CVA6

OpenHW contributors

Feb 06, 2023

CONTENTS

The goal of the CVA6 project is create a family of production quality, open source, application class RISC-V CPU cores. The CVA6 targets both ASIC and FPGA implementations, although individual cores may target a specific implementation technology. The CVA6 is written in SystemVerilog and is heavily parameterizable. For example parameters can set the ILEN to be either 32- or 64-bits and support for floating point can be enabled/disabled.

CHAPTER

ONE

CORE-V NOMENCLATURE

CORE-V is the name of the OpenHW Group family of RISC-V cores. CVA6 is the name of a GitHub repository for the source code for a set of application class CORE-V cores. The CV prefix identifies it as a member of the CORE-V family and the A6 indicates that it is an application class processor with a six stage execution pipeline. However, the CVA6 "as is" is not intended to implement a specific production core. Rather, the CVA6 is expected to be the basis for a number of application class cores. The naming convention for these cores is:

`CV <ILEN> <class> <# of pipeline stages> <product identifier>`

Thus, the CV64A60 would be a 64-bit application core with a six stage pipeline. Note that in this example, the product identifer is "0".

CHAPTER

TWO

ORGANIZATION OF THIS DOCUMENT

This documentation is split into multiple parts.

The *CVA6 User Guide* provides a detailed introduction to the CVA6. This document is based on the original Ariane documentation and is aimed at hardware developers integrating CVA6 into a design.

The *CVA6 Requirements Specification* is the top-level specification of the CVA6. One of the key attributes of this document is to specify the feature set of specific CORE-V products based on CVA6. This document focuses on _what_ the CVA6 does, without detailed consideration of _how_ a specific requirement is implemented. The target audience of this document is current and existing members of the OpenHW Group who wish to participate in the definition of future cores based on the CVA6.

The *CV32A6 Design Specification* describes in detail the **CV32A6**, the first production quality 32-bit application processor derived from the CVA6. The primary audience for this documentation are design and verification engineers working to bring the CV32A6 to TRL-5.

2.1 OpenHW Group CVA6 User Manual

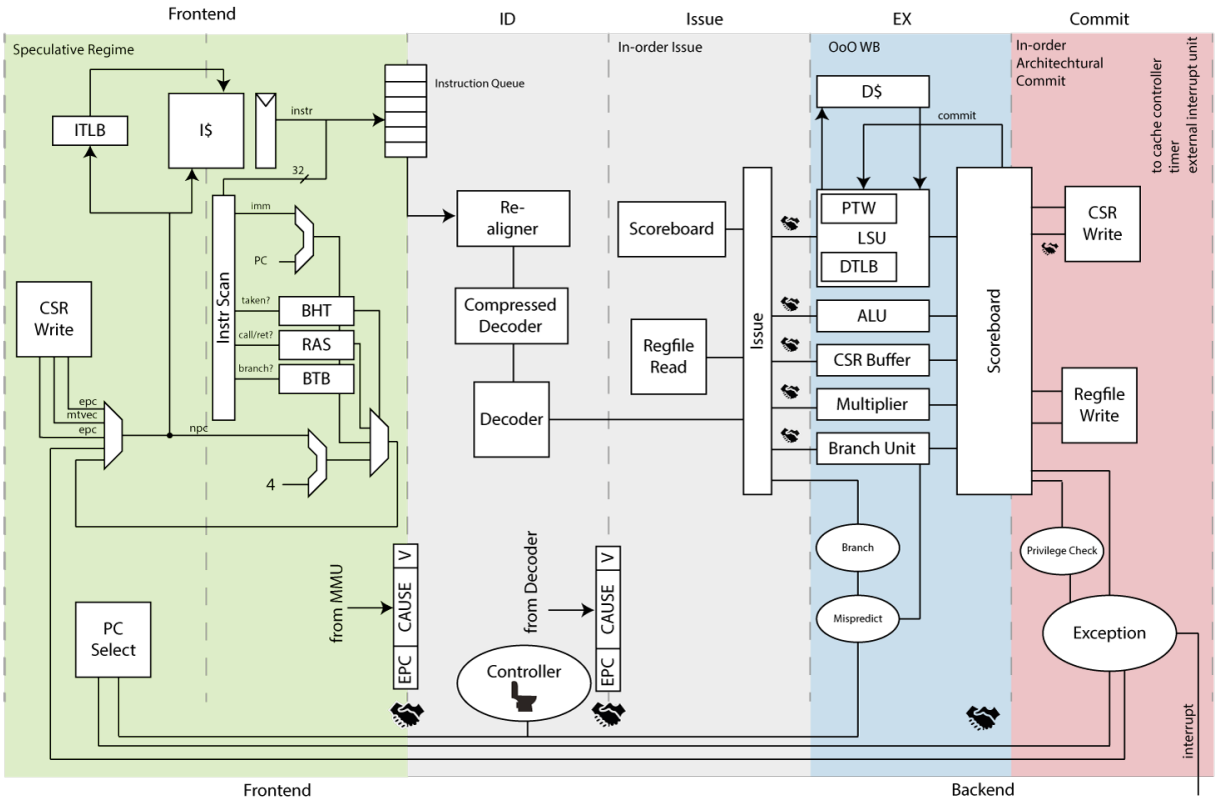
Editor: **Florian Zaruba** florian@openhwgroup.org

2.1.1 Introduction

This document describes the 6-stage, single issue Ariane CPU which implements the 64-bit RISC-V instruction set. It fully implements I, M and C extensions as specified in Volume I: User-Level ISA V 2.1 as well as the draft privilege extension 1.10. It implements three privilege levels M, S, U to fully support a Unix-like operating system.

Scope and Purpose

The purpose of the core is to run a full OS at reasonable speed and IPC. To achieve the necessary speed the core features a 6-stage pipelined design. In order to increase the IPC the CPU features a scoreboard which should hide latency to the data RAM (cache) by issuing data-independent instructions. The instruction RAM has (or L1 instruction cache) an access latency of 1 cycle on a hit, while accesses to the data RAM (or L1 data cache) have a longer latency of 3 cycles on a hit.



2.1.2 PC Generation

PC gen is responsible for generating the next program counter. All program counters are logical addressed. If the logical to physical mapping changes a `fence.vm` instruction should flush the pipeline and TLBs.

This stage contains speculation on the branch target address as well as the information if the branch is taken or not. In addition, it houses the branch target buffer (BTB) and a branch history table (BHT).

If the BTB decodes a certain PC as a jump the BHT decides if the branch is taken or not. Because of the various statefull memory components this stage is split into two pipeline stages. PC Gen communicates with the IF via a handshake signal. Instruction fetch signals its readiness with an asserted ready signal while PC Gen signals a valid request by asserting the `fetch_valid` signal.

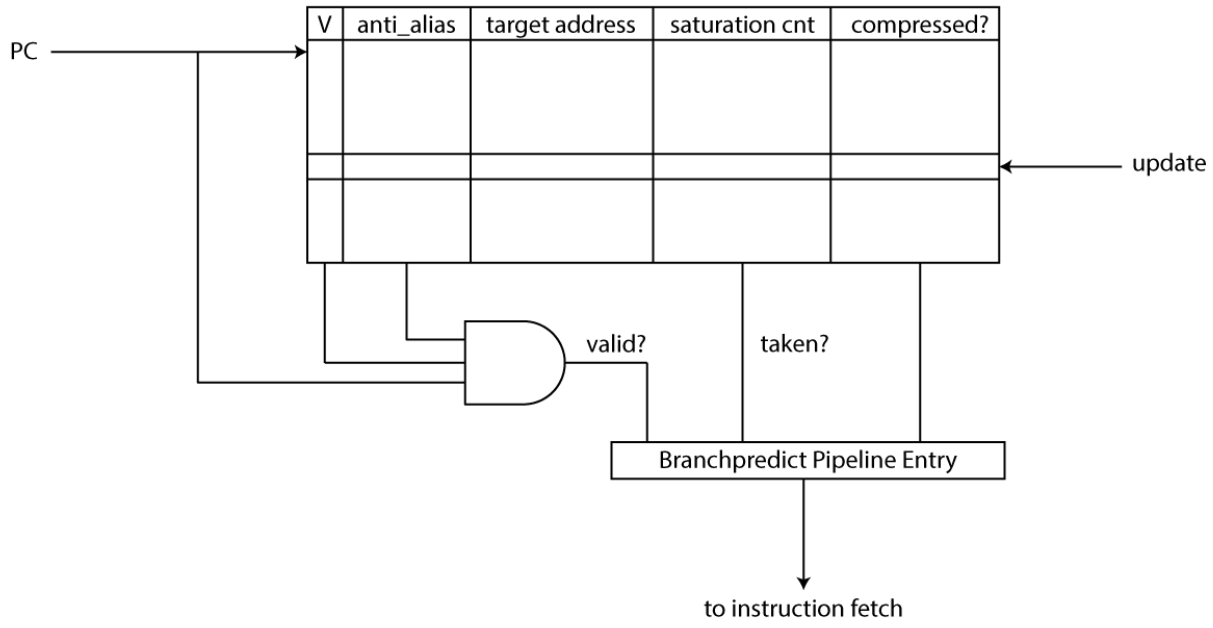
The next PC can originate from the following sources (listed in order of precedence):

1. **Default assignment**: The default assignment is to fetch PC + 4. PC Gen always fetches on a word boundary (32-bit). Compressed instructions are handled in a later pipeline step.
2. **Branch Predict**: If the BHT and BTB predict a branch on a certain PC, PC Gen sets the next PC to the predicted address and also informs the IF stage that it performed a prediction on the PC. This is needed in various places further down the pipeline (for example to correct prediction). Branch information which is passed down the pipeline is encapsulated in a structure called `branchpredict_sbe_t`. In contrast to branch prediction information which is passed up the pipeline which is just called `bp_resolve_t`. This is used for corrective actions (see next bullet point). This naming convention should make it easy to detect the flow of branch information in the source code.
3. **Control flow change request**: A control flow change request occurs from the fact that the branch predictor mispredicted. This can either be a 'real' mis-prediction or a branch which was not recognized as one. In any case we need to correct our action and start fetching from the correct address.

4. **Return from environment call**: A return from an environment call performs corrective action of the PC in terms of setting the successive PC to the one stored in the `[m|s]epc` register.
5. **Exception/Interrupt**: If an exception (or interrupt, which is in the context of RISC-V systems quite similar) occurs PC Gen will generate the next PC as part of the trap vector base address. The trap vector base address can be different depending on whether the exception traps to S-Mode or M-Mode (user mode exceptions are currently not supported). It is the purpose of the CSR Unit to figure out where to trap to and present the correct address to PC Gen.
6. **Pipeline Flush because of CSR side effects**: When a CSR with side-effects gets written we need to flush the whole pipeline and start fetching from the next instruction again in order to take the up-dated information into account (for example virtual memory base pointer changes).
7. **Debug**: Debug has the highest order of precedence as it can interrupt any control flow requests. It also the only source of control flow change which can actually happen simultaneously to any other of the forced control flow changes. The debug unit reports the request to change the PC and the PC which the CPU should change to.

This unit also takes care of a signal called `fetch_enable` which purpose is to prevent fetching if not asserted. Also note that no flushing takes place in this unit. All the flush information is distributed by the controller. Actually the controller's only purpose is to flush different pipeline stages.

Branch Prediction



Ariane

Block Diagram

All branch prediction data structures reside in a single register-file like data structure. It is indexed with the appropriate number of bits from the PC and contains information about the predicted target address as well as the outcome of a

configurable-width saturation counter (two by default). The prediction result is used in the subsequent stage to jump (or not).

In addition of providing prediction result the BTB also updates its information on mis-predictions. It can either correct the saturation counter or clear the branch prediction entry. The latter is done when the branch unit saw that the predicted PC didn't match or an when an instruction with privilege changing side-effect is committing.

The branch-outcome and the branch target address are calculated in the same functional unit therefore a mis-prediction on the target address is as costly as a mis-prediction on the branch decision. As the branch unit (the functional unit which does all the branch-handling) is already quite critical in terms of timing this is a potential improvement.

As Ariane fully implements the compressed instruction set branches can also happen on 16-bit (or half word) instructions. As this would significantly increase the size of the BTB the BTB is indexed with a word aligned PC. This brings the potential draw-back that branch-prediction does always mis-predict on a instruction fetch word which contains two compressed branches. However, such case should be rare in reality.

A trick we played here is to take the next PC (e.g.: the word aligned PC of the upper 16-bit of this instruction) of an un-aligned instruction to index the BTB. This naturally allows the the IF stage to fetch all necessary instruction data. Actually it will fetch two more unused bytes which are then discarded by the instruction re-aligner. For that reason we also need to keep an additional bit whether the instruction is on the lower or upper 16-bit.

For branch prediction a potential source of unnecessary pipeline bubbles is aliasing. To prevent aliasing from happening (or at least make it more unlikely) a couple of tag bits (upper bits from the indexed PC) are used and compared on every access. This is a trade-off necessary as we are lacking sufficiently fast SRAMs which could be used to host the BTB. Instead we are forced to use register which have a significantly larger impact on over all area and power consumption.

2.1.3 Instruction Fetch Stage

Instruction Fetch stage (IF) gets its information from the PC Gen stage. This information includes information about branch prediction (was it a predicted branch? which is the target address? was it predicted to be taken?), the current PC (word-aligned if it was a consecutive fetch) and whether this request is valid. The IF stage asks the MMU to do address translation on the requested PC and controls the I$ (or just an instruction memory) interface. The instruction memory interface is described in more detail in .

The delicate part of the instruction fetch is that it is very timing critical. This fact prevents us from implementing some more elaborate handshake protocol (as round-times would be too large). Therefore the IF stage signals the I$ interface that it wants to do a fetch request to memory. Depending on the cache's state this request may be granted or not. If it was granted the instruction fetch stage puts the request in an internal FIFO. It needs to do so as it has to know at any point in time how many transactions are outstanding. This is mostly due to the fact that instruction fetch happens on a very speculative basis because of branch prediction. It can always be the case that the controller decides to flush the instruction fetch stage in which case it needs to discard all outstanding transactions.

The current implementation allows for a maximum of two outstanding transaction. If there are more than two the IF stage will simply not acknowledge any new request from PC Gen. As soon as a valid answer from memory returns (and the request is not considered out-dated because of a flush) the answer is put into a FIFO together with the fetch address and the branch prediction information.

Together with the answer from memory the MMU will also signal potential exceptions. Therefore this is the first place where exceptions can potentially happen (bus errors, invalid accesses and instruction page faults).

Fetch FIFO

The fetch FIFO contains all requested (valid) fetches from instruction memory. The FIFO currently has one write port and two read ports (of which only one is used). In a future implementation the second read port could potentially be used to implement macro-op fusion or widen the issue interface to cover two instructions.

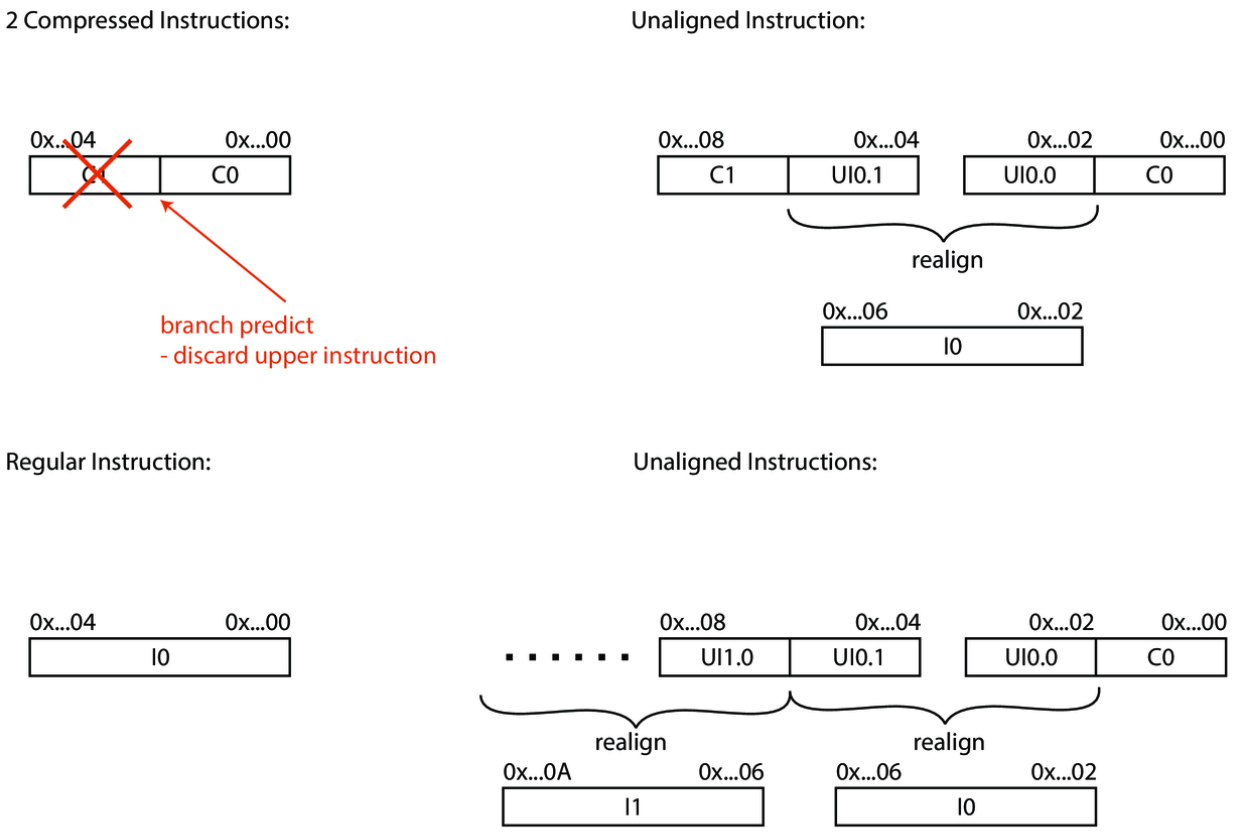
The fetch FIFO also fully decouples the processor's front-end and its back-end. On a flush request the whole fetch FIFO is reset.

2.1.4 Instruction Decode

Instruction decode is the fist pipeline stage of the processor's back-end. Its main purpose is to distill instructions from the data stream it gets from IF stage, decode them and send them to the issue stage.

With the introduction of compressed instructions (in general variable length instructions) the ID stage gets a little bit more complicated: It has to search the incoming data stream for potential instructions, re-align them and (in the case of compressed instructions) decompress them. Furthermore, as we will know at the end of this stage whether the decoded instruction is branch instruction it passes this information on to the issue stage.

Instruction Re-aligner



Instruction re-alignment Process

As mentioned above the instruction re-aligner checks the incoming data stream for compressed instructions. Compressed instruction have their last bit unequal to 11 while normal 32-bit instructions have their last two bit set to 11. The main complication arises from the fact that a compressed instruction can make a normal instruction unaligned

(e.g.: the instruction starts at a half word boundary). This can (in the worst case) mandate two memory accesses before the instruction can be fully decoded. We therefore need to make sure that the fetch FIFO has enough space to keep the second part of the instruction. Therefore the instruction re-aligner needs to keep track of whether the previous instruction was unaligned or compressed to correctly decide what to do with the upcoming instruction.

Furthermore, the branch-prediction information is used to only output the correct instruction to the issue stage. As we only predict on word-aligned PCs the passed on branch prediction information needs to be investigated to rule out which instruction we are actually need, in case there are two instructions (compressed or unaligned) present. This means that we potentially have to discard one of the two instructions (the instruction before the branch target). For that reason the instruction re-aligner also needs to check whether this fetch entry contains a valid and taken branch. Depending on whether it is predicted on the upper 16 bit it has to discard the lower 16 bit accordingly. This process is illustrate in .

