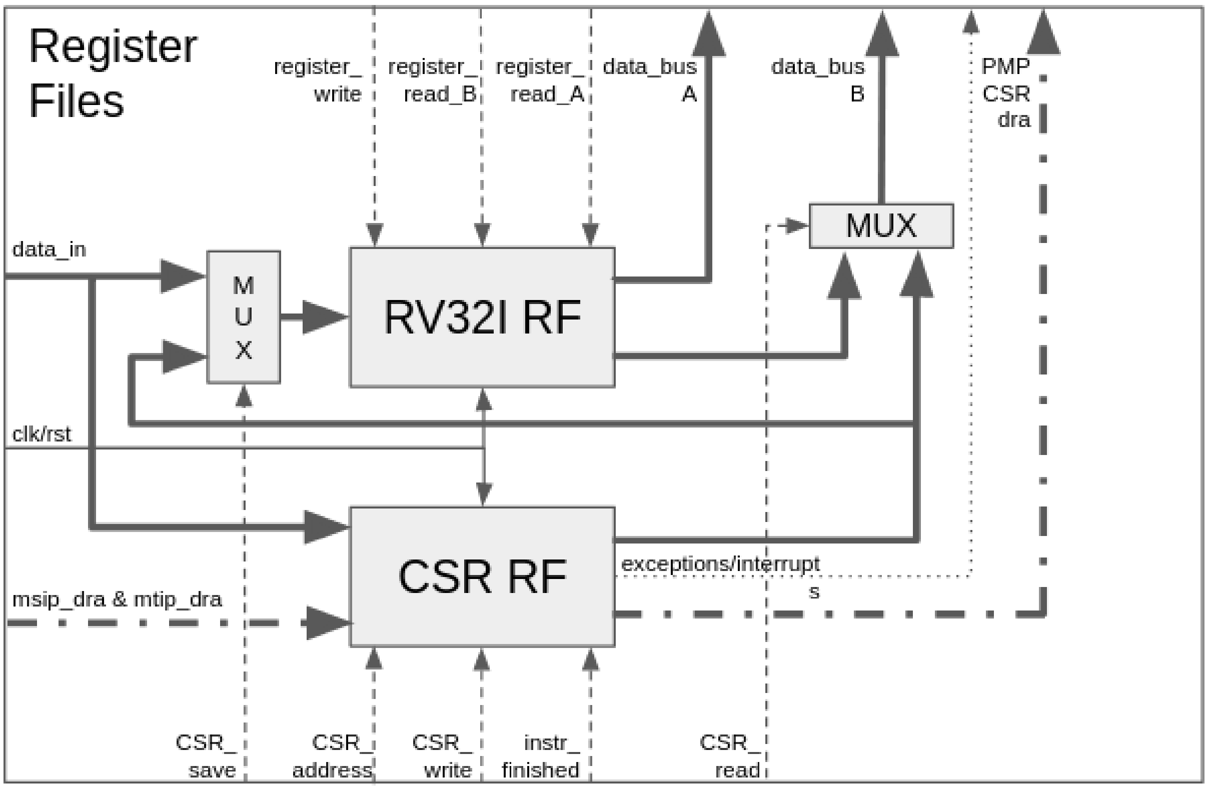
# Register File

## Register Files Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| data_in | std_logic_vector(31 downto 0) | data_bus_A | std_logic_vector(31 downto 0) |
| clk | std_logic | data_bus_B | std_logic_vector(31 downto 0) |
| reset | std_logic | iie_CSR | std_logic |
| CSR_address | std_logic_vector(11 downto 0) | si_CSR | std_logic |
| CSR_read | std_logic | ti_CSR | std_logic |
| CSR_write | std_logic | pmpaddr0-15 | 15*std_logic_vector(31 downto 0) |
| CSR_save | std_logic | pmpcfg0-15 | 15*std_logic_vector(7 downto 0) |
| register_read_A | std_logic(31 downto 0) | | |
| register_read_B | std_logic(31 downto 0) | | |
| register_write | std_logic(31 downto 0) | | |
| msip_dra | std_logic | | |
| mtip_dra | std_logic | | |
| instr_finished | std_logic | | |

## Register File Architecture

The following diagram describes the Architecture of the Register Files:



The two main components are the RV32I RF and CSR RF, these are discussed in more detail inside the corresponding chapters.

In order to write data to a register, the value needs to be put on the data_in bus and the corresponding control signals need to be configured. Data can be read either via the data_bus A or B, the CSR registers can only be outputted to data_bus B via a MUX controlled by the CSR_read bit. If the CSR_read bit is set, the CSR specified by CSR_address will be visible on data_bus B.