Compressed Decoder

As mentioned earlier we also need to decompress all the compressed instructions. This is done by a small combinatorial circuit which takes a 16-bit compressed instruction and expands it to its 32-bit equivalent. All compressed instructions have a 32-bit equivalent.

Decoder

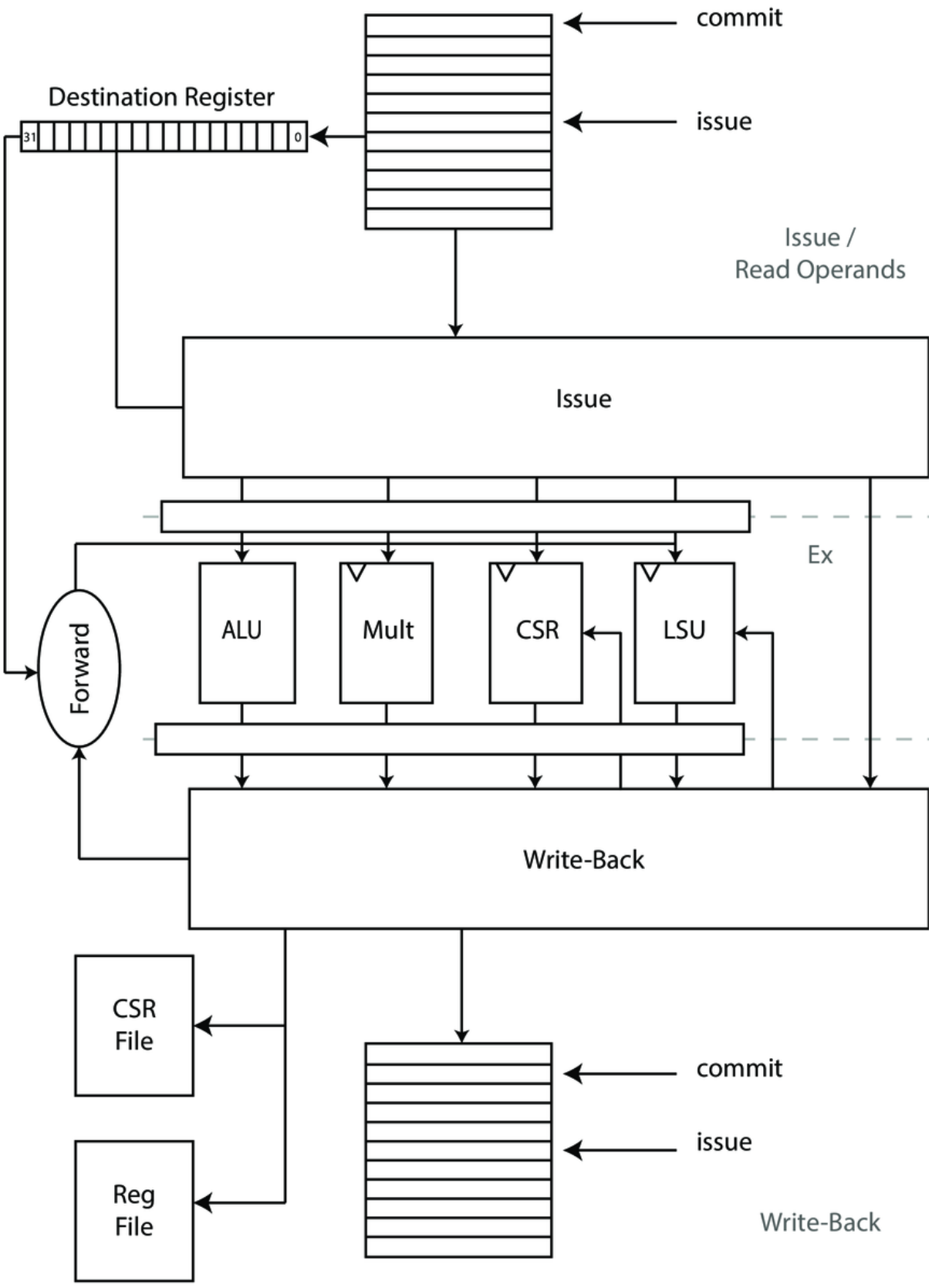
The decoder either takes the raw instruction data or the uncompressed equivalent of the 16-bit instruction and decodes them accordingly. It transforms the raw bits to the most fundamental control structure in Ariane, a scoreboard entry:

- **PC**: PC of instruction
- **FU**: functional unit to use
- **OP**: operation to perform in each functional unit
- **RS1**: register source address 1
- **RS2**: register source address 2
- **RD**: register destination address
- **Result**: for unfinished instructions this field also holds the immediate
- **Valid**: is the result valid
- **Use I Immediate**: should we use the immediate as operand b?
- **Use Z Immediate**: use zimm as operand a
- **Use PC**: set if we need to use the PC as operand a, PC from exception
- **Exception**: exception has occurred
- **Branch predict**: branch predict scoreboard data structure
- **Is compressed**: signals a compressed instructions, we need this information at the commit stage if we want jump accordingly e.g.: +4, +2

It gets incrementally processed further down the pipeline. The scoreboard entry controls operand selection, dispatch and the execution. Furthermore it contains an exception entry which strongly ties the particular instruction to its potential exception. As the first time an exception could have occoured was already in the IF stage the decoder also makes sure that this exception finds its way into the scoreboard entry. A potential illegal instruction exception can occur during decoding. If this is the case and no previous exception has happened the decoder will set the corresponding exceptions field along with the faulting bits (in `[s|m]tval`). As this is not the only point in which illegal instruction exception can happen and an illegal instruction exception always asks for the faulting address in the `[s|m]tval` field this field gets set here anyway. But only if instruction fetch didn't throw an exception for this instruction yet.

2.1.5 Issue Stage

The issue stage's purpose is to receive the decoded instructions and issue them to the various functional units. Furthermore the issue stage keeps track of all issued instructions, the functional unit status and receives the write-back data from the execute stage. Furthermore it contains the CPU's register file. By using a data-structure called scoreboard (see) it knows exactly which instructions are issued, which functional unit they are in and which register they will write-back to. As previously mentioned you can roughly divide the execution in four parts **1. issue**, **2. read operands**, **3. execute** and **4. write-back**. The issue stage handles step one, two and four.


commit
Destination Register
31
0
issue
Issue /
Read Operands
Issue
Ex
Forward
ALU
Mult
CSR
LSU
Write-Back
CSR
File
Reg
File
commit
issue
Write-Back

Ariane

Scoreboard

Issue

When the issue stage gets a new decoded instruction it checks whether the required functional unit is free or will be free in the next cycle. Then it checks if its source operands are available and if no other, currently issued, instruction will write the same destination register. Furthermore it keeps track that no unresolved branch gets issued. The latter is mainly needed to simplify hardware design. By only allowing one branch we can easily back-track if we later find-out that we've mis-predicted on it.

By ensuring that the scoreboard only allows one instruction to write a certain destination register it easies the design of the forwarding path significantly. The scoreboard has a combinatorial circuit which outputs the status of all 32 destination register together with what functional unit will produce the outcome. This signal is called `rd_clobber`.

The issue stage communicates with the various functional units independently. This in particular means that it has to monitor their ready and valid signals, receive and store their write-back data unconditionally. It will always have enough space as it allocates a slot in the scoreboard for every issued instruction. This solves the potential structural hazards of smaller microprocessors. This modular design will also allow to explore more advanced issuing technique like out-of-order issue ().

The issuing of instructions happen in-order, that means order of program flow is naturally maintained. What can happen out-of-order is the write-back of each functional unit. Think for example, that the issue stage issues a multiplication which takes n clock cycles to produce a valid result. In the next cycle the issue stage issues an ALU instruction like an addition. The addition will just take one clock cycle to return and therefore return before the multiplication's result is ready. Because of this we need to assign IDs to the various issue stages. The ID resembles the (unique) position in which the scoreboard will store the result of this instruction. The ID (called transaction ID) has enough bits to uniquely represent each slot in the scoreboard and needs to be passed along with the other data to the corresponding functional unit.

This scheme allows the functional units to operate in complete independence of the issue logic. They can return different transactions in different order. The scoreboard will know where to put them as long as the corresponding ID is signaled alongside the result. This scheme even allows the functional unit to buffer results and process them entirely out-of-order if it makes sense to them. This is a further example of how to efficiently decouple the different modules of a processor.

Read Operands

Read operands is physically happens in the same cycle as the issuing of instructions but can be conceptually thought of as another stage. As the scoreboard knows which registers are getting written it can handle the forwarding of those operands if necessary. The design goal was to execute two ALU instructions back to back (e.g.: with no bubble in between). The operands come from either the register file (if no other instruction currently in the scoreboard will write that register) or be forwarded by the scoreboard (by looking at the `rd_clobber` signal).

The operand selection logic is a classical priority selection giving precedence to results form the scoreboard over the register file as the functional unit will always produce the more up to date result. To obtain the right register value we need to poll the scoreboard for both source operands.

Scoreboard

The scoreboard is implemented as a FIFO with one read and one write port with valid and acknowledge signals. In addition to that it provides the aforementioned signals which tell the rest of the CPU which registers are going to be clobbered by a previously scheduled instruction. Instruction decode directly writes to the scoreboard if it is not already full. The commit stage looks for already finished instructions and updates the architectural state. Which either means going for an exception, updating the register or CSR file.

2.1.6 Execute Stage

The execute stage is a logical stage which encapsulates all the functional units (FUs). The FUs are not supposed to have inter-unit dependencies for the moment, e.g.: every FU must be able to perform its operation independently of every other unit. Each functional unit maintains a valid signal with which it will signal valid output data and a ready signal which tells the issue logic whether it is able to accept a new request or not. Furthermore, as briefly explained in the section about instruction issue (), they also receive a unique transaction ID. The functional unit is supposed to return this transaction ID together with the valid signal an the result. At the time of this writing the execute stage houses an ALU, a branch unit, a load store unit (LSU), a CSR buffer and a multiply/divide unit.

ALU

The arithmetic logic unit (ALU) is a small piece of hardware which performs 32 and 64-bit subtraction, addition, shifts and comparisons. It always completes its operation in a single cycle and therefore does not contain any state-full elements. Its ready signal is always asserted and it simply passes the transaction ID from its input to its output. Together with the two operands it also receives an operator which tells it which operation to perform.

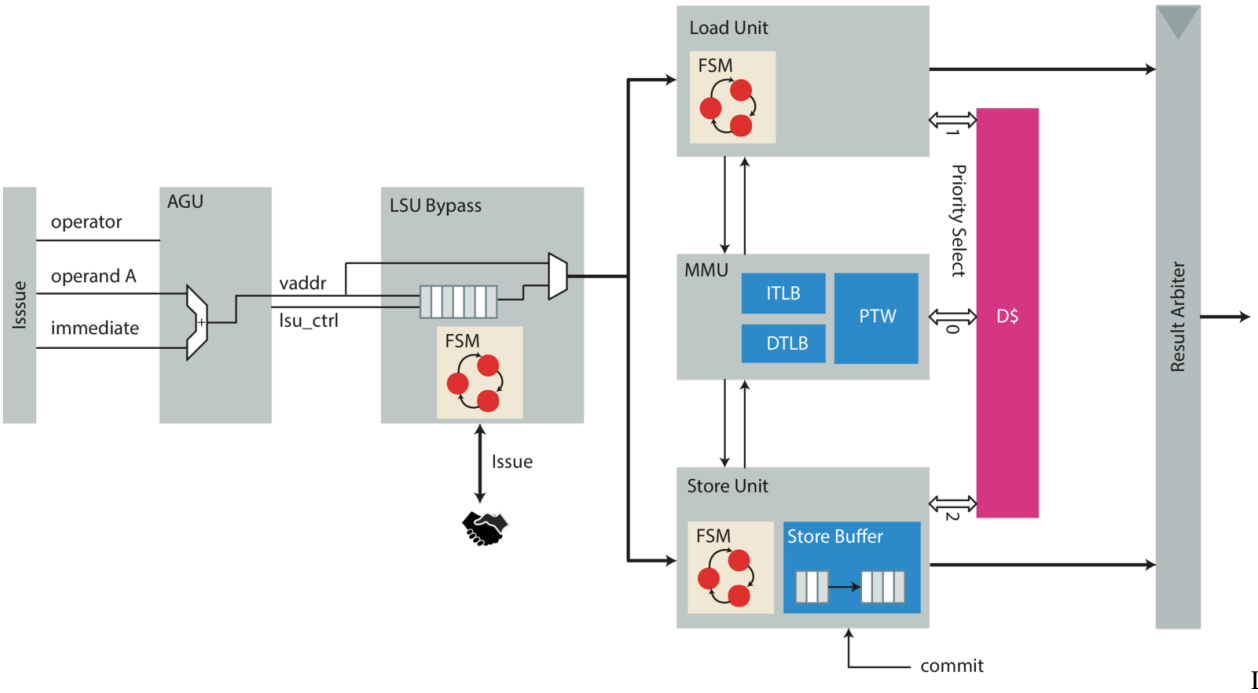
Branch Unit

The branch unit's purpose is to manage all kind of control flow changes i.e.: conditional and unconditional jumps. It does so by providing an adder to calculate the target address and some comparison logic to decide whether to take the branch or not. Furthermore it also decides if a branch was mis-predicted or not and reporting corrective actions to the PC Gen stage. Corrective actions include updating the BHT and setting the PC if necessary. As it can be that jumps are predicted on any instruction (including instructions which are no jumps at all - see aliasing problem in PC Gen section) it needs to know whenever an instruction gets issued to a functional unit and monitor the branch prediction information. If a branch was accidentally predicted on a non-branch instruction it also takes corrective action and re-sets the PC to the correct address (depending on whether the instruction was compressed or not it add PC + 2 or PC + 4).

As briefly mentioned in the section about instruction re-aligning the branch unit places the PC from an unaligned 32-bit instruction on the upper 16-bit (e.g.: on a new word boundary). Moreover if an instruction is compressed it also has an influence on the reported prediction as it needs to set a bit if the prediction occurred on the lower 16 bit (e.g.: the lower compressed instruction).

As can be seen this all adds a lot of costly operations to this stage, mostly comparison and additions. Therefore the branch unit is on the critical path of the overall design. Nevertheless, it was our design-choice to keep branches a single cycle operation. Still, it could be the case that in a future version it might make sense to split this path. This would bring some costly IPC implications to the overall design mainly because of the current restriction that the scoreboard is only admitting new instructions if there are no unresolved branches. With a single cycle operation all branches are resolved in the same cycle of issue which doesn't introduce any pipeline stalls.

Load Store Unit (LSU)



Load/Store Unit

The load store unit is similar to every other functional unit. In addition, it has to manage the interface to the data memory (D$). In particular, it houses the DTLB (Data Translation Lookaside Buffer), the hardware page table walker (PTW) and the memory management unit (MMU). It also arbitrates the access to data memory between loads, stores and the PTW - giving precedence to PTW lookups. This is done in order to resolve TLB misses as soon as possible. A high level block diagram of the LSU can be found in .

The LSU can issue load request immediately while stores need to be kept back as long as the scoreboard does not issue a commit signal: This is done because the whole processor is designed to only have a single commit point (see). Because issuing loads to the memory hierarchy does not have any semantic side effects the LSU can issue them immediately, totally in contrast to the nature of a store. Stores alter the architectural state and are therefore placed in a store buffer only to be committed in a later step by the commit stage. Sometimes this is also called *posted-store* because the store request is posted to the store queue and waiting for entering the memory hierarchy as soon as the commit signal goes high and the memory interface is not in use.

Therefore, upon a load, the LSU also needs to check the store buffer for potential aliasing. Should it find uncommitted data it stalls, since it can't satisfy the current request.

This means:

- Two loads to the same address are allowed. They will return in issue order.
- Two stores to the same address are allowed. They are issued in-order by the scoreboard and stored in-order in the store buffer as long as the scoreboard didn't give the signal to commit them.
- A store followed by a load to the same address can only be satisfied if the store has already been committed (marked as committed in the store buffer). Otherwise the LSU stalls until the scoreboard commits the instruction. We cannot guarantee that the store will eventually be committed (e.g.: an exception occurred).

For the moment being, the LSU does not handle misaligned accesses. In particular this means that access which are not aligned to a 64 bit boundary for double word accesses, access which are not aligned to a 32-bit boundary for word

access and the accesses which are not aligned on 16-bit boundary for half word access. If encounters such a load or store it will throw a misaligned exception and lets the exception handler resolve the load or store. In addition to mis-aligned exceptions it can also throw page fault exceptions.

To ease the design of the LSU it is split in 6 major parts of which each is described in more detail in the upcoming paragraphs:

1. **LSU Bypass**
2. **D$ Arbiter**
3. **Load Unit**
4. **Store Unit**
5. **MMU (including TLBs and PTW)**
6. **Non-blocking data cache**

LSU Bypass {#par:lsu_bypass}

The LSU bypass module is a auxiliary module which manages the LSU status information (full flag etc.) which it presents to the issue stage. This is necessary for a the following reason: The design of the LSU is critical in most aspects as it directly interfaces the relatively slow SRAMs. It additionally needs to do some costly operation in sequence. The most costly (in terms of timing) being address generation, address translation and checking the store buffer for potential aliasing. Therefore it is only known very late whether the current load/store can go to memory or if additional cycles are needed. From which aliasing on the store buffer and TLB miss are the most prominent ones. As the issue stage relies on the ready signal to dispatch new instructions this would result in an overly long path which would considerably slow down the whole design because of some corner cases.

To mitigate this problem a FIFO is added which can hold another request from issue stage. Therefore the ready flag of the functional units can be delayed by one cycle which eases timing. The LSU bypass model further decouples the functional unit from the issue stage. This is mostly necessary as the issue stage can't stall as soon as it issued an instruction. In particular the LSU bypass is called that way because it is either bypassed or serves the load or store unit from its internal FIFO until they signal completion to the LSU bypass module.

Load Unit {#par:load_unit}

The load unit takes care of all loads. Loads are issued as soon as possible as they do not have any side effects. Before issuing a load the load unit needs to check the store buffer for stores which are not committed into the memory hierarchy yet in order to avoid loading stale data. As a full comparison is quite costly only the lower 12 bit (the page-offset where physical and virtual addresses are the same) are compared. This has two major advantages: the comparison is only 12-bit instead of 64-bit and therefore faster when done on the whole buffer and the physical address is not needed which implies that we don't need to wait for address translation to finish. If the page offset matches with one of the outstanding stores the load unit simply stalls and waits until the store buffer is drained. As an improvement one could do some more elaborate data forwarding as the data in the store buffer is the most up-to-date. This is not done at the moment.

Furthermore the load unit needs to perform address translation. It makes use of virtually indexed and physically tagged D$ access scheme in order to reduce the number of cycles needed for load accesses. As it can happen that a load blocks the D$ it has to kill the current request on the memory interface to give way to the hardware PTW on the cache side. Some more advanced caching infrastructure (like a non-blocking cache) would alleviate this problem.

Store Unit {#par:store_unit}

The store unit manages all stores. It does so by calculating the target address and setting the appropriate byte enable bits. Furthermore it also performs address translation and communicates with the load unit to see if any load matches an outstanding store in one of its buffers. Most of the store units business logic resides in the store buffer which is described in detail in the next section.

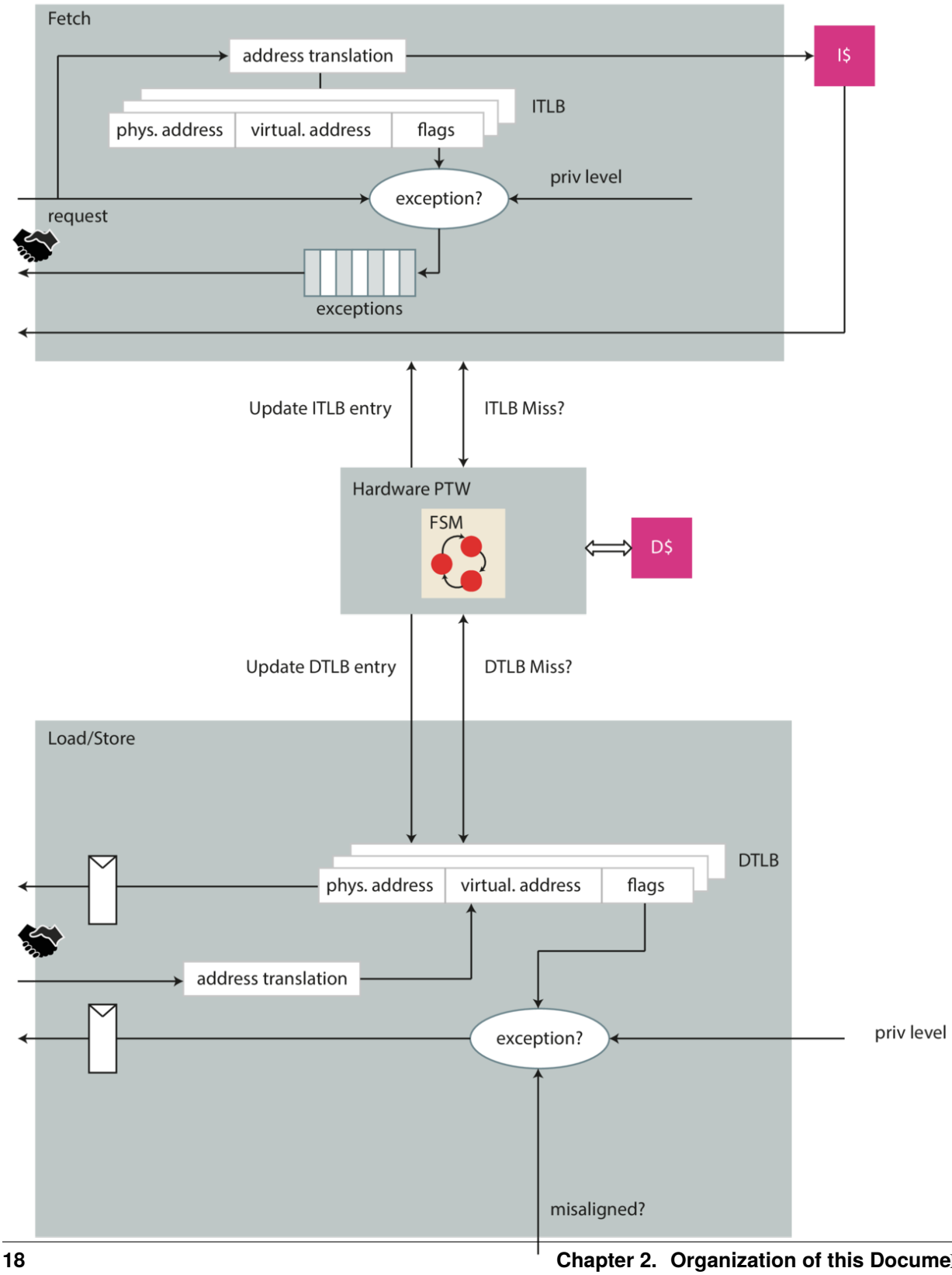
Store Buffer {#par:store_buffer}

The store buffer keeps track of all stores. It actually consists of two buffers: One is for already committed instructions and one is for outstanding instructions which are still speculative. On a flush only the instruction which are already committed are persisted while the speculative queue is completely emptied. To prevent buffer overflows the two queues maintain a full flag. The full flag of the speculative queue directly goes to the store unit, which will stall the LSU bypass module and therefore not receive any more requests. On the contrast the full signal of the commit queue goes to the commit stage. Commit stage will stall if it the commit queue can't accept any new data items. On every committed store the commit stage also asserts the `lsu_commit` signal which will put the particular entry from the speculative queue into the non-sepculative (commit) queue.

As soon as a store is in the commit queue the queue will automatically try to commit the oldest store in the queue to memory as soon as the cache grants the request.

The store buffer only works with physical addresses. At the time when they are committed the translation is already correct. For stores in the speculative queue addresses are potentially not correct but this fact will resolve if address translation data structures are updated as those instructions will also automatically flush the whole speculative buffer.

Memory Management Unit (MMU) {#par:mmu}



The memory management unit (MMU) takes care of address translation (see) and memory accesses in general. Address translation needs to be separately activated by writing the corresponding control and status register and switching to a lower privilege mode than machine mode. As soon as address translation is enabled it will also handle page faults. The MMU contains an ITLB, DTLB and hardware page table walker (HPTW). Although logically not really entangled - the fetch interface is also routed through the MMU. In general the fetch and data interface are handled differently. They only share the HPTW with each other (see .

There are mainly two fundamentally different paths through the MMU: one from the instruction fetch stage and the other from the LSU. Lets begin with the instruction fetch interface: The IF stage makes a request to get the memory content at a specific address. Instruction fetch will always ask for virtual addresses. Depending on whether the address translation is enabled the MMU will either transparently let the request directly go to the I$ or do address translation.

In case address translation is activated, the request to the instruction cache is delayed until a valid translation can be found. If no valid translation can be found the MMU will signal this with an exception. Furthermore, if an address translation can be performed with a hit on the ITLB it is a purely combinational path. The TLB is implemented as a fully set-associative caches made out of flops. This in turn means that the request path to memory is quite long and may become critical quite easily.

If an exception occurred the exception is returned to the instruction fetch stage together with the valid signal and not the grant signal. This has the implication that we need to support multiple out-standing transactions on the exception path as well (see). The MMU has a dedicated buffer (FIFO) which stores those exceptions and returns them as soon as the answer is valid.

The MMUs interface on the data memory side (D$) is entirely different. It has a simple request-response interfaces guarded by handshaking signals. Either the load unit or the store unit will ask the MMU to perform address translation. However the address translation process is not combinatorial as it is the case for the fetch interface. An additional bank of registers delays the MMU's answer (on a TLB hit) an additional cycle. As already mentioned in the previous paragraph address translation is a quite critical process in terms of timing. The particular problem on the data interface is the fact that the LSU needs to generate the address beforehand. Address generation involves another costly addition. Together with address translation this path definitely becomes critical. As the data cache is virtually indexed and physical tagged this additional cycle does not cost any loss in IPC. But, it makes the process of memory requests a little bit more complicated as we might need to abort memory accesses because of exceptions. If an exception occurred on a load request the load unit needs to kill the memory request it sent the cycle earlier. An excepting load (or store) will never go to memory.

Both TLBs are fully set-associative and configurable in size. Also the application specifier ID (ASID) can be changed in size. The ASID can prevent flushing of certain regions in the TLB (for example when switching applications). This is currently **not implemented**.

Page Table Walker (PTW)

The purpose of a page table walker has already been introduced in . The page table walker listens on both ITLB and DTLB for incoming translation requests. If it sees that either one of the requests is missing on the TLB it saves the virtual address and starts its page table walk. If the page table walker encounters any error state it will throw a page fault exception which in return is caught by the MMU and propagated to either the fetch interface or the LSU.

The page table walker gives precedence to DTLB misses. The page table walking process is described in more detail in the RISC-V Privileged Architecture.

PMA/PMP Checks

The core supports PMA and PMP checks in physical mode as well as with virtual memory enabled. PMA checks are performed only on the final access to the (translated) physical address. However, PMPs must be checked during the page table walk as well. During a page walk, all memory access must pass the PMP rules.

The amount of entries is parametrizable under the `ArianeCfg.NrPMPEntries` parameter. However, the core only supports granularity 8 (G=8). This simplifies the implementation since we do not have to worry about any unaligned accesses. There are a total of three distinct PMP units in the design. They verify instruction accesses, data loads and stores, and the page table walk respectively.

MMU Implementation Details

The MMU prioritizes instruction address translations to data address translations. The behavior of the MMU is described in the following:

1. As soon as a request from the instruction fetch stage arrives, the ITLB checked for a cached entry (combinatorial path). Upon a cache miss, the PTW is invoked.
2. The PTW will perform the page table walk in multiple cycles. During this walk, the PTW will update the content of the ITLB. The MMU checks every cycle if a cache hit in the ITLB exists, and therefore, the page table walk has concluded.

Multiplier

The multiplier contains a division and multiplication unit. Multiplication is performed in two cycles and is fully pipelined (re-timing needed). The division is a simple serial divider which needs 64 cycles in the worst case.

CSR Buffer

The CSR buffer a functional unit which its only purpose is to store the address of the CSR register the instruction is going to read/write. There are two reasons why we need to do this. The first reason is that an CSR instruction alters the architectural state, hence this instruction has to be buffered and can only be executed as soon as the commit stage decides to commit the instruction. The second reason is the way the scoreboard entry is structured: It has only one result field but for any CSR instruction we need to keep the data we want to write and the address of the CSR which this instruction is going to alter. In order to not clutter the scoreboard with some special case bit fields the CSR buffer comes into play. It simply holds the address and if the CSR instruction is going to execute it will use the stored address.

The clear disadvantage is that with the buffer being just one element we can't execute more than one CSR instruction back to back without a pipeline stall. Since CSR instructions are quite rare this is not too much of a problem. Some CSR instructions will cause a pipeline flush anyway.

2.1.7 Commit Stage

The commit stage is the last stage in the processor's pipeline. Its purpose is to take incoming instruction and update the architectural state. This includes writing CSR registers, committing stores and writing back data to the register file. The golden rule is that no other pipeline stage is allowed to update the architectural state under any circumstances. If it keeps an internal state it must be re-settable (e.g.: by a flush signal, see).

We can distinguish two categories of retiring instructions. The first category just write the architectural register file. The second might as well write the register file but needs some further business logic to happen. At the time of this writing the only two places where this is necessary it the store unit where the commit stage needs to tell the store unit

to actually commit the store to memory and the CSR buffer which needs to be freed as soon as the corresponding CSR instruction retires.

In addition to retiring instructions the commit stage also manages the various exception sources. In particular at time of commit exceptions can arise from three different sources. First an exception has occurred in any of the previous four pipeline stages (only four as PC Gen can't throw an exception). Second an exception happend during commit. The only source where during commit an exception can happen is from the CS register file and from an interrupt.

To allow precise interrupts to happen they are considered during the commit only and associated with this particular instruction. Because we need a particular PC to associate the interrupt with it, it can be the case that an interrupt needs to be deferred until another valid instruction is in the commit stage.

Furthermore commit stage controls the overall stalling of the processor. If the halt signal is asserted it will not commit any new instruction which will generate back-pressure and eventually stall the pipeline. Commit stage also communicates heavily with the controller to execute fence instructions (cache flushes) and other pipeline re-sets.

2.1.8 CVA6 System on Chip (SoC)

Memory Map

Base	Length	Attributes	Description
0x0000_0000	0x1000	EX	Debug Module
0x0001_0000	0x10000	EX	ROM
0x0200_0000	0xC0000		CLINT
0x0C00_0000	0x400_0000		PLIC
0x1000_0000	0x1000		UART
0x1800_0000	0x1000		Timer
0x2000_0000	0x80_0000		SPI
0x3000_0000	0x10000		Ethernet
0x4000_0000	0x1000		GPIO
0x8000_0000	0x4000_0000	EX, NI, C	DRAM

(EX: Executable, NI: Non-idempotent, C: Cached)

Platform-Level Interrupt Controller (PLIC)

The specification of CVA6's platform-level interrupt controller (PLIC) is aligned with the PLIC of SiFive's FU540-C000. It shares the same functionality and memory map and has the following interrupt sources:

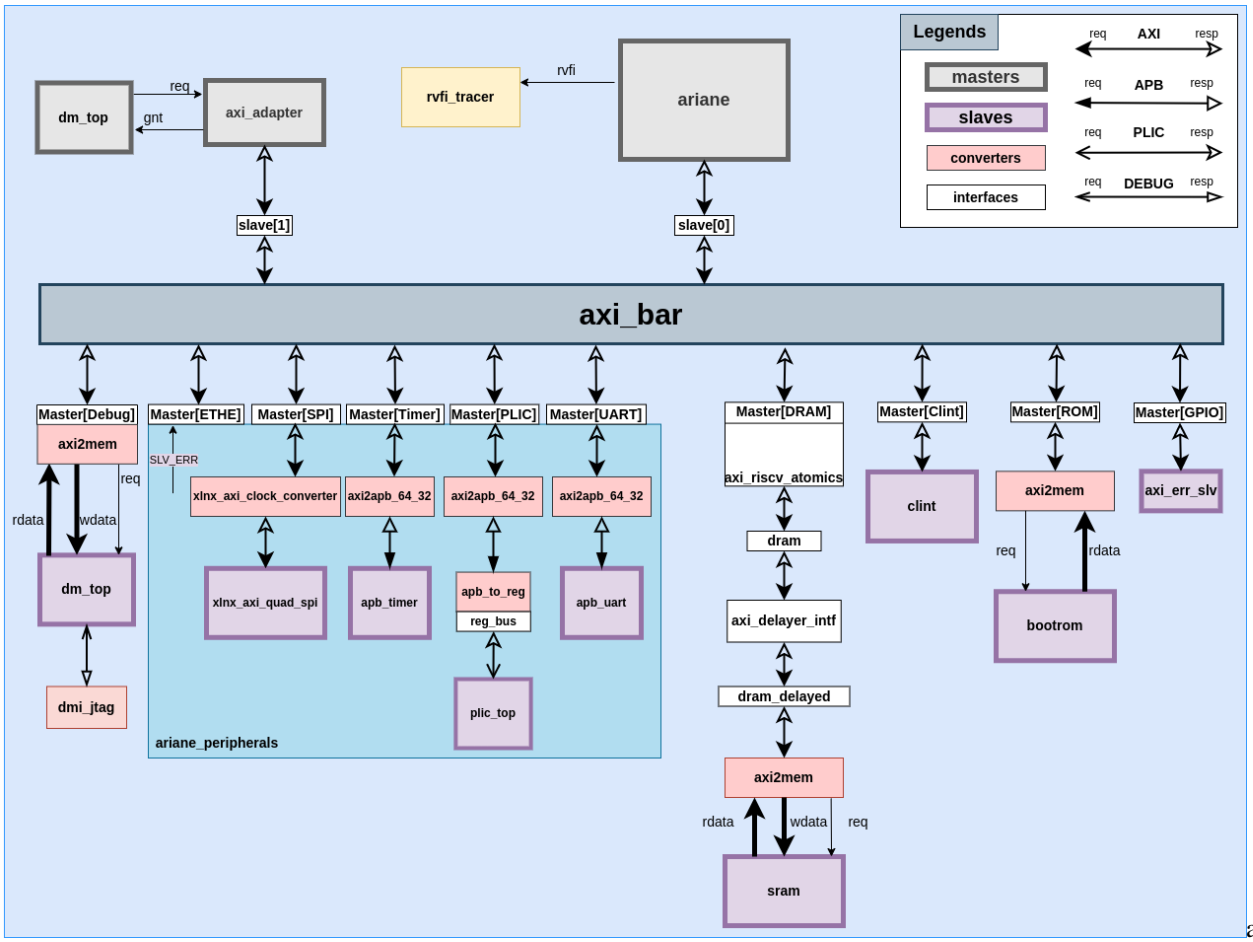
Interrupt ID	Source
1	UART
2	SPI
3	Ethernet
4	Timer 0 (OVF)
5	Timer 0 (CMP)
6	Timer 1 (OVF)
7	Timer 1 (CMP)
8 – 30	*Reserved*

2.1.9 CVA6 Testharness

`ariane_testharness` is the module where all the masters and slaves have been connected with the axi crossbar.There are two masters and ten slaves in this module.Their names and interfaces have been mentioned in the table below.

Slaves	Interfaces	Masters	Interfaces	
DRAM	master[0]	ariane	slave[0]
GPIO	master[1]	debug	slave[1]
Ethernet	master[2]
SPI	master[3]
Timer	master[4]
UART	master[5]
PLIC	master[6]
CLINT	master[7]
ROM	master[8]
Debug	master[9]

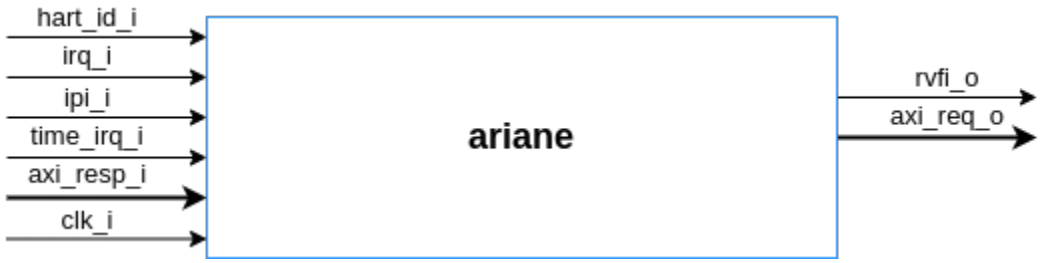
The following block diagram shows the connections of the slaves and masters in the `ariane_testharness` module.



ariane_testharness

Ariane

The `ariane` core is instantiated as `i_ariane` in `ariane_testharness` module. It is acting as a master in `ariane_testharness`.The following is the diagram of the `ariane` module along with its inputs/outputs ports.



ariane

ipi, irq and time_irq are being sent to this module from the ariane_testharness module.The AXI request and response signals that are being passed from the ariane_testharness to ariane module are the following:

```
.axi_req_o ( axi_ariane_req ),.axi_resp_i ( axi_ariane_resp )
```

In the ariane_testharness module, axi_ariane_req and axi_ariane_resp structs are being linked with the slave[0] (AXI_BUS interface) in a way that the information of axi_ariane_req is being passed to the slave[0] and the information from the slave[0] is being passed to the axi_ariane_resp struct. The following compiler directives are being used for this purpose.

```
AXI_ASSIGN_FROM_REQ(slave[0], axi_ariane_req) AXI_ASSIGN_TO_RESP(axi_ariane_resp,
slave[0])
```

Rvfi_o is the output of ariane and it will go into the rvfi_tracer module.

Debug

Master

axi_adapter is acting as a master for the debug module.The following is the diagram of the axi_adapter module along with its signals.

axi_adapter

The AXI request and response that signals are being passed from the test_harness module are the following:

```
.axi_req_o ( dm_axi_m_req ).axi_resp_i ( dm_axi_m_resp )
```

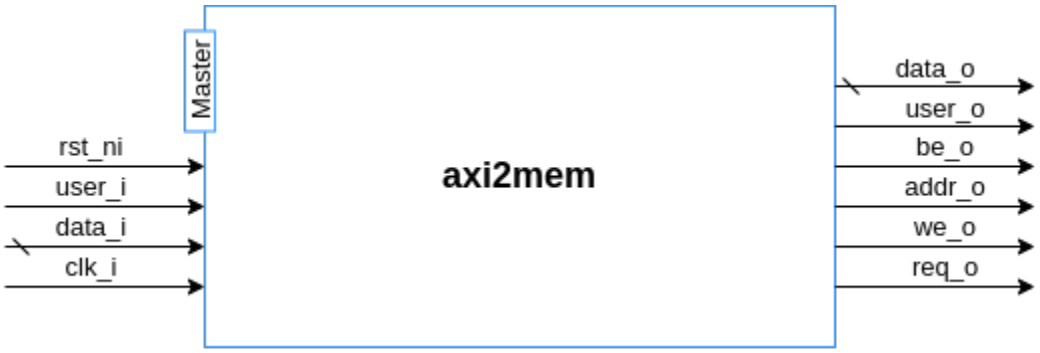
Slave[1] is the interface of AXI_BUS and it actually acts as a master for axi_protocol.