The CSR RF independently checks the mie and mip registers for interrupts and signals them to the Exception Control whenever a pending bit and the corresponding enable bit as well as the global enable bit are set.

While the CSR RF is accessed by 12 Bit addresses, RV32I registers are accessed without addressing. Each register therefore requires 3 Signals resulting in a total of 96 signals for read and write.

The RFs are clocked with the system clock and reset on system reset. As per usual, data is stored on the falling clock and displayed on the rising clock.

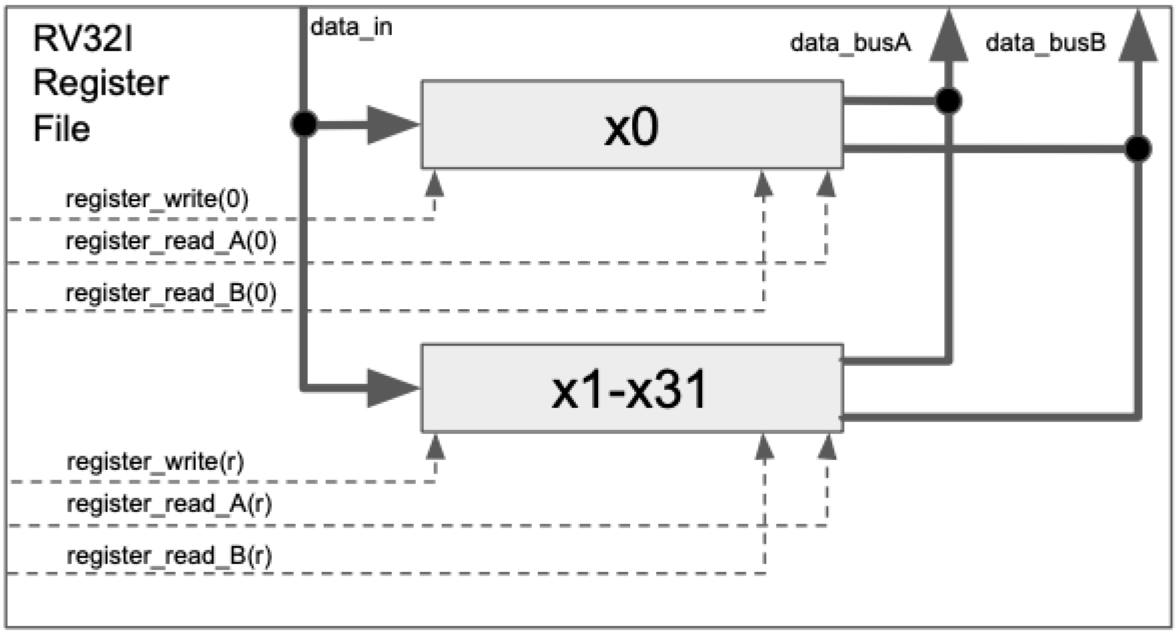
If the CSR_save signal is applied (set to high), the CSR Register File output is redirected to the MUX, therefore CSR register may be stored directly to the RV32I Register File.

## RV32I Register File

### RV32I Register File Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| data_in | std_logic_vector(31 downto 0) | data_busA | std_logic_vector(31 downto 0) |
| register_write | std_logic_vector(31 downto 0) | data_busB | std_logic_vector(31 downto 0) |
| register_read_A | std_logic_vector(31 downto 0) | | |
| register_read_B | std_logic_vector(31 downto 0) | | |

### RV32I Register File Architecture



Register x0 is hardwired to zero, writes are allowed but will have no effect to it.
x1-x31 can be read and written.
Writes are executed on a falling edge if the corresponding register_write signal is highe.
Reads are performed on a rising edge if the corresponding register_read_A/B is high.