The dm_axi_m_req and dm_axi_m_resp are being linked with the slave[1] AXI_BUS interface in this way that the requests signals of the dm_axi_m_req are being passed to the slave[1] and the response signals from the slave[1] are being passed to the dm_axi_m_resp struct.

```
AXI_ASSIGN_FROM_REQ(slave[1], dm_axi_m_req)    AXI_ASSIGN_TO_RESP(dm_axi_m_resp,
slave[1])
```

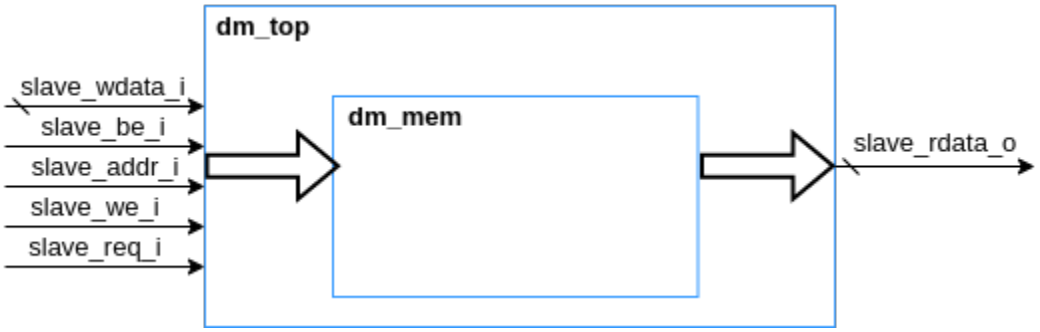
Slave

This is the memory of debug and `axi2mem` converter is used whenever a read or write request is made to memory by the master.`axi2mem` module simply waits for the ar_valid or aw_valid of the master (actual slave) interface and then passes the req_o, we_o, addr_o, be_o, user_o signals and data_o to the memory and will receive the data_i and user_i from the memory.



axi2mem

The memory is has been instantiated in the `dm_top` module and the hierarchy is as follows:



dm_top_&_dm_mem

CLINT

Clint is a slave in this SoC. The signals of the `clint` module are as follows:

clint

`ipi_o` (inter-processing interrupt) and `timer_irq_o` (timer_interrupt request) are generated from the `clint` module and are the inputs of the ariane core.This module interacts with the axi bus interface through the following assignments:

```
AXI_ASSIGN_TO_REQ(axi_clint_req, master[ariane_soc::CLINT])
```

This compiler directive is used to transfer the request signals of the master via the interface mentioned as `master[ariane_soc::CLINT]` to the struct `axi_clint_req`.

```
AXI_ASSIGN_FROM_RESP(master[ariane_soc::CLINT], axi_clint_resp)
```

This compiler directive is used to assign the response of the slave (in this case `clint` module) from the`Axi_clint_resp` struct to the interface `master[ariane_soc::CLINT]`.

Bootrom

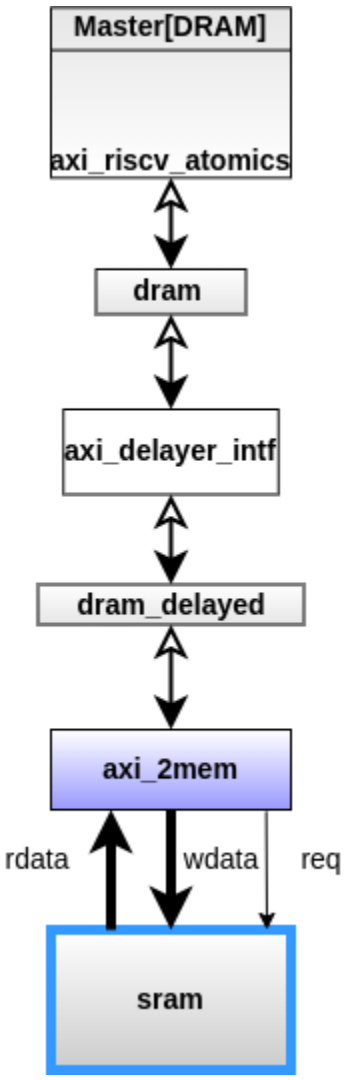
`axi2mem` module is used to communicate with `bootrom` module. The signals of this memory have been shown in the diagram below:

bootrom

Bootrom is pre-initialized with `ROM_SIZE = 186`.

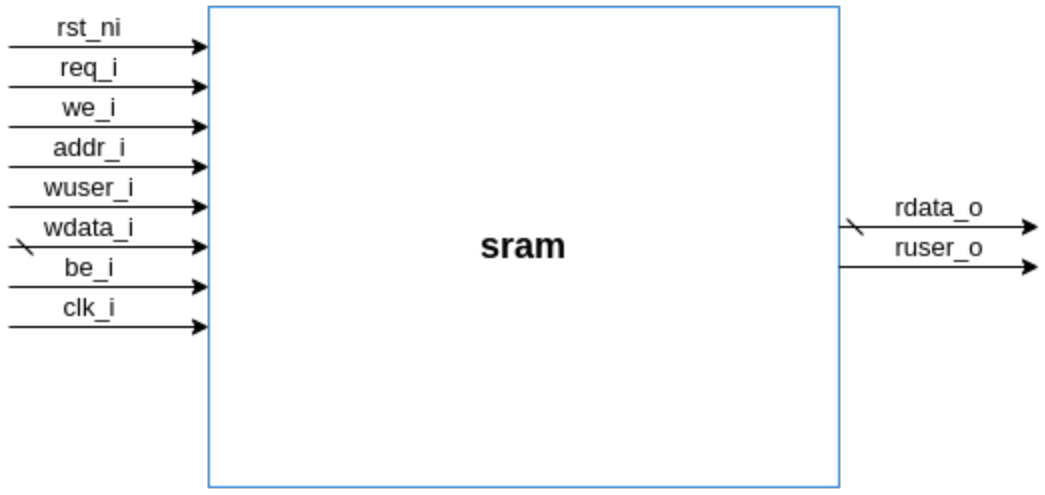
SRAM

The complete sequence through which a request to SRAM is transferred is as follows:



sequence

`dram` and `dram_delayed` are two AXI_BUS interfaces. The slave modport of AXI_BUS interface for `Master[DRAM]` has been linked with `axi_riscv_atomics` module and the request of the master has been passed to `dram` interface (another instantiation of interface of AXI_BUS). All this is for the exclusive accesses and no burst is supported in this exclusive access.`dram` and `dram_delayed` interfaces have also been passed to `axi_delayer_intf` module as a slave modport and master modport of the AXI_BUS interface, respectively. The `axi_delayer_intf` module is used to introduce the delay.`dram_delayed` is also passed to the `axi2mem` module as a slave modport of AXI_BUS interface. `axi2mem` module with `dram_delayed` as an AXI_Bus interface will interact with SRAM.SRAM is a word addressable memory with the signals as follows:



sram

GPIO

GPIO is not implemented, error slave has been added in place of it.

UART

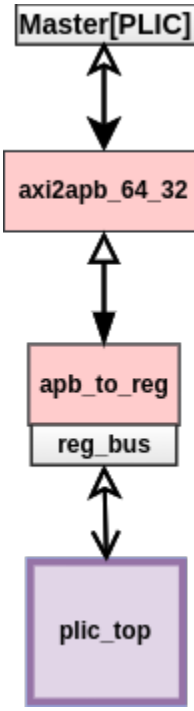
There are two signals for the `apb_uart` module in the `ariane_testharness`, namely `tx` and `rx` for transmitting and receiving the data.`axi2apb_64_32`, module has been used to convert the axi protocol five channel signals to a single channel apb signals. The `axi2apb_64_32` module has been used between AXI_BUS and `apb_uart module`.The signals of the `apb_uart` module have been shown in the diagram below:

apb_uart

Only the signals related to the test_harness have been shown in the above diagram.

PLIC

PLIC is a slave in this SoC. The hiearchy through which the request is propagated to the plic_top module is as follows:



plic_hierarchy

`axi2apb_64_32` has been used to convert all the plic axi signals into apb signals.apb_to_reg is used to assign the apb signals to the `reg_bus` interface which basically communicates with the `plic_top` module. In `apb_to_reg` module, the logical `AND` of `psel` and `penable` signals of apb makes the `valid` signal of `reg_bus` interface.The signals of the `plic_top` have been shown below:

plic_top

Timer

The `axi2apb_64_32` module has been used to convert all the timer axi signals into timer apb signals.The diagram of the apb_timer is as follows.

apb_timer

The signals of apb protocol have been shown in the form of `apb_timer_req` and `apb_timer_resp` in the above diagram.

Ethernet

Ethernet is a slave in this testharness.

Ethernet support has not been added in the 'ariane_testharness' at this time. For any read or write request from the master to this module is returned with

```
"ethernet.b_resp = axi_pkg::RESP_SLVERR"
```

where,

```
"localparam RESP_SLVERR = 2'b10;" in axi_pkg
```

which shows `"Slave error"`. It is used when the access has reached the slave successfully, but the slave wishes to return an error condition to the originating master."

SPI

SPI is a slave in this testharness. Support of the of SPI protocol is present in the SoC, but at this time it is turned off, as the `.spi_clk_o ( )`,`.spi_mosi ( )`,`.spi_miso ( )` ,and `.spi_ss ( )` signals of SPI have been left open in the `ariane_testharness` module. Any read or write request from the master to this module is returned with `"Slave error"`.

2.1.10 Indices and tables

- genindex
- modindex
- search

2.1.11 Documentation

The documentation is re-generated on pushes to master. When contributing to the project please consider the [contribution guide](https://github.com/openhwgroup/cva6/blob/master/CONTRIBUTING.md).

2.2 CVA6 Requirement Specification

Revision 1.0.1

2.2.1 License

2.2.2 Introduction

CVA6 is a RISC-V compatible application processor core that can be configured as a 32- or 64-bit core (RV32 or RV64). It includes L1 caches, optional MMU, optional PMP and optional FPU.

It is an industrial evolution of ARIANE created by ETH Zürich and the University of Bologna. It is written in SystemVerilog and maintained by the OpenHW Group.

This specification is organized as requirements that apply to the "Scope of the IP".

The requirement list is to be approved by the OpenHW Group Technical Work Group (TWG), as well as its change requests.

The specification will be complemented by a user's guide.

Revision 1.0.0 refers to the product of the first CVA6 project led at OpenHW Group. It is a placeholder in case of future evolutions after project freeze (PF gate).

A list of abbreviations is available at the end of this document.

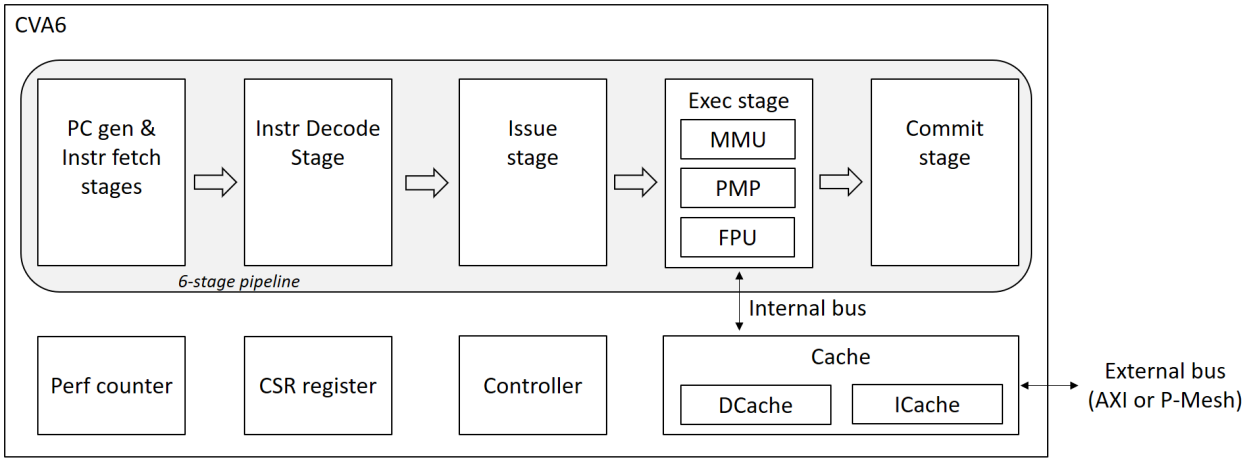
2.2.3 Scope

Scope of the IP

The **scope of the IP** is the subsystem that is specified below and that will undergo verification with a 100% coverage goal. In the verification plans, the scope of the IP can be broken down in several DUT (design under test).

The scope of the IP is the **CVA6 hardware** supporting all the features used in products based on CVA6.

CVA6 exists in two main configurations: **CV64A6** and **CV32A6**. A requirement referring to CVA6 applies to both configurations.



As displayed in the picture above, the IP comprises:

- The CVA6 core;
- L1 write-through cache;
- Optional FPU;
- Optional MMU;
- Optional PMP;
- CSR;
- Performance counters;
- AXI interface;
- Interface with the P-Mesh coherence system of OpenPiton.

These are not part of the IP (several solutions can be used):

- CLINT or PLIC Interrupt modules;
- Debug module (such as DTM);
- Support of L1 write-back cache (this might come later as an update).

In addition to these main configurations, several fine grain parameters are available.

Unless otherwise stated, an optional feature is controlled by a SystemVerilog parameter. If not selected, the optional feature will not be present in the netlist after synthesis.

The reader's attention is drawn to the difference between an optional feature ("…shall support as an option…") and a desired goal ("…should support…", "…should reduce latency…").

These are not in the scope of this specification:

- SW layers, such as compiler and OSes (that can however be part of the OpenHW Group CVA6 project);
- SW emulation of RISC-V optional extensions (feasible but the scope of the IP is the core hardware);
- Other features included in the testbench (main memory, firmware, interconnect…), the verification coverage of which will not be measured;
- The vector coprocessor (CV-VEC) that is planned to interface with CV64A6.

Initial Release

The CVA6 is highly configurable via SystemVerilog parameters. It is not practical to fully document and verify all possible combinations of parameters, so a set of "viable IP configurations" has been defined. The full list of parameters for this configuration will be detailed in the users' guide.

Below is the configuration of the first release of the CVA6.

Release ID	Target	ISA	XLEN	FPU	CV-X-IF	MMU	L1 D$	L1 I$
CV32A60X	ASIC	IMC	32	No	Yes	Sv32	None	16 kB

Possible Future Releases

Below is a proposed list of configurations that could undergo verification and their main parameters. The full list of parameters for these configurations will be detailed in the users' guide if and when these configurations are fully verified.

Configuation ID	Target	ISA	XLEN	FPU	CV-X-IF	MMU	L1 D$	L1 I$
cv32a6_imacf_sv32	FPGA	IMACF	32	Yes	TBD	Sv32	32 kB	16 kB
cv32a6_imac_sv32	FPGA	IMAC	32	No	TBD	Sv32	32 kB	16 kB
cv64a6_imacfd_sv39	ASIC	IMACFD	64	Yes	Yes	Sv39	16 kB	16 kB
cv32a6_imac_sv0	ASIC	IMAC	32	No	Yes	None	None	4 kB

2.2.4 References

Applicable specifications

To ease the reading, the reference to these specifications can be implicit in the requirements below. For the sake of precision, the requirements identify the versions of RISC-V extensions from these specifications.

[RVunpriv] "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 13, 2019.

[RVpriv] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203", Editors Andrew Waterman, Krste Asanović and John Hauser, RISC-V Foundation, December 4, 2021.

[RVdbg] "RISC-V External Debug Support, Document Version 0.13.2", Editors Tim Newsome and Megan Wachs, RISC-V Foundation, March 22, 2019.

[RVcompat] "RISC-V Architectural Compatibility Test Framework", https://github.com/riscv-non-isa/riscv-arch-test.

[AXI] AXI Specification, https://developer.arm.com/documentation/ihi0022/hc.

[CV-X-IF] Placeholder for the CV-X-IF coprocessor interface currently prepared at OpenHW Group; current version in https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/.

[OpenPiton] "OpenPiton Microarchitecture Specification", Princeton University, https://parallel.princeton.edu/openpiton/docs/micro_arch.pdf.

Reference documents

[RVcmo] "RISC-V Base Cache Management Operation ISA Extensions, version 1.0-fd39d01, 2022-01-12"

[CLINT] Core-Local Interruptor (CLINT), "SiFive E31 Core Complex Manual v2p0", chapter 6, https://static.dev.sifive.com/SiFive-E31-Manual-v2p0.pdf

2.2.5 Functional requirements

General requirement

GEN-10	CVA6 shall be **fully compliant with RISC-V specifications** [RVunpriv], [RVpriv] and [RVdbg] by implementing all mandatory features for the set of extensions that are selected and by passing [RVcompat] compatibility tests.

As the RISC-V specification leaves space for variations, this specification specificies some of these variations.

RISC-V standard instructions

To ease tracing to verification, the extensions have been split in independent requirements.

ISA-10	CV64A6 shall support **RV64I** base instruction set, version 2.1.
ISA-20	CV32A6 shall support **RV32I** base instruction set, version 2.1.
ISA-30	CVA6 shall support the **M** extension (integer multiply and divide), version 2.0.
ISA-40	CVA6 shall support the **A** extension (atomic instructions), version 2.1.
ISA-50	CV32A6 shall support as an **option** the **F** extension (single-precision floating-point), version 2.2.
ISA-60	CV64A6 shall support as an **option** the **F** and **D** extensions (single- and double-precision floating-point), version 2.2.
ISA-70	CV64A6 shall support as an **option** the **F** extension (single-precision without double-precision floating-point), version 2.2.
ISA-80	CVA6 shall support as an **option** the **C** extension (compressed instructions), version 2.0.
ISA-90	CVA6 shall support the **Zicsr** extension (CSR instructions), version 2.0.
ISA-100	CVA6 shall support the **Zifencei** extension, version 2.0.
ISA-110	As an **option**, the duration of instructions shall be independent from the operand values. *Unlike other options, this one can be design-time (selected before compiling the RTL) or run-time (selected through a register).*

Note to ISA-60 and ISA-70: CV64A6 cannot support the D extension with the F extension.

Note to ISA-110: In the current design, the duration of the division is data-dependent, which can be a security issue.

Privileges and virtual memory

The MMU includes a TLB and a hardware PTW.

PVL-10	CVA6 shall support **machine**, **supervisor, user** and **debug** privilege modes.
PVL-20	CV64A6 shall support as an **option** the **Sv39** virtual memory, version 1.11.
PVL-30	CV32A6 shall support as an **option** the **Sv32** virtual memory version 1.11.
PVL-40	CVA6 instances that do not feature virtual memory shall support the **Bare** mode.
PVL-50	CVA6 shall feature PMP (physical memory protection) as an **option**.
PVL-60	CV64A6 shall support as an **option** the **H** extension (hypervisor) version 1.0.

CSR

There are no requirements related to CSR as they derive from other requirements, such as PVL-10, PVL-60… Details of CSRs will be available in the user's manual.

Performance counters

Performance counters are important features for safety-critical applications.

HPM-10	CVA6 shall implement the 64-bit `mcycle` and `minstret` standard performance counters (including their upper 32 bits counterparts `mcycleh` and `minstreth` in CV32A6) as per [RVpriv].
HPM-20	CVA6 shall implement as an **option** six generic 64-bit performance counters located in `hpmcounter3` to `hpmcounter8` (including their upper 32 bits counterparts in CV32A6: `hpmcounter3h` to `hpmcounter8h`).
HPM-30	Each of the six generic performance counters shall be able to count events from one of these sources: 1. L1 I-Cache misses 2. L1 D-Cache misses 3. ITLB misses 4. DTLB misses 5. Load accesses 6. Store accesses 7. Exceptions 8. Exception handler returns 9. Branch instructions 10. Branch mispredicts 11. Branch exceptions 12. Call 13. Return 14. MSB Full 15. Instruction fetch Empty 16. L1 I-Cache accesses 17. L1 D-Cache accesses 18. L1$ line invalidation 19. I-TLB flush 20. Integer instructions 21. Floating point instructions 22. Pipeline bubbles
HPM-40	The source of events counted by the six generic performance counters shall be selected by the `mhpmevent3` to `mhpmevent8` CSRs.
HPM-50	CVA6 shall allow the supervisor access of performance counters through enabling of `mcounteren` CSR.
HPM-60	CVA6 shall allow the user access of performance counters through enabling of `scounteren` CSR.
HPM-70	CVA6 shall implement the `mcountinhibit` counter-inhibit register.
HPM-80	CVA6 shall implement the read-only `cycle`, `instret`, `hpmcounter3` to `hpmcounter8` access to counters (and their upper 32-bit counterparts in CV32A6).