### RV32I Register File Reset

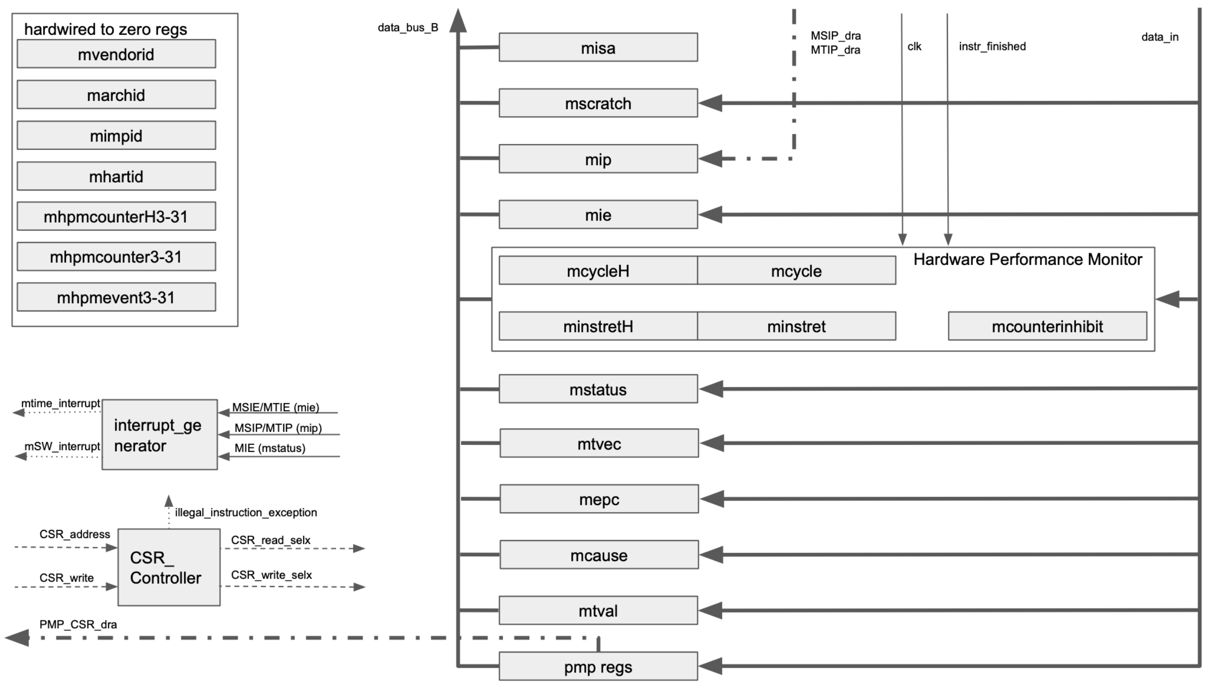
- reset all registers to 0x00000000

## CSRs

### CSR Register Files Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| data_in | std_logic_vector(31 downto 0) | data_bus_B | std_logic_vector(31 downto 0) |
| MSIP_dra | std_logic | mtime_interrupt | std_logic |
| MTIP_dra | std_logic | mSW_interrupt | std_logic |
| CSR_address | std_logic_vector(11 downto 0) | PMP_CSR_dra | 20*std_logic_vector(31 downto 0) |
| CSR_write | std_logic | illegal_instruction_exception | std_logic |
| instr_finished | std_logic | | |
| clk | std_logic | | |

### CSR Register Files Architecture



The CSR Registers File is mainly composed of the different CSR Registers, some are writable and some writable and readable. The CSR_Controller checks if a CSR access is allowed and produces access signals for each register from the given address. If an address is not writable but yet a write is requested, the CSR_Controller raises an illegal_instruction_exception.

The interrupt generator checks if an interrupt is pending, enabled and if interrupts are globally enabled. In that case the corresponding interrupt is raised. The interrupts remain pending, as long as the corresponding direct register access signals it.

Some registers are hardwired to zero. These do not need to be implemented, much rather the CSR_controller handles access to these as follows:

- read: return zero
- write: ignore / illegal_instruction_exception, if writes are not allowed.

The Hardware Performance Monitor registers are incremented at special events (clock cycle and instruction finish) therefore two 64-Bit increment units have to be implemented. The increments are only performed, if the corresponding bit inside the mcounterinhibit register is set.

The pmp regs are composed of the 16 pmpaddr registers and four pmpcfg registers. They can be read and written. In addition direct register read access is provided.

## CSR Register Files Reset

The CSR Registers will be reset to the following values:

| Register | Value |
|---|---|
| misa | 0x00000100 |
| mscratch | 0x00000000 |
| mip | 0x00000000 |
| mie | 0x00000000 |
| mcycleH | 0x00000000 |
| mcycle | 0x00000000 |
| minstretH | 0x00000000 |
| minstret | 0x00000000 |
| mcounterinhibit | 0x00000005 |
| mstatus | 0x00001800 |
| mtvec | 0x00000000 |
| mepc | 0x00000000 |
| mcause | 0x00000000 |
| mtval | 0x00000000 |
| pmpaddr0-15 | 0x00000000 |
| pmpcfg0-3 | 0x00000000 |