The user's manual will detail the list of counters, events and related controls.

Cache requirements

Caches increase the performance of the processor with regard to memory accesses. Most of their added value for the IP is specified through performance requirements in another section. Here below are specific requirements for these caches.

The project would like to adopt the recently ratified [RVcmo] specification. The analysis yet needs to be performed and will likely lead to an evolution of this specification.

L1 write-through data cache

In the requirements below, L1WTD refers to the L1 write-through data cache that is part of the CVA6.

The first two requirements express the write-through feature. Some requirements are useful for security- and safety-critical applications where a high level of timing predictability is needed.

L1W-10	L1WTD shall reflect all write accesses (stores) by the CVA6 core to the external memory within an upper-bounded number of cycles. The upper-bound is fixed but not specified here.
L1W-20	L1WTD shall not change the order of write accesses to the external memory with respect to the order of write accesses (stores) received from the CVA6 core.
L1W-30	L1WTD should offer the following size/ways configurations: • 0 kbyte (no cache), • 4 kbytes (4 or 8 ways), • 8 kbytes (4, 8 or 16 ways), • 16 kbytes (4, 8 or 16 ways), • 32 kbytes (8 or 16 ways).
L1W-40	L1WTD shall support datasize extension to store EDC, ECC or other information. The numbers of bits of the extension is defined by a compile-time parameter.
L1W-50	To interface with the P-Mesh coherence system of OpenPiton, L1WTD shall have a line invalidate external command that invalidates the content of a line upon request.
L1W-60	Some physical memory regions shall be configurable as not L1WTD cacheable at design time.
L1W-70	It shall be possible to invalidate L1WTD content with the `FENCE.T` command.
L1W-80	The replacement policy of L1WTD shall be LFSR (pseudo-random) or LRU (least recently used).
L1W-90	L1WTD should offer a feature to transform cache ways into a scratchpad. Alternatively, this requirement can be realized with a separate scratchpad.
L1W-100	A custom CSR shall allow to disable or enable L1WTD.

Cache counters are defined in the performance counters.

32 kbytes & 4 ways is not feasible with the current architecture. Other size/ways configurations may be implemented in the design.

The design will support one replacement policy allowed by L1W-80.

L1 Instruction cache

In the requirements below, L1I refers to the L1 instruction cache that is part of the CVA6.

Some requirements are useful for security- and safety-critical applications where a high level of timing predictability is needed.

L1I-10	L1I should offer the following size/ways configurations: • 4 kbytes: 3, 4 or 8 ways, • 8 kbytes: 4, 8, or 16 ways, • 16 kbytes: 4, 8 or 16 ways, • 32 kbytes: 8 or 16 ways.
L1I-20	L1I shall support datasize extension to store EDC, ECC or other information. The numbers of bits of the extension is defined by a compile-time parameter.
L1I-30	To interface with the P-Mesh coherence system of OpenPiton, L1I shall have a line invalidate external command that invalidates the content of a line upon request.
L1I-40	It shall be possible to invalidate L1I content with the `FENCE.T` command.
L1I-50	The replacement policy of L1I shall be LFSR (pseudo-random) or LRU (least recently used).
L1I-60	L1I should offer a feature to transform cache ways into a scratchpad. Alternatively, this requirement can be realized with a separate scratchpad.
L1I-70	A custom CSR shall allow to disable or enable L1I.

Cache counters are defined in the performance counters section.

32 kbytes & 4 ways is not feasible with the current architecture. Other size/ways configurations may be implemented in the design.

The design will support one replacement policy allowed by L1I-50.

FENCE.T custom instruction

There are discussions within RISC-V International to define a specification for `FENCE.T`. The specification below reflects the situation prior to this RISC-V specification, based on Nils Wistoff's work. If a RISC-V specification is ratified, the CVA6 specification will likely switch to it.

FET-10	CVA6 shall support the `FENCE.T` instruction that ensures that the execution time of subsequent instructions is unrelated with predecessor instructions.
FET-20	`FENCE.T` shall be available in all privilege modes (machine, supervisor, user and hypervisor if present).

FENCE.T goes beyond `FENCE` and `FENCE.I` as it clears L1 caches, TLB, branch predictors… It is a countermeasure for SPECTRE-like attacks. It is also useful in safety-critical applications to increase execution time predictability.

It is not yet decided if the `FENCE.T` instruction arguments can be used to select a subset of microarchitecture features that will be cleared. The list of arguments, if any, will be detailed in the user's guide.

Anticipation of verification: It can be cumbersome to prove the timing decorrelation as expressed in the requirement with digital simulations. We can simulate the microarchitecture features and explain how they satisfy the requirement as Nils Wistoff's work demonstrated.

2.2.6 PPA targets

These PPA targets will likely be updated when performance monitoring is integrated in the continuous integration flow.

PPA-10	CVA6 should be resource-optimized on FPGA and ASIC targets.
PPA-20	CVA6 should deliver more than 2.1 CoreMark/MHz.
PPA-30	CV32A6 should run at more than 150 MHz in the cv32a6_imac_sv32 configuration on Kintex 7 FPGA technology, commercial -2 speed grade.
PPA-40	CV64A6 should run at more than 900 MHz in the cv64a6_imacfd_sv39 configuration on 28FDSOI technology in the worst case frequency corner with the fastest threshold voltage.
PPA-50	TBD: Placeholder for single-precision floating performance per MHz.
PPA-60	TBD: Placeholder for double-precision floating performance per MHz.

2.2.7 Interface requirements

Memory bus

MEM-10	CVA6 memory interface shall comply with AXI5 specification including the Atomic_Transactions property support as defined in [AXI] section E1.1.
MEM-20	CVA6 AXI memory interface shall feature user bit extensions on the data bus (`WUSER` and `RUSER` as per [AXI]) in connection with the L1I and L1WTD datasize extensions, with a number of user bits greater or equal to 0.

The interface complies with AXI4. However, Atomic_Transactions is only defined in AXI5. For the sake of clarity, we do not use the AXI5-Lite interface.

Debug

DBG-10	CVA6 shall implement both the Abstracted Command and Execution based features outlined in chapter 4 of [RVdbg].

In addition, there can be an external debug module, not in the scope of the IP.

Interrupts

IRQ-10	CVA6 shall implement interrupt handling registers as per the RISC-V privilege specification and interface with a CLINT implementation.

Coprocessor interface

XIF-10	To extend the supported instructions, CVA6 shall have a coprocessor interface that supports the "Issue", "Commit" and "Result" interfaces of the [CV-X-IF] specification.

The goal is to have a compatible interface between CORE-V cores (CVA6, CV32E40X…). The feasibility still needs to be confirmed; including the speculative execution.

CVA6 can interface with several coprocessors simultaneously through a specific external feature implemented on the CV-X-IF interface.

Multi-core interface

TRI-10	CVA6 shall have the Transaction-Response Interface (TRI) needed to interface with the P-Mesh coherence system of OpenPiton, according to [OpenPiton].

2.2.8 Design rules

As different teams have different design rules and to ease the integration in FPGA and ASIC design flows:

RUL-10	CVA6 should have a configurable reset signal: synchronous/asynchronous, active on high or low levels.
RUL-20	CVA6 shall be a super-synchronous design with a single clock input.
RUL-30	CVA6 should not include multi-cycle paths.
RUL-40	CVA6 should not include technology-dependent blocks.

If technology-dependent blocks are used, e.g. to improve PPA on certain targets, the equivalent technology-independent block should be available. Parameters can be used to select between the implementations.

2.2.9 List of abbreviations

ASIC: Application Specific Integrated Circuit
CSR: Control and Status Register
D$: Data cache
DTM: Debug Transport Module
DUT: Design Under Test
DV: Design Verification
ECC: Error Correction Code
EDC: Error Detection Code
FPGA: Field Programmable Gate Array
FPU: Floating Point Unit
I$: Instruction cache
IP: Intellectual Property block
ISA: Instruction Set Architecture
kB: kilo-bytes
L1: Level 1 cache
L1I: Level 1 Instruction cache
L1WTD: Level 1 Write-Through data cache

LFSR: Linear Feedback Shift Register
LRU: Least Recently Used
MMU: Memory Management Unit
OS: Operating System
PF: Project Freeze
PPA: Power Performance Area
PMP: Physical Memory Protection
PTW: Page Table Walk
RW: Read Write
SW: Software
TLB: Translation Lookaside Buffer
TWG: Technical Work Group
WB: Write-Back
WT: Write-Through

2.3 CV32A6 Design Document

2.3.1 Introduction

The OpenHW Group uses semantic versioning to describe the release status of its IP. This document describes v0.1.0 of the CV32A6. This is not intended to be a formal release of CVA6. Currently, the first planned release of CVA6 is the CV32A6 v0.2.0.

CVA6 is a 6-stage in-order and single issue processor core which implements the RISC-V instruction set. CVA6 can be configured as a 32- or 64-bit core (RV32 or RV64), called CV32A6 or CV64A6. This document describes an initial version (v0.1.0) of the CV32A6 processor configuration.

The objective of this document is to provide enough information to allow the RTL modification (by designers) and the RTL verification (by verificators). This document is not dedicated to CVA6 users looking for information to develop software like instructions or registers.

The CVA6 architecture is illustrated in the following figure extracted from a paper written by F.Zaruba and L.Benini.

License

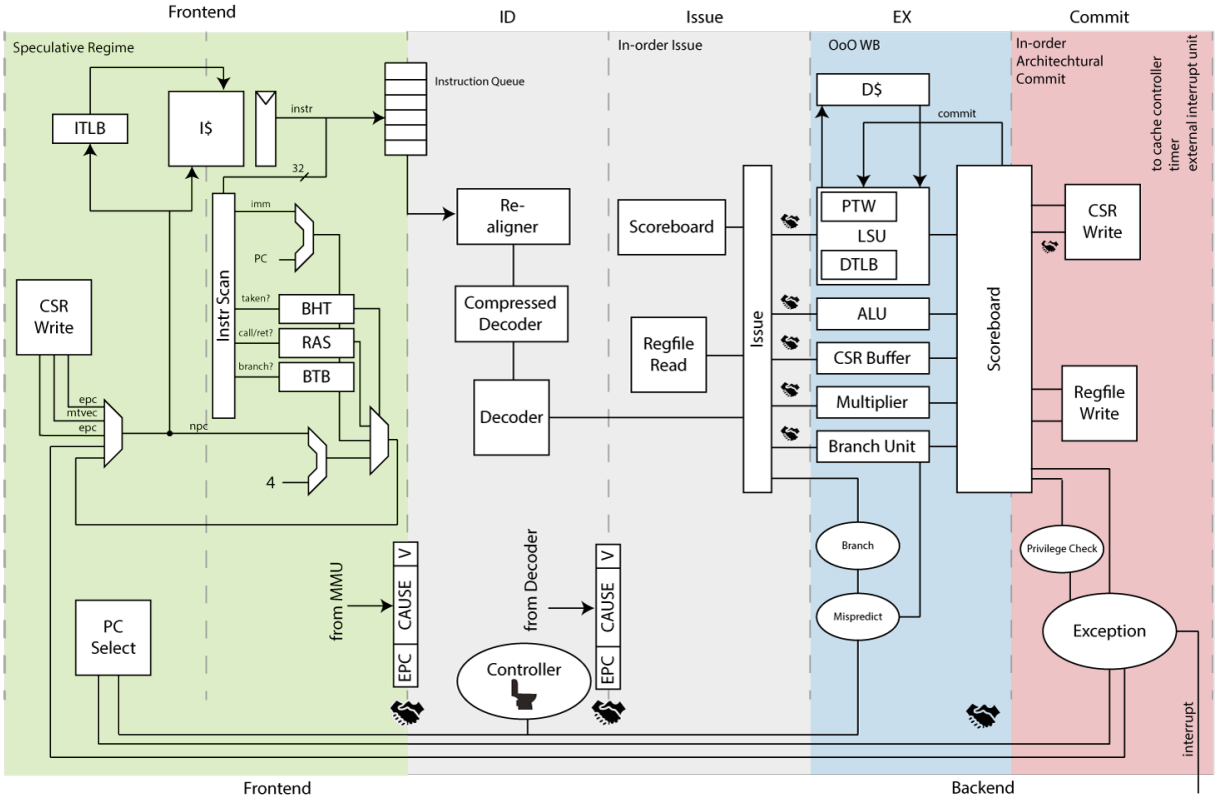


Fig. 1: CVA6 Architecture

Standards Compliance

To ease the reading, the reference to these specifications can be implicit in the requirements below. For the sake of precision, the requirements identify the versions of RISC-V extensions from these specifications.

- **[CVA6req]** "CVA6 requirement specification", https://github.com/openhwgroup/cva6/blob/master/docs/specifications/cva6_requirement_specification.rst, HASH#767c465.
- **[RVunpriv]** "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 13, 2019.
- **[RVpriv]** "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203", Editors Andrew Waterman, Krste Asanović and John Hauser, RISC-V Foundation, December 4, 2021.
- **[RVdbg]** "RISC-V External Debug Support, Document Version 0.13.2", Editors Tim Newsome and Megan Wachs, RISC-V Foundation, March 22, 2019.
- **[RVcompat]** "RISC-V Architectural Compatibility Test Framework", https://github.com/riscv-non-isa/riscv-arch-test.
- **[AXI]** AXI Specification, https://developer.arm.com/documentation/ihi0022/hc.
- **[CV-X-IF]** Placeholder for the CV-X-IF coprocessor interface currently prepared at OpenHW Group; current version in https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/.
- **[OpenPiton]** "OpenPiton Microarchitecture Specification", Princeton University, https://parallel.princeton.edu/openpiton/docs/micro_arch.pdf.

CV32A6 is a standards-compliant 32-bit processor fully compliant with RISC-V specifications: [RVunpriv], [RVpriv] and [RVdbg] and passes [RVcompat] compatibility tests, as requested by [GEN-10] in [CVA6req].

Documentation framework

The framework of this document is inspired by the Common Criteria. The Common Criteria for Information Technology Security Evaluation (referred to as Common Criteria or CC) is an international standard (ISO/IEC 15408) for computer security certification.

Description of the framework:

- Processor is split into module corresponding to the main modules of the design
- Modules can contain several modules
- Each module is described in a chapter, which contains the following subchapters: *Description*, *Functionalities*, *Architecture and Modules* and *Registers* (if any)
- The subchapter *Description* describes the main features of the submodule, the interconnections between the current module and the others and the inputs/outputs interface.
- The subchapter *Functionality* lists in details the module functionalities. Please avoid using the RTL signal names to explain the functionalities.
- The subchapter *Architecture and Modules* provides a drawing to present the module hierarchy, then the functionalities covered by the module
- The subchapter *Registers* specifies the module registers if any

Contributors

Jean-Roch Coulon (jean-roch.coulon@thalesgroup.com)
Ayoub Jalali (ayoub.jalali@external.thalesgroup.com)
Alae Eddine Ezzejjari (alae-eddine.ez-zejjari@external.thalesgroup.com)

[TO BE COMPLETED]

2.3.2 CV32A6 Subsystem

The CV32A6 v0.1.0 is a subsystem composed of the modules and protocol interfaces as illustrated *CV32A6 v0.1.0 modules* The processor is a Harvard-based modern architecture. Instructions are issued in-order through the DECODE stage and executed out-of-order but committed in-order. The processor is Single issue, that means that at maximum one instruction per cycle can be issued to the EXECUTE stage.

The CV32A6 implements a 6-stage pipeline composed of PC Generation, Instruction Detch, Instruction Decode, Issue stage, Execute stage and Commit stage. At least 6 cycles are needed to execute one instruction.

Instantiation

Table 1: CV32A6 v0.1.0 parameterization

Parameter	Type	Value	Description
`ArianeCfg`	ariane_pkg::ariane_cfg_t	ariane_pkg::v0.1.0_Config	CVA6 v0.1.0 configuration

Table 2: CV32A6 v0.1.0 interface signals

Signal	IO	Type	Description
`clk_i`	in	logic	subsystem clock
`rst_ni`	in	logic	Asynchronous reset active low
`boot_addr_i`	in	logic[VLEN-1:0]	Reset boot address
`hart_id_i`	in	logic[XLEN-1:0]	Hart id in a multicore environment (reflected in a CSR)
`irq_i`	in	logic[1:0]	Level sensitive IR lines, mip & sip (async)
`ipi_i`	in	logic	Inter-processor interrupts (async)
`time_irq_i`	in	logic	Timer interrupt in (async)
`debug_req_i`	in	logic	Debug request (async)
`rvfi_o`	out	trace_port_t	RISC-V Formal Interface port (RVFI)
`cvxif_req_o`	out	cvxif_req_t	Coprocessor Interface request interface port (CV-X-IF)
`cvxif_resp_i`	in	cvxif_resp_t	Coprocessor Interface response interface port (CV-X-IF)
`axi_req_o`	out	req_t	AXI master request interface port
`axi_resp_i`	in	resp_t	AXI master response interface port

Functionality

CV32A6 v0.1.0 implements a configuration which allows to connect coprocessor through CV-X-IF coprocessor interface, but the lack of MMU, A extension and data cache prevent from executing Linux.

Table 3: CV32A6 v0.1.0 Standard Configuration

Standard Extension	Specification	Configurability
I: RV32i Base Integer Instruction Set	[RVunpriv]	ON
C: Standard Extension for Compressed Instructions	[RVunpriv]	ON
M: Standard Extension for Integer Multiplication and Division	[RVunpriv]	ON
A: Standard Extension for Atomic transaction	[RVunpriv]	OFF
F and D: Single and Double Precision Floating-Point	[RVunpriv]	OFF
Zicount: Performance Counters	[RVunpriv]	OFF
Zicsr: Control and Status Register Instructions	[RVpriv]	ON
Zifencei: Instruction-Fetch Fence	[RVunpriv]	ON
Privilege: Standard privilege modes M, S and U	[RVpriv]	ON
SV39, SV32, SV0: MMU capability	[RVpriv]	OFF
PMP: Memory Protection Unit	[RVpriv]	OFF
CSR: Control and Status Registers	[RVpriv]	ON
AXI: AXI interface	[CV-X-IF]	ON
TRI: Translation Response Interface (TRI)	[OpenPiton]	OFF

Table 4: CV32A6 v0.1.0 Micro-Architecture Configuration

Micro-architecture	Specification	Configurability
I$: Instruction cache	current spec	ON
D$: Data cache	current spec	OFF
Rename: register Renaming	current spec	OFF
Double Commit: out of order pipeline execute stage	current spec	ON
BP: Branch Prediction	current spec	ON with no info storage

CVA6 memory interface complies with AXI5 specification including the Atomic_Transactions property support as defined in [AXI] section E1.1.

CVA6 coprocessor interface complies with CV-X-IF protocol specification as defined in [CV-X-IF].

The CV32A6 v0.1.0 core is fully synthesizable. It has been designed mainly for ASIC designs, but FPGA synthesis is supported as well.

For ASIC synthesis, the whole design is completely synchronous and uses positive-edge triggered flip-flops. The core occupies an area of about 80 kGE. The clock frequency can be more than 1GHz depending of technology.

Architecture and Modules

The CV32A6 v0.1.0 subsystem is composed of 8 modules.

Connections between modules are illustrated in the following block diagram. FRONTEND, DECODE, ISSUE, EXECUTE, COMMIT and CONTROLLER are part of the pipeline. And CACHES implements the instruction and data caches and CSRFILE contains registers.

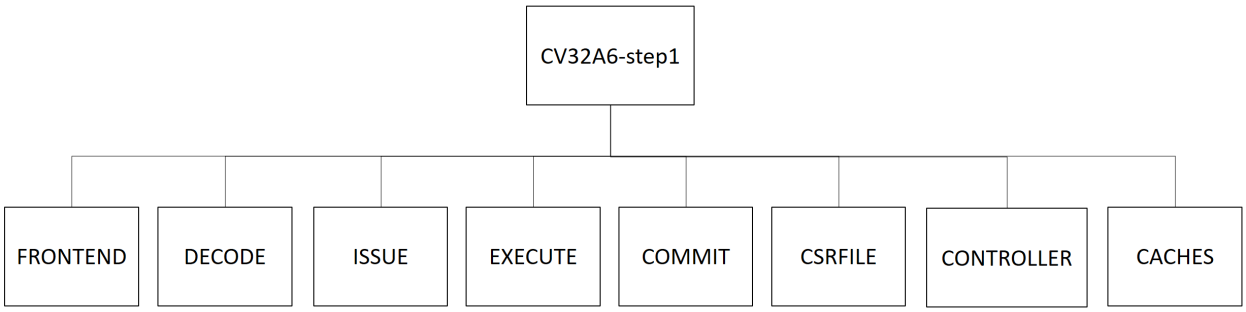


Fig. 2: CV32A6 v0.1.0 modules

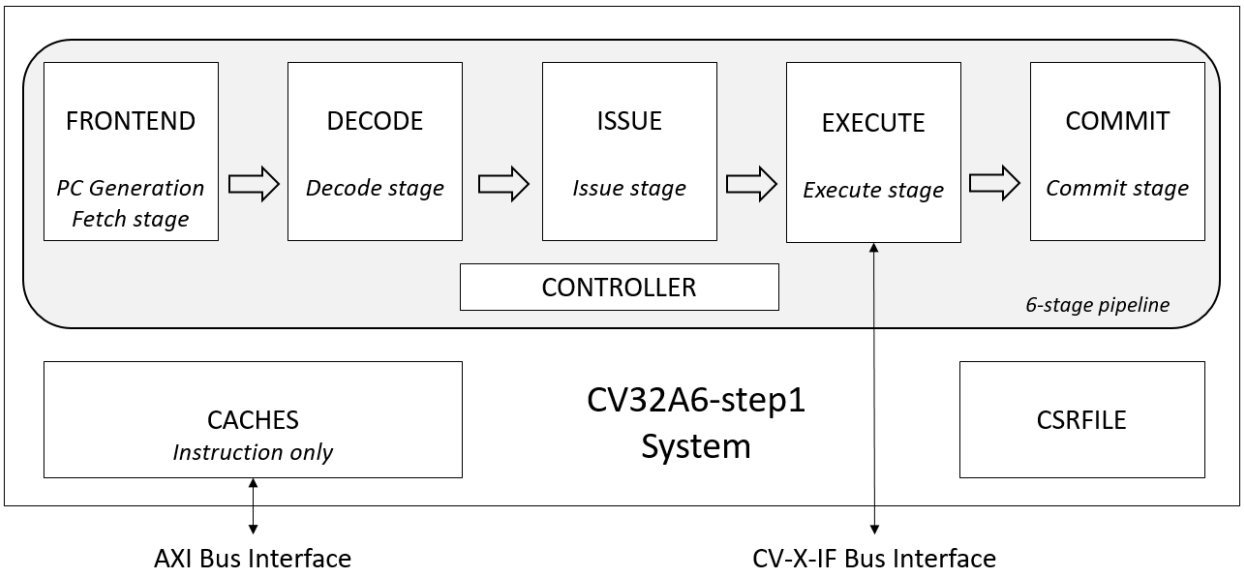


Fig. 3: CV32A6 v0.1.0 pipeline and modules

2.3.3 FRONTEND Module

Description

The FRONTEND module implements two first stages of the cva6 pipeline, PC gen and Fetch stages.

PC gen stage is responsible for generating the next program counter hosting a Branch Target Buffer (BTB) a Branch History Table (BHT) and a Return Address Stack (RAS) to speculate on the branch target address.

Fetch stage requests data to the CACHE module, realigns the data to store them in instruction queue and transmits the instructions to the DECODE module. FRONTEND can fetch up to 2 instructions per cycles when C extension instructions is used, but as instruction queue limits the data rate, up to one instruction per cycle can be sent to DECODE.

The module is connected to:

- CACHES module provides fethed instructions to FRONTEND.
- DECODE module receives instructions from FRONTEND.
- CONTROLLER module can flush FRONTEND PC gen stage
- EXECUTE, CONTROLLER, CSR and COMMIT modules triggers PC jumping due to a branch mispredict, an exception, a return from exception, a debug entry or pipeline flush. They provides related PC next value.
- CSR module states about debug mode.

Table 5: FRONTEND interface signals

Signal	IO	connection	Type	Description
clk_i	in	SUBSYS-TEM	logic	Subsystem Clock
rst_ni	in	SUBSYS-TEM	logic	Asynchronous reset active low
debug_mode_i	in	CSR	logic	Debug mode state
flush_i	in	CON-TROLLER	logic	Fetch flush request
flush_bp_i	in	tied at zero	logic	flush branch prediction
boot_addr_i	in	SUBSYS-TEM	logic[VLEN-1:0]	Next PC when reset
resolved_branch_i	in	EXECUTE	bp_resolve_t	mispredict event and next PC
eret_i	in	CSR	logic	Return from exception event
epc_i	in	CSR	logic[VLEN-1:0]	Next PC when returning from exception
ex_valid_i	in	COMMIT	logic	Exception event
trap_vector_base_i	in	CSR	logic[VLEN-1:0]	Next PC when jumping into exception
set_pc_commit_i	in	CON-TROLLER	logic	Set the PC coming from COMMIT as next PC
pc_commit_i	in	COMMIT	logic[VLEN-1:0]	Next PC when flushing pipeline
set_debug_pc_i	in	CSR	logic	Debug event
icache_dreq_o	out	CACHES	icache_dreq_i_t	Handshake between CACHE and FRONTEND (fetch)
icache_dreq_i	in	CACHES	icache_dreq_o_t	Handshake between CACHE and FRONTEND (fetch)
fetch_entry_o	out	DECODE	fetch_entry_t	Handshake's data between FRONTEND (fetch) and DECODE
fetch_entry_valid_o	out	DECODE	logic	Handshake's valid between FRONTEND (fetch) and DECODE
fetch_entry_ready_i	in	DECODE	logic	Handshake's ready between FRONTEND (fetch) and DECODE

Functionality

PC Generation stage

PC gen generates the next program counter. The next PC can originate from the following sources (listed in order of precedence):

- **Reset state:** At reset, the PC is assigned to the boot address.
- **Branch Predict:** Fetched instruction is predecoded thanks to instr_scan submodule. When instruction is a control flow, three cases need to be considered:
  - 1) If instruction is a JALR and BTB (Branch Target Buffer) returns a valid address, next PC is predicted by BTB. Else JALR is not considered as a control flow instruction, which will generate a mispredict.
  - 2) If instruction is a branch and BTH (Branch History table) returns a valid address, next PC is predicted by BHT. Else branch is not considered as an control flow instruction, which will generate a mispredict when branch is taken.

  - 3) If instruction is a RET and RAS (Return Address Stack) returns a valid address and RET has already been consummed by instruction queue. Else RET is considered as a control flow instruction but next PC is not predicted. A mispredict wil be generated.

  Then the PC gen informs the Fetch stage that it performed a prediction on the PC. *In CV32A6 v0.1.0, Branch Prediction is simplified: no information is stored in BTB, BHT and RAS. JALR, branch and RET instructions are not considered as control flow instruction and will generates mispredict.*

- **Default:** PC + 4 is fetched. PC Gen always fetches on a word boundary (32-bit). Compressed instructions are handled by fetch stage.
- **Mispredict:** When a branch prediction is mispredicted, the EXECUTE feedbacks a misprediction. This can either be a 'real' mis-prediction or a branch which was not recognized as one. In any case we need to correct our action and start fetching from the correct address.
- **Replay instruction fetch:** When the instruction queue is full, the instr_queue submodule asks the fetch replay and provides the address to be replayed.
- **Return from environment call:** When CSR asks a return from an environment call, the PC is assigned to the successive PC to the one stored in the CSR [m-s]epc register.
- **Exception/Interrupt:** If an exception (or interrupt, which is in the context of RISC-V subsystems quite similar) is triggered by the COMMIT, the next PC Gen is assigned to the CSR trap vector base address. The trap vector base address can be different depending on whether the exception traps to S-Mode or M-Mode (user mode exceptions are currently not supported). It is the purpose of the CSR Unit to figure out where to trap to and present the correct address to PC Gen.
- **Pipeline Flush:** When a CSR with side-effects gets written the whole pipeline is flushed by CONTROLLER and FRONTEND starts fetching from the next instruction again in order to take the up-dated information into account (for example virtual memory base pointer changes). The PC related to the flush action is provided by the COMMIT. Moreover flush is also transmitted to the CACHES through the next fetch CACHES access and instruction queue is reset.
- **Debug:** Debug has the highest order of precedence as it can interrupt any control flow requests. It also the only source of control flow change which can actually happen simultaneously to any other of the forced control flow changes. The debug jump is requested by CSR. The address to be jumped into is HW coded. This debug feature is not supported by CV32A6 v0.1.0.

All program counters are logical addressed. If the logical to physical mapping changes a fence.vm instruction should used to flush the pipeline *and TLBs (MMU is not enabled in CV32A6 v0.1.0).*

Fetch Stage

Fetch stage controls by handshake protocol the CACHE module. Fetched data are 32-bit block with word aligned address. A granted fetch is realigned into instr_realign submodule to produce instructions. Then instructions are pushed into an internal instruction FIFO called instruction queue (instr_queue submodule). This submodule stores the instructions and related information which allow to identify the outstanding transactions. In the case CONTROLLER decides to flush the instruction queue, the outstanding transactions are discarded.

The Fetch stage asks the MMU (MMU is not enabled in CV32A6 v0.1.0) to translate the requested address.

Memory *and MMU (MMU is not enabled in CV32A6 v0.1.0)* can feedback potential exceptions generated by the memory fetch request. They can be bus errors, invalid accesses or instruction page faults.

Architecture and Submodules

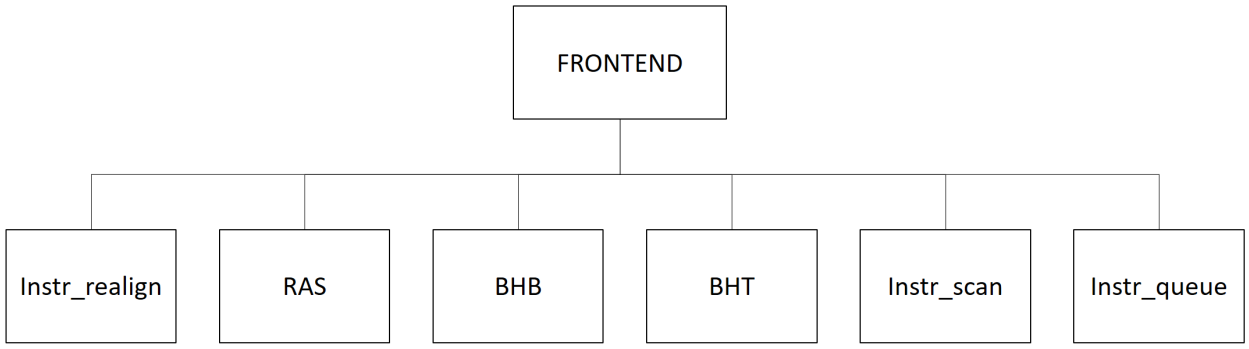


Fig. 4: FRONTEND submodules

Instr_realign submodule

Table 6: instr_realign interface signals

Signal	IO	connection	Type	Description
clk_i	in	SUBSYSTEM	logic	Subystem Clock
rst_ni	in	SUBSYSTEM	logic	Asynchronous reset active low
flush_i	in	FRONTEND	logic	Instr_align Flush
valid_i	in	CACHES (reg)	logic	32-bit block is valid
address_i	in	CACHES (reg)	logic[VLEN-1:0]	32-bit block address
data_i	in	CACHES (reg)	logic[31:0]	32-bit block
valid_o	out	FRONTEND	logic[1:0]	instruction is valid
addr_o	out	FRONTEND	logic[1:0][VLEN-1:0]	Instruction address
instr_o	out	instr_scan, instr_queue	logic[1:0][31:0]	Instruction
serving_unaligned_o	out	FRONTEND	logic	Instruction is unaligned

The 32-bit aligned block coming from the CACHE module enters the instr_realign submodule. This submodule extracts the instructions from the 32-bit blocks, up to two instructions because it is possible to fetch two instructions when C extension is used. If the instructions are not compressed, it is possible that the instruction is not aligned on the block size but rather interleaved with two cache blocks. In that case, two cache accesses are needed. The instr_realign submodule provides at maximum one instruction per cycle. Not complete instruction is stored in instr_realign submodule before being provided in the next cycles.

In case of mispredict, flush, replay or branch predict, the instr_realign is re-initialized, the internal register storing the instruction alignment state is reset.

Instr_queue submodule

Table 7: instr_realign interface signals

Signal	IO	connection	Type	Description
clk_i	in	SUBSYSTEM	logic	Subystem Clock
rst_ni	in	SUBSYSTEM	logic	Asynchronous reset active low
flush_i	in	CONTROLLER	logic	Fetch flush request
valid_i	in	instr_realign	logic[1:0]	Instruction is valid
instr_i	in	instr_realign	logic[1:0][31:0]	Instruction
addr_i	in	instr_realign	logic[1:0][VLEN-1:0]	Instruction address
predict_address_i	in	FRONTEND	logic[VLEN-1:0]	Instruction predict address
cf_type_i	in	FRONTEND	logic[1:0]	Instruction control flow type
ready_o	out	CACHES	logic	Handshake's ready between CACHE and FRONTEND (fetch stage)
consumed_o	out	FRONTEND	logic[1:0]	Indicates instructions consummed, that is to say popped by DECODE
exception_i	in	CACHES (reg)	logic	Exception
exception_addr_i	in	CACHES (reg)	logic[VLEN-1:0]	Exception address
replay_o	out	FRONTEND	logic	Replay instruction because one of the FIFO was already full
replay_addr_o	out	FRONTEND	logic[VLEN-1:0]	Address at which to replay the fetch
fetch_entry_o	out	DECODE	fetch_entry_t	Handshake's data between FRONTEND (fetch stage) and DECODE
fetch_entry_valid_o	out	DECODE	logic	Handshake's valid between FRONTEND (fetch stage) and DECODE
fetch_entry_ready_i	in	DECODE	logic	Handshake's ready between FRONTEND (fetch stage) and DECODE

The instr_queue receives 32bit block from CACHES to create a valid stream of instructions to be decoded (by DECODE), to be issued (by ISSUE) and executed (by EXECUTE). FRONTEND pushes in FIFO to store the instructions and related information needed in case of mispredict or exception: instructions, instruction control flow type, exception, exception address and predicted address. DECODE pops them when decode stage is ready and indicates to the FRONTEND the instruction has been consummed.

The instruction queue contains max 4 instructions.

In instruction queue, exception can only correspond to page-fault exception.

If the instruction queue is full, a replay request is sent to inform the fetch mechanism to replay the fetch.

The instruction queue can be flushed by CONTROLLER.

Instr_scan submodule

Table 8: instr_scan interface signals

Signal	IO	Connection	Type	Description
instr_i	in	instr_realign	logic[31:0]	Instruction to be predecoded
rvi_return_o	out	FRONTEND	logic	Return instruction
rvi_call_o	out	FRONTEND	logic	JAL instruction
rvi_branch_o	out	FRONTEND	logic	Branch instruction
rvi_jalr_o	out	FRONTEND	logic	JALR instruction
rvi_jump_o	out	FRONTEND	logic	unconditional jump instruction
rvi_imm_o	out	FRONTEND	logic[VLEN-1:0]	Instruction immediat
rvc_branch_o	out	FRONTEND	logic	Branch compressed instruction
rvc_jump_o	out	FRONTEND	logic	unconditional jump compressed instruction
rvc_jr_o	out	FRONTEND	logic	JR compressed instruction
rvc_return_o	out	FRONTEND	logic	Return compressed instruction
rvc_jalr_o	out	FRONTEND	logic	JALR compressed instruction
rvc_call_o	out	FRONTEND	logic	JAL compressed instruction
rvc_imm_o	out	FRONTEND	logic[VLEN-1:0]	Instruction compressed immediat

The instr_scan submodule pre-decodes the fetched instructions, instructions could be compressed or not. The outputs are used by the branch prediction feature. The instr_scan submodule tells if the instruction is compressed and provides the intruction type: branch, jump, return, jalr, imm, call or others.

BHT (Branch History Table) submodule

Table 9: BHT interface signals

Signal	IO	Connection	Type	Description
clk_i	in	SUBSYSTEM	logic	Subystem clock
rst_ni	in	SUBSYSTEM	logic	Asynchronous reset active low
flush_i	in	tied at zero	logic	Flush request
debug_mode_i	in	CSR	logic	Debug mode state
vpc_i	in	CACHES (reg)	logic[VLEN-1:0]	Virtual PC
bht_update_i	in	EXECUTE	bht_update_t	Update btb with resolved address
bht_prediction_o	out	FRONTEND	bht_prediction_t	Prediction from bht

When a branch instruction is resolved by the EXECUTE, the relative information is stored in the Branch History Table.

The information is stored in a 1024 entry table.

The Branch History table is a two-bit saturation counter that takes the virtual address of the current fetched instruction by the CACHE. It states whether the current branch request should be taken or not. The two bit counter is updated by the successive execution of the current instructions as shown in the following figure.

The BHT is not updated if processor is in debug mode.

When a branch instruction is pre-decoded by instr_scan submodule, the BHT informs whether the PC address is in the BHT. In this case, the BHT predicts whether the branch is taken and provides the corresponding target address.

The BTB is never flushed.

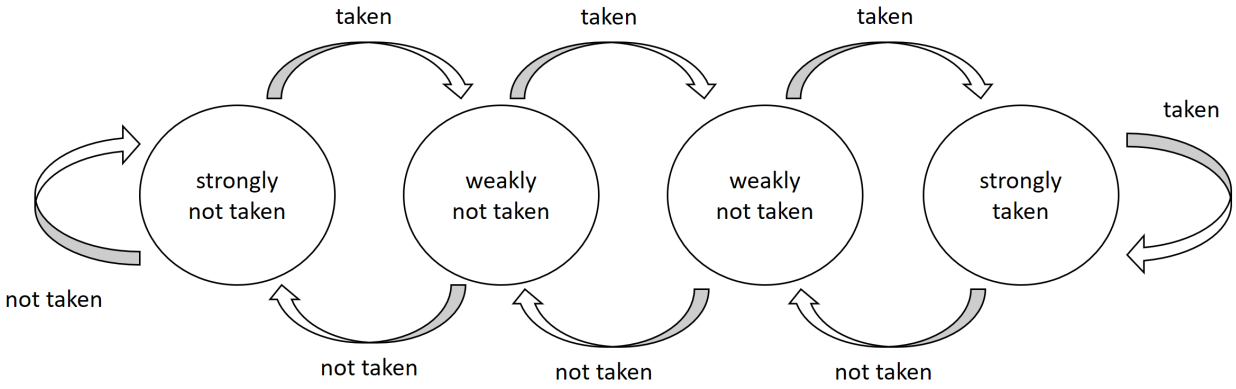


Fig. 5: BHT saturation

BTB (Branch Target Buffer) submodule

Table 10: BTB interface signals

Signal	IO	Connection	Type	Description
clk_i	in	SUBSYSTEM	logic	Subystem clock
rst_ni	in	SUBSYSTEM	logic	Asynchronous reset active low
flush_i	in	tied at zero	logic	Flush request state
debug_mode_i	in	CSR	logic	Debug mode
vpc_i	in	CACHES (reg)	logic	Virtual PC
btb_update_i	in	EXECUTE	btb_update_t	Update BTB with resolved address
btb_prediction_o	out	FRONTEND	btb_prediction_t	BTB Prediction

When a unconditional jumps to a register (JALR instruction) is mispredicted by the EXECUTE, the relative information is stored into the BTB, that is to say the JALR PC and the target address.