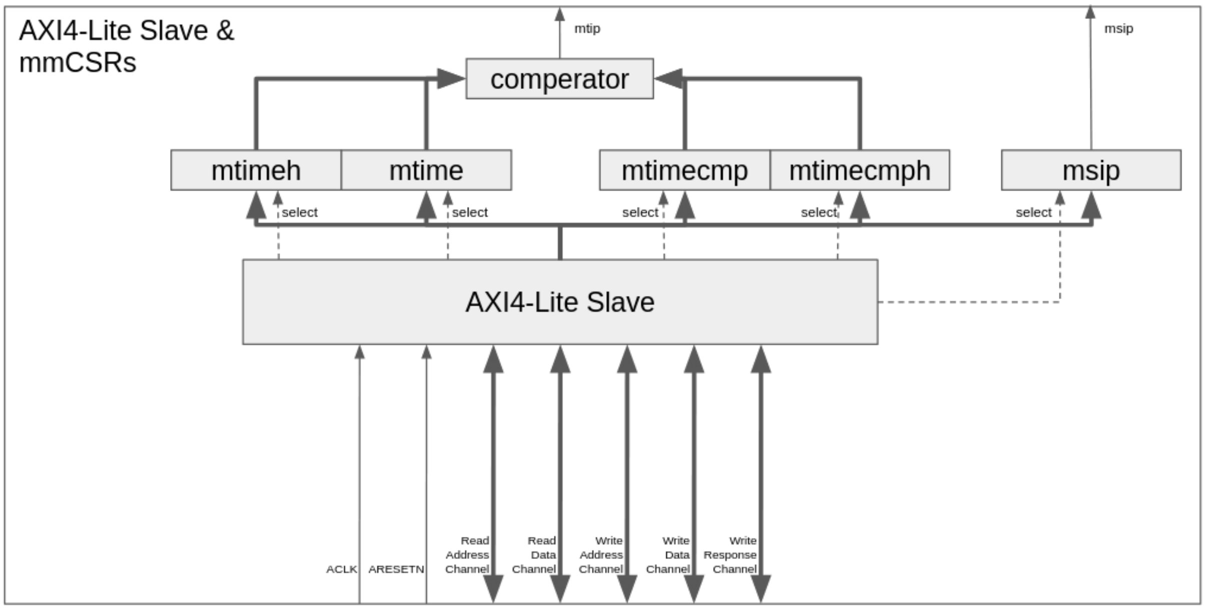
## AXI4-Lite Slave & memory mapped CSRs

### AXI4-Lite Slave & memory mapped CSRs Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| ACLK | std_logic | RA Channel | - |
| ARESETN | std_logic | RD Channel | - |
| RA Channel | - | WA Channel | - |
| RD Channel | - | WD Channel | - |
| WA Channel | - | WR Channel | - |
| WD Channel | - | mtip | std_logic |
| WR Channel | - | msip | std_logic |
| | | | |
| | | | |
| | | | |

## AXI4-Lite Slave & memory mapped CSRs Architecture

As AXI4-Lite slave an IP core may be used. In fact the Vivado AXI Interface generator may be used to generate the AXI4-Lite Slave as well as the registers, this IP could then be modified with the comperator and the msip and mtip output



The registers are addressed corresponding to the CLINT module by sifive since this is a common known address map.

| address | register |
|---|---|
| 0x0200_0000 | msip |
| 0x0200_4000 | mtimecmp |
| 0x0200_4004 | mtimecmph |
| 0x0200_BFF8 | mtime |
| 0x0200_BFFB | mtimeh |

The AHB Slave allowes reads and writes to the 32 bit registers mtimeh, mtime, mtimecmp and mtimecmph and the one byte msip register.

The comparator is used to trigger a timer interrupt. It compares mtime and mtimecmp. If the value of mtime is equal or greater than the one in mtimecmp, the mtip signal is asserted. The interrupt remains posted, until it is cleared by writing to the mtimecmp register.

The msip register is a one byte register, the remaining 31 bits are hardwired to zero. Its reset value is zero. If it is set to one, the msip bit in mip is set and therefore, a softare interrupts may be risen.

## Memory Mapped CSRs Reset

| Register | Value |
|---|---|
| msip | ‘0’ |
| mtimeh | 0x00000000 |
| mtime | 0x00000000 |
| mtimecmph | 0xFFFFFFF |
| mtimecmp | 0xFFFFFFF |

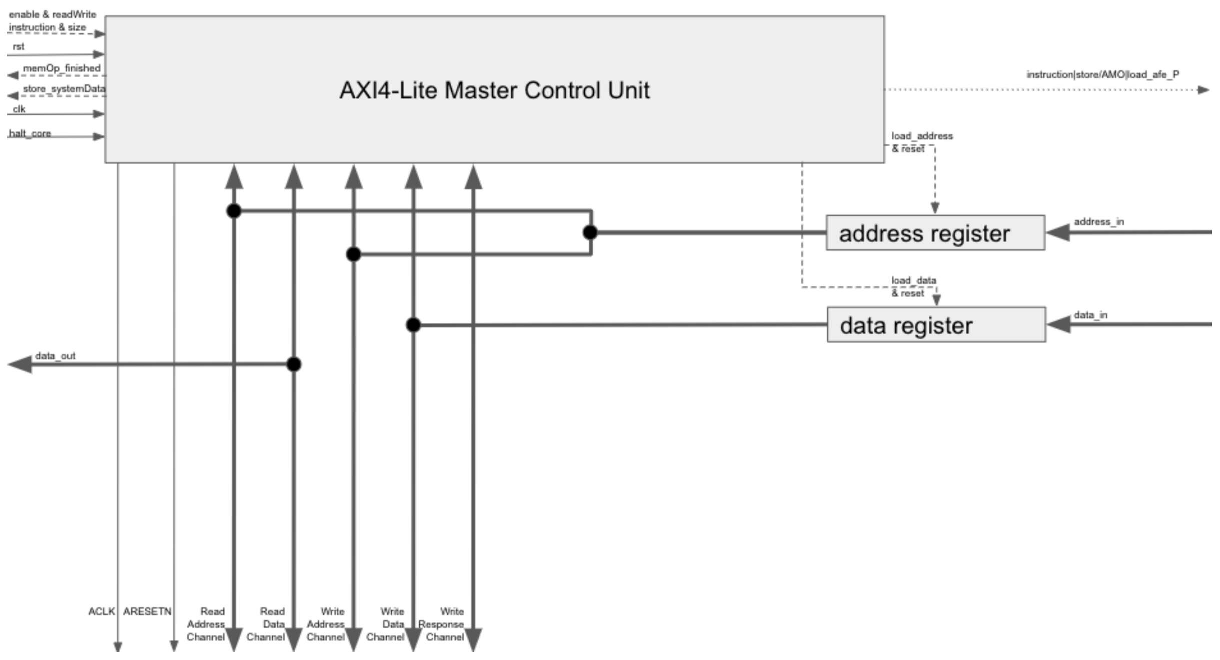
## AXI4-Lite Master

The AHB master implements the AMBA AXI4-Lite protocol.

### AXI4-Lite Master Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| data_in | std_logic_vector(31 downto 0) | load_afe_P | std_logic |
| address_in | std_logic_vector(31 downto 0) | storeAMO_afe_P | std_logic |
| reset_global | std_logic | instruction_afe_P | std_logic |
| clk | std_logic | data_out | std_logic_vector(31 downto 0) |
| readWrite | std_logic | store_systemData | std_logic |
| instruction | std_logic | memOp_finished | std_logic |
| enable | std_logic | RA Channel | - |
| halt_core | std_logic | RD Channel | - |
| size | std_logic(1 downto 0) | | |
| RA Channel | - | WA Channel | - |
| RD Channel | - | WD Channel | - |
| WA Channel | - | WR Channel | - |
| WD Channel | - | | |
| WR Channel | - | | |