The information is stored in a 8 entry table.

The BTB is not updated if processor is in debug mode.

When a branch instruction is pre-decoded by instr_scan submodule, the BTB informs whether the input PC address is in BTB. In this case, the BTB provides the corresponding target address.

The BTB is never flushed.

RAS (Return Address Stack) submodule

Table 11: RAS interface signals

Signal	IO	Connection	Type	Description
clk_i	in	SUBSYSTEM	logic	Subystem clock
rst_ni	in	SUBSYSTEM	logic	Asynchronous reset active low
flush_i	in	tied at zero	logic	Flush request
push_i	in	FRONTEND	logic	Push address in RAS
pop_i	in	FRONTEND	logic	Pop address from RAS
data_i	in	FRONTEND	logic[VLEN-1:0]	Data to be pushed
data_o	out	FRONTEND	ras_t	Popped data

When an unconditional jumps to a known target address (JAL instruction) is consummed by the instr_queue, the next pc after the JAL instruction and the return address are stored into a FIFO.

The RAS FIFO depth is 2.

When a branch instruction is pre-decoded by instr_scan submodule, the RAS informs whether the input PC address is in RAS. In this case, the RAS provides the corresponding target address.

The RAS is never flushed.

2.3.4 RV32 Instructions

Introduction

In this document, we present ISA (Instruction Set Architecture) for C32VA6_v0.1.0, illustrating different supported instructions, the Base Integer Instruction set RV32I, and also other instructions in some extensions supported by the core as:

- RV32M – Standard Extension for Integer Multiplication and Division Instructions
- RV32A – Standard Extension for Atomic Instructions
- RV32C – Standard Extension for Compressed Instructions
- RV32Zicsr – Standard Extension for CSR Instructions
- RV32Zifencei – Standard Extension for Instruction-Fetch Fence

The base RISC-V ISA has fixed-length 32-bit instructions or 16-bit instructions (the C32VA6_v0.1.0 support C extension), so that must be naturally aligned on 4-byte boundary or 2-byte boundary. The C32VA6_v0.1.0 supports:

- Only 1 hart,
- Misaligned accesses to the memory.

General purpose registers

As shown in the Table 1.1, There are 31 general-purpose registers x1–x31, which hold integer values. Register x0 is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register x1 to hold the return address on a call. For C32VA6_v0.1.0, the x registers are 32 bits wide. There is one additional register also 32 bits wide: the program counter pc holds the address of the current instruction.

Table 1.1 shows the general-purpose registers :

5-bit Encoding (rx)	3-bit Compressed Encoding (rx')	Register (ISA name)	Register (ABI name)	Description
0		x0	zero	Hardwired zero
1		x1	ra	Return address
2		x2	sp	Stack pointer
3		x3	gp	Global pointer
4		x4	tp	Thread pointer
5		x5	t0	Temporaries/alternate link register
6 - 7		x6 - x7	t1 - t2	Temporaries
8	0	x8	s0/fp	Saved register/frame pointer
9	1	x9	s1	Saved registers
10 - 11	2 - 3	x10 - x11	a0 - a1	Function arguments/return value
12 - 15	4 - 7	x12 - x15	a2 - a5	Function arguments
16 - 17		x16 - x17	a6 - a7	Function arguments
18 - 27		x18 - x27	s2 - s11	Saved registers
28 - 31		x28 - x31	t3 - t6	Temporaries

RV32I Base Integer Instruction Set

This section describes the RV32I base integer instruction set.

Integer Register-Immediate Instructions

- **ADDI**: Add Immediate

  Format: addi rd, rs1, imm[11:0]

  Description: add sign-extended 12-bit immediate to register rs1, and store the result in register rd.

  Pseudocode: x[rd] = x[rs1] + sext(imm[11:0])

  Invalid values: NONE

  Exception raised: NONE

- **ANDI**: AND Immediate

  Format: andi rd, rs1, imm[11:0]

  Description: perform bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

  Pseudocode: x[rd] = x[rs1] & sext(imm[11:0])

  Invalid values: NONE

  Exception raised: NONE

- **ORI**: OR Immediate

  Format: ori rd, rs1, imm[11:0]

  Description: perform bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

Pseudocode: x[rd] = x[rs1] | sext(imm[11:0])

Invalid values: NONE

Exception raised: NONE

- **XORI**: XOR Immediate

  Format: xori rd, rs1, imm[11:0]

  Description: perform bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

  Pseudocode: x[rd] = x[rs1] ^ sext(imm[11:0])

  Invalid values: NONE

  Exception raised: NONE

- **SLTI**: Set Less Then Immediate

  Format: slti rd, rs1, imm[11:0]

  Description: set register rd to 1 if register rs1 is less than the sign extended immediate when both are treated as signed numbers, else 0 is written to rd.

  Pseudocode: if (x[rs1] < sext(imm[11:0]) x[rd] = 1 else x[rd] = 0

  Invalid values: NONE

  Exception raised: NONE

- **SLTIU**: Set Less Then Immediate Unsigned

  Format: sltiu rd, rs1, imm[11:0]

  Description: set register rd to 1 if register rs1 is less than the sign extended immediate when both are treated as unsigned numbers, else 0 is written to rd.

  Pseudocode: if (x[rs1] <u sext(imm[11:0]) x[rd] = 1 else x[rd] = 0

  Invalid values: NONE

  Exception raised: NONE

- **SLLI**: Shift Left Logic Immediate

  Format: slli rd, rs1, imm[4:0]

  Description: logical left shift (zeros are shifted into the lower bits).

  Pseudocode: x[rd] = x[rs1] << imm[4:0]

  Invalid values: NONE

  Exception raised: NONE

- **SRLI**: Shift Right Logic Immediate

  Format: srli rd, rs1, imm[4:0]

  Description: logical right shift (zeros are shifted into the upper bits).

  Pseudocode: x[rd] = x[rs1] >> imm[4:0]

  Invalid values: NONE

  Exception raised: NONE

- **SRAI**: Shift Right Arithmetic Immediate

Format: srai rd, rs1, imm[4:0]

Description: arithmetic right shift (the original sign bit is copied into the vacated upper bits).

Pseudocode: x[rd] = x[rs1] >>s imm[4:0]

Invalid values: NONE

Exception raised: NONE

- **LUI**: Load Upper Immediate

  Format: lui rd, imm[19:0]

  Description: place the immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.

  Pseudocode: x[rd] = sext(imm[31:12] << 12)

  Invalid values: NONE

  Exception raised: NONE

- **AUIPC**: Add Upper Immediate to PC

  Format: auipc rd, imm[19:0]

  Description: form a 32-bit offset from the 20-bit immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then place the result in register rd.

  Pseudocode: x[rd] = pc + sext(immediate[31:12] << 12)

  Invalid values: NONE

  Exception raised: NONE

Integer Register-Register Instructions

- **ADD**: Addition

  Format: add rd, rs1, rs2

  Description: add rs2 to register rs1, and store the result in register rd.

  Pseudocode: x[rd] = x[rs1] + x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **SUB**: Subtraction

  Format: sub rd, rs1, rs2

  Description: subtract rs2 from register rs1, and store the result in register rd.

  Pseudocode: x[rd] = x[rs1] - x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **AND**: AND logical operator

  Format: and rd, rs1, rs2

  Description: perform bitwise AND on register rs1 and rs2 and place the result in rd.

Pseudocode: x[rd] = x[rs1] & x[rs2]

Invalid values: NONE

Exception raised: NONE

- **OR**: OR logical operator

  Format: or rd, rs1, rs2

  Description: perform bitwise OR on register rs1 and rs2 and place the result in rd.

  Pseudocode: x[rd] = x[rs1] | x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **XOR**: XOR logical operator

  Format: xor rd, rs1, rs2

  Description: perform bitwise XOR on register rs1 and rs2 and place the result in rd.

  Pseudocode: x[rd] = x[rs1] ^ x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **SLT**: Set Less Then

  Format: slt rd, rs1, rs2

  Description: set register rd to 1 if register rs1 is less than rs2 when both are treated as signed numbers, else 0 is written to rd.

  Pseudocode: if (x[rs1] < x[rs2]) x[rd] = 1 else x[rd] = 0

  Invalid values: NONE

  Exception raised: NONE

- **SLTU**: Set Less Then Unsigned

  Format: sltu rd, rs1, rs2

  Description: set register rd to 1 if register rs1 is less than rs2 when both are treated as unsigned numbers, else 0 is written to rd.

  Pseudocode: if (x[rs1] <u x[rs2]) x[rd] = 1 else x[rd] = 0

  Invalid values: NONE

  Exception raised: NONE

- **SLL**: Shift Left Logic

  Format: sll rd, rs1, rs2

  Description: logical left shift (zeros are shifted into the lower bits).

  Pseudocode: x[rd] = x[rs1] << x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **SRL**: Shift Right Logic

Format: srl rd, rs1, rs2

Description: logical right shift (zeros are shifted into the upper bits).

Pseudocode: x[rd] = x[rs1] >> x[rs2]

Invalid values: NONE

Exception raised: NONE

- **SRA**: Shift Right Arithmetic

  Format: sra rd, rs1, rs2

  Description: arithmetic right shift (the original sign bit is copied into the vacated upper bits).

  Pseudocode: x[rd] = x[rs1] >>s x[rs2]

  Invalid values: NONE

  Exception raised: NONE

Control Transfer Instructions

Unconditional Jumps

- **JAL**: Jump and Link

  Format: jal rd, imm[20:1]

  Description: offset is sign-extended and added to the pc to form the jump target address (pc is calculated using signed arithmetic), then setting the least-significant bit of the result to zero, and store the address of instruction following the jump (pc+4) into register rd.

  Pseudocode: x[rd] = pc+4; pc += sext(imm[20:1])

  Invalid values: NONE

  Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception. An exception is raised on taken branch or unconditional jump if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.

- **JALR**: Jump and Link Register

  Format: jalr rd, rs1, imm[11:0]

  Description: target address is obtained by adding the 12-bit signed immediate to the register rs1 (pc is calculated using signed arithmetic), then setting the least-significant bit of the result to zero, and store the address of instruction following the jump (pc+4) into register rd.

  Pseudocode: t = pc+4; pc = (x[rs1]+sext(imm[11:0]))&1 ; x[rd] = t

  Invalid values: NONE

  Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception. An exception is raised on taken branch or unconditional jump if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.

Conditional Branches

- **BEQ**: Branch Equal

  Format: beq rs1, rs2, imm[12:1]

  Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 and rs2 are equal.

Invalid values: NONE

Pseudocode: if (x[rs1] == x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **BNE**: Branch Not Equal

  Format: bne rs1, rs2, imm[12:1]

  Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 and rs2 are not equal.

  Invalid values: NONE

  Pseudocode: if (x[rs1] != x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

  Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **BLT**: Branch Less Than

  Format: blt rs1, rs2, imm[12:1]

  Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 less than rs2 (using signed comparison).

  Invalid values: NONE

  Pseudocode: if (x[rs1] < x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

  Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **BLTU**: Branch Less Than Unsigned

  Format: bltu rs1, rs2, imm[12:1]

  Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 less than rs2 (using unsigned comparison).

  Invalid values: NONE

  Pseudocode: if (x[rs1] <u x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

  Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **BGE**: Branch Greater or Equal

  Format: bge rs1, rs2, imm[12:1]

  Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 is greater than or equal rs2 (using signed comparison).

  Pseudocode: if (x[rs1] >= x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

  Invalid values: NONE

  Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **BGEU**: Branch Greater or Equal Unsigned

Format: bgeu rs1, rs2, imm[12:1]

Description: takes the branch (pc is calculated using signed arithmetic) if registers rs1 is greater than or equal rs2 (using unsigned comparison).

Pseudocode: if (x[rs1] >=u x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4

Exception raised: no instruction fetch misaligned exception is generated for a conditional branch that is not taken.

Load and Store Instructions

- **LB**: Load Byte

  Format: lb rd, imm(rs1)

  Description: loads a 8-bit value from memory, then sign-extends to 32-bit before storing in rd (rd is calculated using signed arithmetic), the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: x[rd] = sext(M[x[rs1] + sext(imm[11:0])][7:0])

  Invalid values: NONE

  Exception raised: loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.

- **LH**: Load Halfword

  Format: lh rd, imm(rs1)

  Description: loads a 16-bit value from memory, then sign-extends to 32-bit before storing in rd (rd is calculated using signed arithmetic), the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: x[rd] = sext(M[x[rs1] + sext(imm[11:0])][15:0])

  Invalid values: NONE

  Exception raised: loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded, also an exception is raised if the memory address isn't aligned (2-byte boundary).

- **LW**: Load Word

  Format: lw rd, imm(rs1)

  Description: loads a 32-bit value from memory, then storing in rd (rd is calculated using signed arithmetic). The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Invalid values: NONE

  Pseudocode: x[rd] = sext(M[x[rs1] + sext(imm[11:0])][31:0])

  Exception raised: loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded, also an exception is raised if the memory address isn't aligned (4-byte boundary).

- **LBU**: Load Byte Unsigned

  Format: lbu rd, imm(rs1)

Description: loads a 8-bit value from memory, then zero-extends to 32-bit before storing in rd (rd is calculated using unsigned arithmetic), the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

Pseudocode: x[rd] = zext(M[x[rs1] + sext(imm[11:0])][7:0])

Invalid values: NONE

Exception raised: loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.

- **LHU**: Load Halfword Unsigned

  Format: lhu rd, imm(rs1)

  Description: loads a 16-bit value from memory, then zero-extends to 32-bit before storing in rd (rd is calculated using unsigned arithmetic), the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: x[rd] = zext(M[x[rs1] + sext(imm[11:0])][15:0])

  Invalid values: NONE

  Exception raised: loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded, also an exception is raised if the memory address isn't aligned (2-byte boundary).

- **SB**: Store Byte

  Format: sb rs2, imm(rs1)

  Description: stores a 8-bit value from the low bits of register rs2 to memory, the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: M[x[rs1] + sext(imm[11:0])][7:0] = x[rs2][7:0]

  Invalid values: NONE

  Exception raised: NONE

- **SH**: Store Halfword

  Format: sh rs2, imm(rs1)

  Description: stores a 16-bit value from the low bits of register rs2 to memory, the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: M[x[rs1] + sext(imm[11:0])][15:0] = x[rs2][15:0]

  Invalid values: NONE

  Exception raised: an exception is raised if the memory address isn't aligned (2-byte boundary).

- **SW**: Store Word

  Format: sw rs2, imm(rs1)

  Description: stores a 32-bit value from register rs2 to memory, the effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

  Pseudocode: M[x[rs1] + sext(imm[11:0])][31:0] = x[rs2][31:0]

  Invalid values: NONE

  Exception raised: an exception is raised if the memory address isn't aligned (4-byte boundary).

Memory Ordering

- **FENCE**: Fence Instruction

  Format: fence pre, succ

  Description: order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE, as the core support 1 hart, the fence instruction has no effect so we can considerate it as a nop instruction.

  Pseudocode: No operation (nop)

  Invalid values: NONE

  Exception raised: NONE

Environment Call and Breakpoints

- **ECALL**: Environment Call

  Format: ecall

  Description: make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

  Pseudocode: RaiseException(EnvironmentCall)

  Invalid values: NONE

  Exception raised: Raise an Environment Call exception.

- **EBREAK**:Environment Break

  Format: ebreak

  Description: cause control to be transferred back to a debugging environment.

  Pseudocode: RaiseException(Breakpoint)

  Invalid values: NONE

  Exception raised: Raise a Breakpoint exception.

RV32M Multiplication and Division Instructions

This chapter describes the standard integer multiplication and division instruction extension, which is named “M” and contains instructions that multiply or divide values held in two integer registers.

Multiplication Operations

- **MUL**: Multiplication

  Format: mul rd, rs1, rs2

  Description: performs a 32-bit × 32-bit multiplication and places the lower 32 bits in the destination register (Both rs1 and rs2 treated as signed numbers).

  Pseudocode: x[rd] = x[rs1] * x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **MULH**: Multiplication Higher

  Format: mulh rd, rs1, rs2

  Description: performs a 32-bit × 32-bit multiplication and places the upper 32 bits in the destination register of the 64-bit product (Both rs1 and rs2 treated as signed numbers).

  Pseudocode: x[rd] = (x[rs1] s*s x[rs2]) >>s 32

  Invalid values: NONE

  Exception raised: NONE

- **MULHU**: Multiplication Higher Unsigned

  Format: mulhu rd, rs1, rs2

  Description: performs a 32-bit × 32-bit multiplication and places the upper 32 bits in the destination register of the 64-bit product (Both rs1 and rs2 treated as unsigned numbers).

  Pseudocode: x[rd] = (x[rs1] u*u x[rs2]) >>u 32

  Invalid values: NONE

  Exception raised: NONE

- **MULHSU**: Multiplication Higher Signed Unsigned

  Format: mulhsu rd, rs1, rs2

  Description: performs a 32-bit × 32-bit multiplication and places the upper 32 bits in the destination register of the 64-bit product (rs1 treated as signed number, rs2 treated as unsigned number).

  Pseudocode: x[rd] = (x[rs1] s*u x[rs2]) >>s 32

  Invalid values: NONE

  Exception raised: NONE

Division Operations

- **DIV**: Division

  Format: div rd, rs1, rs2

  Description: perform signed integer division of 32 bits by 32 bits (rounding towards zero).

  Pseudocode: x[rd] = x[rs1] /s x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **DIVU**: Division Unsigned

  Format: divu rd, rs1, rs2

  Description: perform unsigned integer division of 32 bits by 32 bits (rounding towards zero).

  Pseudocode: x[rd] = x[rs1] /u x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **REM**: Remain

  Format: rem rd, rs1, rs2

  Description: provide the remainder of the corresponding division operation DIV (the sign of rd equals the sign of rs1).

  Pseudocode: x[rd] = x[rs1] %s x[rs2]

  Invalid values: NONE

  Exception raised: NONE

- **REMU**: Remain Unsigned

  Format: rem rd, rs1, rs2

  Description: provide the remainder of the corresponding division operation DIVU.

  Pseudocode: x[rd] = x[rs1] %u x[rs2]

  Invalid values: NONE

  Exception raised: NONE

RV32A Atomic Instructions

The standard atomic instruction extension is denoted by instruction subset name "A", and contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics.

Load-Reserved/Store-Conditional Instructions

- **LR.W**: Load-Reserved Word

  Format: lr.w rd, (rs1)

  Description: LR loads a word from the address in rs1, places the sign-extended value in rd, and registers a reservation on the memory address.

  Pseudocode: x[rd] = LoadReserved32(M[x[rs1]])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **LR.W**: Store-Conditional Word

Format: sc.w rd, rs2, (rs1)

Description: SC writes a word in rs2 to the address in rs1, provided a valid reservation still exists on that address. SC writes zero to rd on success or a nonzero code on failure.