## AXI4-Lite Master Architecture



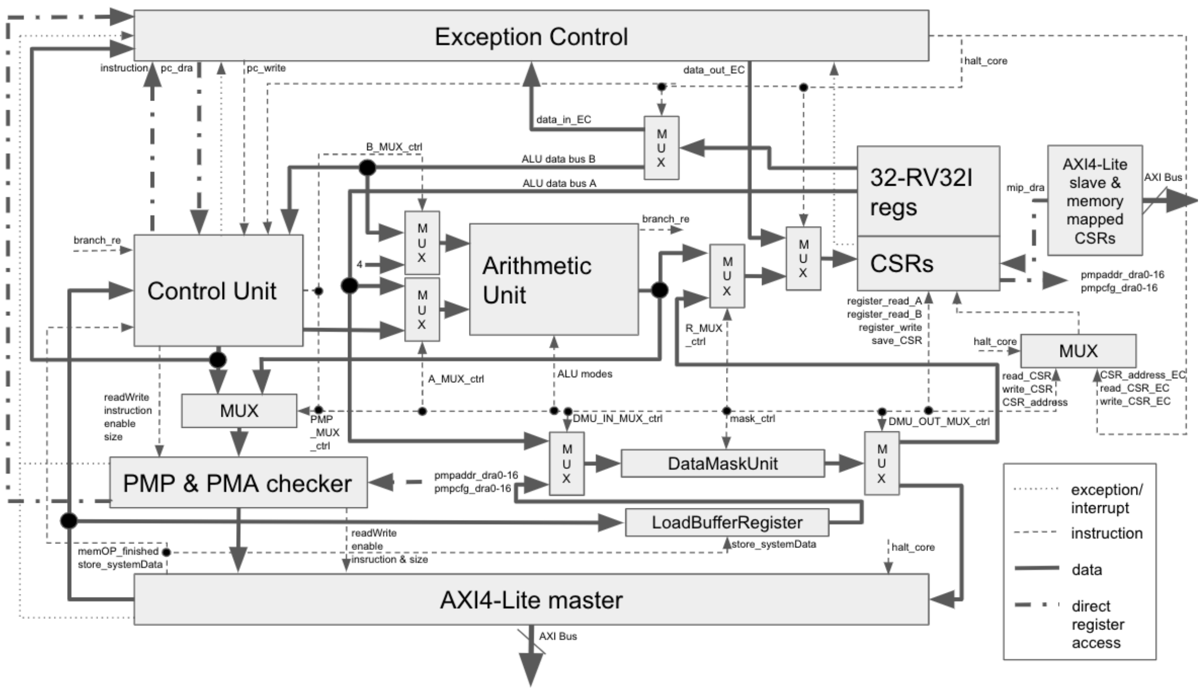
The AXI master includes two registers, the data register and the address register. It is loaded on the rising edge of the system clock if the corresponding load signal is applied, and read on the rising edge of the clk. Both register outputs are constantly applied to the corresponding axi signals (AWADDR, ARADDR and WDATA)

The AXI4-Lite Master Control Unit is in charge of controling the AXI Transfer. It controlls the Ready and Valid signals as well as the register reads and writes. It is clocked with clk. The rst signal will reset it. The Master also provides the clock and reset for every slave connected to the AXI interconnect. To start a transfer, the enable signal must be high on a rising edge of the clock, in that case the address and data register are loaded on the next falling edge of system clock. If data is ready to be read from the system bus, it is routed to an output of the Master and the store_systemSystemData is set to high, in order to ensure a correct read, this process shall not be clocked. The surrounding system must then process these signals to store the data in the correct register.
At the end of an data transfer, the memOp_finished signal is set to high and remains high until a new transfer is initiated.

If an error occurred during an access, the instruction_afe_P, storeAMO_afe_P or load_afe_P are risen. And remain high until the halt_core signal is applied to the AXI4-Lite Master Control Unit.

## Exception Control

The Picture below shows a detailed view of the architecture of the whole IP, including the Exception Control as well as the AHB slave & memory mapped CSRs:



Exceptions can be caused by the following modules:

- Control Unit
- CSR register file
- PMP & PMA checker

The Control Unit may cause the following exceptions:

- illegal instruction exception
- breakpoint exception
- environment-call-from-M-mode exception

The PMP & PMA checker may cause the following exceptions:

- load access fault exception
- store/AMO access fault exception
- instruction access fault exception
- load address misaligned exception
- store/AMO address misaligned exception
- instruction address misaligned exception

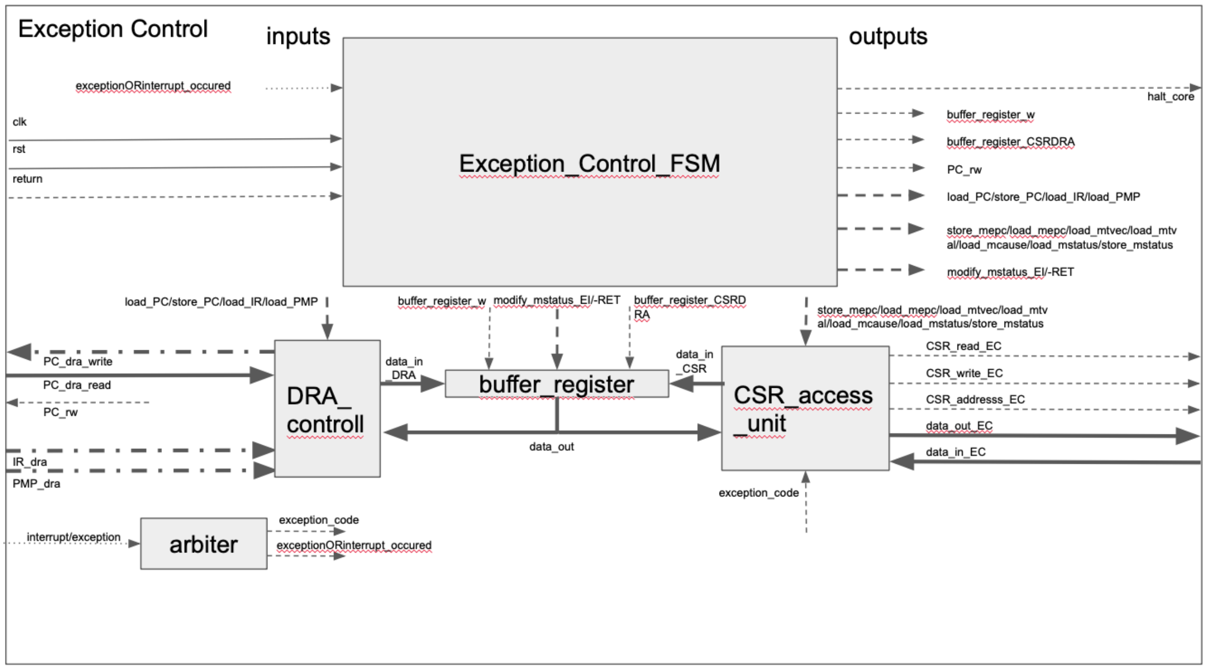
The CSR register file may cause the following exceptions/interrupts:

- timer interrupt
- software interrupt
- illegal instruction exception

## Exception Control Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| load_afe_P | std_logic | CSR_read_EC | std_logic |
| storeAMO_afe_P | std_logic | CSR_write_EC | std_logic |
| instruction_afe_P | std_logic | CSR_address_EC | std_logic_vector(11 downto 0) |
| load_ame_P | std_logic | data_out_EC | std_logic_vector(31 downto 0) |
| storeAMO_ame_P | std_logic | pc_write | std_logic |
| instruction_ame_P | std_logic | halt_core | std_logic |
| load_afe_AXI | std_logic | PC_dra_write | std_logic_vector(31 downto 0) |
| storeAMO_afe_AXI | std_logic | | |
| instruction_afe_AXI | std_logic | | |
| iie_CU | std_logic | | |
| ece_CU | std_logic | | |
| be_CU | std_logic | | |
| iie_CSR | std_logic | | |
| si_CSR | std_logic | | |
| ti_CSR | std_logic | | |
| data_in_EC | std_logic_vector(31 downto 0) | | |
| IR_dra | std_logic_vector(31 downto 0) | | |
| PMP_dra | std_logic_vector(31 downto 0) | | |
| clk | std_logic | | |
| reset | std_logic | | |
| return | std_logic | | |
| pc_dra | std_logic_vector(31 downto 0) | | |

## Exception Control Architecture



The Exception Control module consists of a FSM, an arbiter to decide if an interrupt or exception is taking place and which one shall be handled if multiple are raised at a time. To modify registers a DRA_controll as well as a CSR_access_unit and a buffer register are added to the design.

The arbiter receives the different exception and interrupt signals, it's purpose is to decide which interrupt will be executed (Table 3.7 privileged spec) and generate the according exception code as well as the signal for the FSM to start the routine.

The DRA_controll module is used to load the Instruction Register, PMP address register and Program Counter. A load to the PC may also be performed, in order to do so the PC_rw signals must be asserted one clock cycle earlier by the FSM.

The buffer_register is implemented to allow data shares between the DRA_controll and CSR_access_unit. It implements another functionality that allows to set either the MIE register to 1 (return) or the MPIE register to 0 (entry) and switch the two bits (MIE and MPIE) on the data_out line. This feature is used during the phase of modifying the mstatus register and allows the modify to happen in at least to clock cycles.

The CSR_access_unit is used to perform register accesses to the CSR Register File. During Exception entry or return the data bus B must be connected to the data_in_EC bus and the data_in bus to the data_out_EC bus. This is done by implementing a multiplexer at the corresponding buses controlled by the halt_core signal.

If an exception entry is performed if the exceptionORinterrupt_occured signal on the FSM is high, if the return signal is high it indicates that an exit shall be performed, however if both signals are high (should not happen in normal operation) the entry will be performed.

## Exception Control Reset

- reset buffer_register to 0x00000000
- reset FSM to wait_for_exceptionORinterrupt state

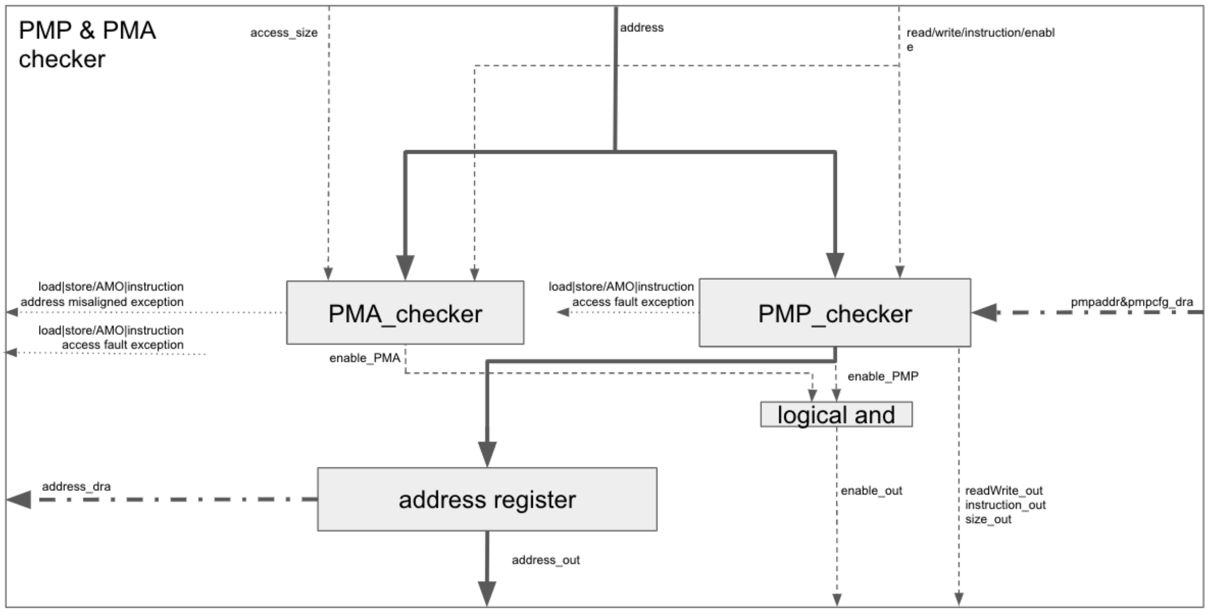
# PMP & PMA Checker

## PMP & PMA Checker Entity

| Input | Data Type | Output | Data Type |
|---|---|---|---|
| acces_size | std_logic_vector (1 downto 0) | load_afe_P | std_logic |
| enable | std_logic | storeAMO_afe_P | std_logic |
| readWrite | std_logic | instruction_afe_P | std_logic |
| instruction | std_logic | load_ame_P | std_logic |
| clk | std_logic | storeAMO_ame_P | std_logic |
| rst | std_logic | instruction_ame_P | std_logic |
| address | std_logic_vector(31 downto 0) | address_dra | std_logic_vector(31 downto 0) |
| pmpaddr_dra | 16*std_logic_vector(31 downto 0) | address_out | std_logic_vector(31 downto 0) |
| pmpcfg_dra | 16*std_logic_vector(7downto 0) | enable_out | std_logic |
| | | readWrite_out | std_logic |
| | | instruction_out | std_logic |
| | | size_out | std_logic_vector (1 downto 0) |

## PMP & PMA Checker Architecture

The PMP & PMA checker is described by the following circuit:



The PMA_checker is used to verify the protection of the physical memory attributes. For this version, the only physical memory attribute is whether or not an address is aligned. In order to determine this the size of the acces is needed. A one byte access always passes, whereas for a half-word access (two-bytes) the LSB has to be zero. In order to successfully read a word (4-bytes) the two LSBs of the address need to be zero.
To raise the right exception, the PMA_checker needs to know if the access is a read, write or an instruction fetch.

In order to specify, if reads, writes or instruction fetches are possible to a certain address, it needs to be specified inside the pmpaddr and corresponding pmpcfg CSRs. The PMP_checker has direct access to those and enforces the rules. If an address is not specified every access is allowed.
Address regions are only used if the lock bit in pmpcfg is set, this can only be reseted by a system reset.

The logical and is used, in order to ensure that both the PMA_checker and PMP_checker rules are enforced.

The address register holds the last checked address. Even if an address fails the PMP or PMA checks, it must be loaded into the address register, to provide the Exception Control Unit with information about the exception. It is loaded on a falling clock cycle.

The PMP and PMA_checker must only be combinatorial networks and shall not implement any registers what so ever.

## PMP & PMA Checker Reset

- reset address register to 0x00000000