Pseudocode: x[rd] = StoreConditional32(M[x[rs1]], x[rs2])

Invalid values: NONE

Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

Atomic Memory Operations

- **AMOADD.W**: Atomic Memory Operation: Add Word

  Format: amoadd.w rd, rs2, (rs1)

  Description: AMOADD.W atomically loads a data value from the address in rs1, places the value into register rd, then adds the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] + x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOAND.W**: Atomic Memory Operation: And Word

  Format: amoand.w rd, rs2, (rs1)

  Description: AMOAND.W atomically loads a data value from the address in rs1, places the value into register rd, then performs an AND between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] & x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOOR.W**: Atomic Memory Operation: Or Word

  Format: amoor.w rd, rs2, (rs1)

  Description: AMOOR.W atomically loads a data value from the address in rs1, places the value into register rd, then performs an OR between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] | x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOXOR.W**: Atomic Memory Operation: Xor Word

  Format: amoxor.w rd, rs2, (rs1)

Description: AMOXOR.W atomically loads a data value from the address in rs1, places the value into register rd, then performs a XOR between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

Pseudocode: x[rd] = AMO32(M[x[rs1]] ^ x[rs2])

Invalid values: NONE

Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOSWAP.W**: Atomic Memory Operation: Swap Word

  Format: amoswap.w rd, rs2, (rs1)

  Description: AMOSWAP.W atomically loads a data value from the address in rs1, places the value into register rd, then performs a SWAP between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] SWAP x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOMIN.W**: Atomic Memory Operation: Minimum Word

  Format: amomin.d rd, rs2, (rs1)

  Description: AMOMIN.W atomically loads a data value from the address in rs1, places the value into register rd, then choses the minimum between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] MIN x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOMINU.W**: Atomic Memory Operation: Minimum Word, Unsigned

  Format: amominu.d rd, rs2, (rs1)

  Description: AMOMINU.W atomically loads a data value from the address in rs1, places the value into register rd, then choses the minimum (the values treated as unsigned) between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] MINU x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOMAX.W**: Atomic Memory Operation: Maximum Word, Unsigned

  Format: amomax.d rd, rs2, (rs1)

  Description: AMOMAX.W atomically loads a data value from the address in rs1, places the value into register rd, then choses the maximum between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] MAX x[rs2])

  Invalid values: NONE

Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

- **AMOMAXU.W**: Atomic Memory Operation: Maximum Word, Unsigned

  Format: amomaxu.d rd, rs2, (rs1)

  Description: AMOMAXU.W atomically loads a data value from the address in rs1, places the value into register rd, then choses the maximum (the values treated as unsigned) between the loaded value and the original value in rs2, then stores the result back to the address in rs1.

  Pseudocode: x[rd] = AMO32(M[x[rs1]] MAXU x[rs2])

  Invalid values: NONE

  Exception raised: If the address is not naturally aligned (4-byte boundary), a misaligned address exception will be generated.

RV32C Compressed Instructions

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small;
- one of the registers is the zero register (x0), the ABI link register (x1), or the ABI stack pointer (x2);
- the destination register and the first source register are identical;
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary. With the addition of the C extension, JAL and JALR instructions will no longer raise an instruction misaligned exception.

Integer Computational Instructions

- **C.LI**: Compressed Load Immediate

  Format: c.li rd, imm[5:0]

  Description: loads the sign-extended 6-bit immediate, imm, into register rd.

  Pseudocode: x[rd] = sext(imm[5:0])

  Invalid values: rd = x0

  Exception raised: NONE

- **C.LUI**: Compressed Load Upper Immediate

  Format: c.lui rd, nzimm[17:12]

  Description: loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination.

  Pseudocode: x[rd] = sext(nzimm[17:12] << 12)

  Invalid values: rd = x0 & rd = x2 & nzimm = 0

  Exception raised: NONE

- **C.ADDI**: Compressed Addition Immediate

Format: c.addi rd, nzimm[5:0]

Description: adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd.

Pseudocode: x[rd] = x[rd] + sext(nzimm[5:0])

Invalid values: rd = x0 & nzimm = 0

Exception raised: NONE

- **C.ADDI16SP**: Addition Immediate Scaled by 16, to Stack Pointer

  Format: c.addi16sp nzimm[9:4]

  Description: adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. C.ADDI16SP shares the opcode with C.LUI, but has a destination field of x2.

  Pseudocode: x[2] = x[2] + sext(nzimm[9:4])

  Invalid values: rd != x2 & nzimm = 0

  Exception raised: NONE

- **C.ADDI4SPN**: Addition Immediate Scaled by 4, to Stack Pointer

  Format: c.addi4spn nzimm[9:2]

  Description: adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. This instruction is used to generate pointers to stack-allocated variables.

  Pseudocode: x[8 + rd'] = x[2] + zext(nzimm[9:2])

  Invalid values: nzimm = 0

  Exception raised: NONE

- **C.SLLI**: Compressed Shift Left Logic Immediate

  Format: c.slli rd, uimm[5:0]

  Description: performs a logical left shift (zeros are shifted into the lower bits).

  Pseudocode: x[rd] = x[rd] << uimm[5:0]

  Invalid values: rd = x0 & uimm[5] = 0

  Exception raised: NONE

- **C.SRLI**: Compressed Shift Right Logic Immediate

  Format: c.srli rd', uimm[5:0]

  Description: performs a logical right shift (zeros are shifted into the upper bits).

  Pseudocode: x[8 + rd'] = x[8 + rd'] >> uimm[5:0]

  Invalid values: uimm[5] = 0

  Exception raised: NONE

- **C.SRAI**: Compressed Shift Right Arithmetic Immediate

  Format: c.srai rd', uimm[5:0]

  Description: performs an arithmetic right shift (sign bits are shifted into the upper bits).

  Pseudocode: x[8 + rd'] = x[8 + rd'] >>s uimm[5:0]

Invalid values: uimm[5] = 0

Exception raised: NONE

- **C.ANDI**: Compressed AND Immediate

  Format: c.andi rd', imm[5:0]

  Description: computes the bitwise AND of the value in register rd', and the sign-extended 6-bit immediate, then writes the result to rd'.

  Pseudocode: x[8 + rd'] = x[8 + rd'] & sext(imm[5:0])

  Invalid values: NONE

  Exception raised: NONE

- **C.ADD**: Compressed Addition

  Format: c.add rd, rs2

  Description: adds the values in registers rd and rs2 and writes the result to register rd.

  Pseudocode: x[rd] = x[rd] + x[rs2]

  Invalid values: rd = x0 & rs2 = x0

  Exception raised: NONE

- **C.MV**: Move

  Format: c.mv rd, rs2

  Description: copies the value in register rs2 into register rd.

  Pseudocode: x[rd] = x[rs2]

  Invalid values: rd = x0 & rs2 = x0

  Exception raised: NONE

- **C.AND**: Compressed AND

  Format: c.and rd', rs2'

  Description: computes the bitwise AND of of the value in register rd', and register rs2', then writes the result to rd'.

  Pseudocode: x[8 + rd'] = x[8 + rd'] & x[8 + rs2']

  Invalid values: NONE

  Exception raised: NONE

- **C.OR**: Compressed OR

  Format: c.or rd', rs2'

  Description: computes the bitwise OR of of the value in register rd', and register rs2', then writes the result to rd'.

  Pseudocode: x[8 + rd'] = x[8 + rd'] | x[8 + rs2']

  Invalid values: NONE

  Exception raised: NONE

- **C.XOR**: Compressed XOR

Format: c.and rd', rs2'

Description: computes the bitwise XOR of of the value in register rd', and register rs2', then writes the result to rd'.

Pseudocode: x[8 + rd'] = x[8 + rd'] ^ x[8 + rs2']

Invalid values: NONE

Exception raised: NONE

- **C.SUB**: Compressed Subtraction

  Format: c.sub rd', rs2'

  Description: subtracts the value in registers rs2' from value in rd' and writes the result to register rd'.

  Pseudocode: x[8 + rd'] = x[8 + rd'] - x[8 + rs2']

  Invalid values: NONE

  Exception raised: NONE

- **C.EBREAK**: Compressed Ebreak

  Format: c.ebreak

  Description: cause control to be transferred back to the debugging environment.

  Pseudocode: RaiseException(Breakpoint)

  Invalid values: NONE

  Exception raised: Raise a Breakpoint exception.

Control Transfer Instructions

- **C.J**: Compressed Jump

  Format: c.j imm[11:1]

  Description: performs an unconditional control transfer. The offset is sign-extended and added to the pc to form the jump target address.

  Pseudocode: pc += sext(imm[11:1])

  Invalid values: NONE

  Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception.

- **C.JAL**: Compressed Jump and Link

  Format: c.jal imm[11:1]

  Description: performs the same operation as C.J, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1.

  Pseudocode: x[1] = pc+2; pc += sext(imm[11:1])

  Invalid values: NONE

  Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception.

- **C.JR**: Compressed Jump Register

Format: c.jr rs1

Description: performs an unconditional control transfer to the address in register rs1.

Pseudocode: pc = x[rs1]

Invalid values: rs1 = x0

Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception.

- **C.JALR**: Compressed Jump and Link Register

  Format: c.jalr rs1

  Description: performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1.

  Pseudocode: t = pc+2; pc = x[rs1]; x[1] = t

  Invalid values: rs1 = x0

  Exception raised: jumps to an incorrect instruction address will usually quickly raise an exception.

- **C.BEQZ**: Branch if Equal Zero

  Format: c.beqz rs1', imm[8:1]

  Description: performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. C.BEQZ takes the branch if the value in register rs1' is zero.

  Pseudocode: if (x[8+rs1'] == 0) pc += sext(imm[8:1])

  Invalid values: NONE

  Exception raised: No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

- **C.BNEZ**: Branch if Not Equal Zero

  Format: c.bnez rs1', imm[8:1]

  Description: performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. C.BEQZ takes the branch if the value in register rs1' isn't zero.

  Pseudocode: if (x[8+rs1'] != 0) pc += sext(imm[8:1])

  Invalid values: NONE

  Exception raised: No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

Load and Store Instructions

- **C.LWSP**: Load Word Stack-Pointer

  Format: c.lwsp rd, uimm(x2)

  Description: loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

  Pseudocode: x[rd] = M[x[2] + zext(uimm[7:2])][31:0]

  Invalid values: rd = x0

  Exception raised: loads with a destination of x0 must still raise any exceptions, also an exception if the memory address isn't aligned (4-byte boundary).

- **C.SWSP**: Store Word Stack-Pointer

  Format: c.lwsp rd, uimm(x2)

  Description: stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2.

  Pseudocode: M[x[2] + zext(uimm[7:2])][31:0] = x[rs2]

  Invalid values: NONE

  Exception raised: An exception raised if the memory address isn't aligned (4-byte boundary).

- **C.LW**: Compressed Load Word

  Format: c.lw rd', uimm(rs1')

  Description: loads a 32-bit value from memory into register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

  Pseudocode: x[8+rd'] = M[x[8+rs1'] + zext(uimm[6:2])][31:0])

  Invalid values: NONE

  Exception raised: An exception raised if the memory address isn't aligned (4-byte boundary).

- **C.SW**: Compressed Store Word

  Format: c.sw rs2', uimm(rs1')

  Description: stores a 32-bit value from memory into register rd'. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1'.

  Pseudocode: M[x[8+rs1'] + zext(uimm[6:2])][31:0] = x[8+rs2']

  Invalid values: NONE

  Exception raised: An exception raised if the memory address isn't aligned (4-byte boundary).

RV32Zicsr Control and Status Register Instructions

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit csr field of the instruction held in bits 31–20. The immediate forms use a 5-bit zero-extended immediate encoded in the rs1 field.

- **CSRRW**: Control and Status Register Read and Write

  Format: csrrw rd, csr, rs1

  Description: reads the old value of the CSR, zero-extends the value to 32 bits, then writes it to integer register rd, the initial value in rs1 is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

  Pseudocode: t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t

  Invalid values: NONE

  Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

- **CSRRS**: Control and Status Register Read and Set

  Format: csrrs rd, csr, rs1

  Description: reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd, the initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if

that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written), if rs1=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs.

Pseudocode: t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t

Invalid values: NONE

Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

- **CSRRC**: Control and Status Register Read and Clear

  Format: csrrc rd, csr, rs1

  Description: reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd, the initial value in integer register rs1 is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in rs1 will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written), if rs1=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs.

  Pseudocode: t = CSRs[csr]; CSRs[csr] = t & x[rs1]; x[rd] = t

  Invalid values: NONE

  Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

- **CSRRWI**: Control and Status Register Read and Write Immediate

  Format: csrrwi rd, csr, uimm[4:0]

  Description: reads the old value of the CSR, zero-extends the value to 32 bits, then writes it to integer register rd. The zero-extends immediate is written to the CSR. If rd=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

  Pseudocode: x[rd] = CSRs[csr]; CSRs[csr] = zext(uimm[4:0])

  Invalid values: NONE

  Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

- **CSRRSI**: Control and Status Register Read and Set Immediate

  Format: csrrsi rd, csr, uimm[4:0]

  Description: reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The zero-extends immediate value is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in zero-extends immediate will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written), if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.

  Pseudocode: t = CSRs[csr]; CSRs[csr] = t | zext(uimm[4:0]); x[rd] = t

  Invalid values: NONE

  Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

- **CSRRCI**: Control and Status Register Read and Clear Immediate

  Format: csrrci rd, csr, uimm[4:0]

  Description: reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register rd. The zero-extends immediate value is treated as a bit mask that specifies bit positions to be

cleared in the CSR. Any bit that is high in zero-extends immediate will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written), if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.

Pseudocode: t = CSRs[csr]; CSRs[csr] = t & zext(uimm[4:0]); x[rd] = t

Invalid values: NONE

Exception raised: Attempts to access a non-existent CSR raise an illegal instruction exception.

RV32Zifencei Instruction-Fetch Fence

- **FENCE.I**: Fence Instruction

  Format: fence.i

  Description: The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart.

  Pseudocode: Fence(Store, Fetch)

  Invalid values: NONE

  Exception raised: NONE

2.3.5 CV32A6_CSR programmers view

Tip: This section was auto-generated by **Register Manager** from Jade Design Automation.

Register Summary

Name	Address Offset	Width	Access Type	Reset Value	Display Name
fflags	0x1	32	RW	0x00000000	Floating-Point Accrued Exceptions
frm	0x2	32	RW	0x00000000	Floating-Point Dynamic Rounding Mode
fcsr	0x3	32	RW	0x00000000	Floating-Point Control and Status Register
sstatus	0x100	32	RW	0x00000000	Supervisor Status
sie	0x104	32	RW	0x00000000	Supervisor Interrupt Enable
stvec	0x105	32	RW	0x00000000	Supervisor Trap Vector Base Address
scounteren	0x106	32	RW	0x00000000	Supervisor Counter Enable
sscratch	0x140	32	RW	0x00000000	Supervisor Scratch
sepc	0x141	32	RW	0x00000000	Supervisor Exception Program Counter
scause	0x142	32	RW	0x00000000	Supervisor Cause
stval	0x143	32	RW	0x00000000	Supervisor Trap Value
sip	0x144	32	RW	0x00000000	Supervisor Interrupt Pending
satp	0x180	32	RW	0x00000000	Supervisor Address Translation and Protection

con

Table 12 – continued from previous page

Name	Address Offset	Width	Access Type	Reset Value	Display Name
mstatus	0x300	32	RW	0x00000000	Machine Status
misa	0x301	32	RW	0x00000000	Machine ISA
medeleg	0x302	32	RW	0x00000000	Machine Exception Delegation
mideleg	0x303	32	RW	0x00000000	Machine Interrupt Delegation
mie	0x304	32	RW	0x00000000	Machine Interrupt Enable
mtvec	0x305	32	RW	0x00000000	Machine Trap Vector
mcountern	0x306	32	RW	0x00000000	Machine Counter Enable
hpmevent[6]	0x323 [+ i*0x1]	32	RW	0x00000000	Hardware Performance-Monitoring Event Select
mscratch	0x340	32	RW	0x00000000	Machine Scratch
mepc	0x341	32	RW	0x00000000	Machine Exception Program Counter
mcause	0x342	32	RW	0x00000000	Machine Cause
mtval	0x343	32	RW	0x00000000	Machine Trap Value
mip	0x344	32	RW	0x00000000	Machine Interrupt Pending
pmpcfg0	0x3A0	32	RW	0x00000000	Physical Memory Protection Config 0
pmpcfg1	0x3A1	32	RW	0x00000000	Physical Memory Protection Config 1
pmpcfg2	0x3A2	32	RW	0x00000000	Physical Memory Protection Config 2
pmpcfg3	0x3A3	32	RW	0x00000000	Physical Memory Protection Config 3
pmpaddr[16]	0x3B0 [+ i*0x1]	32	RW	0x00000000	Physical Memory Protection Address
icache	0x700	32	RW	0x00000001	Instuction Cache
dcache	0x701	32	RW	0x00000001	Data Cache
tselect	0x7A0	32	RW	0x00000000	Trigger Select
tdata1	0x7A1	32	RW	0x00000000	Trigger Data 1
tdata2	0x7A2	32	RW	0x00000000	Trigger Data 2
tdata3	0x7A3	32	RW	0x00000000	Trigger Data 3
tinfo	0x7A4	32	RO	0x00000000	Trigger Info
dcsr	0x7B0	32	RW	0x00000000	Debug Control and Status
dpc	0x7B1	32	RW	0x00000000	Debug PC
dscratch[2]	0x7B2 [+ i*0x1]	32	RW	0x00000000	Debug Scratch Register
ftran	0x800	32	RW	0x00000000	
mcycle	0xB00	32	RW	0x00000000	M-mode Cycle counter
minstret	0xB02	32	RW	0x00000000	Machine Instruction Retired counter
ml1_icache_miss	0xB03	32	RW	0x00000000	L1 Inst Cache Miss
ml1_dcache_miss	0xB04	32	RW	0x00000000	L1 Data Cache Miss
mitlb_miss	0xB05	32	RW	0x00000000	ITLB Miss
mdtlb_miss	0xB06	32	RW	0x00000000	DTLB Miss
mload	0xB07	32	RW	0x00000000	Loads
mstore	0xB08	32	RW	0x00000000	Stores
mexception	0xB09	32	RW	0x00000000	Taken Exceptions
mexception_ret	0xB0A	32	RW	0x00000000	Exception Return
mbranch_jump	0xB0B	32	RW	0x00000000	Software Change of PC
mcall	0xB0C	32	RW	0x00000000	Procedure Call
mret	0xB0D	32	RW	0x00000000	Procedure Return
mmis_predict	0xB0E	32	RW	0x00000000	Branch mis-predicted
msb_full	0xB0F	32	RW	0x00000000	Scoreboard Full
mif_empty	0xB10	32	RW	0x00000000	Instruction Fetch Queue Empty
mcycleh	0xB80	32	RW	0x00000000	Upper 32-bits of M-mode Cycle counter
minstreth	0xB82	32	RW	0x00000000	Upper 32-bits of Machine Instruction Retired co
mhpmcounterh[6]	0xB83 [+ i*0x1]	32	RW	0x00000000	Upper 32-bits of Machine Hardware Performanc
cycle	0xC00	32	RO	0x00000000	Cycle counter

con

Table 12 – continued from previous page

Name	Address Offset	Width	Access Type	Reset Value	Display Name
time	0xC01	32	RO	0x00000000	Timer
instret	0xC02	32	RO	0x00000000	Instruction Retired counter
l1_icache_miss	0xC03	32	RO	0x00000000	L1 Inst Cache Miss
l1_dcache_miss	0xC04	32	RO	0x00000000	L1 Data Cache Miss
itlb_miss	0xC05	32	RO	0x00000000	ITLB Miss
dtlb_miss	0xC06	32	RO	0x00000000	DTLB Miss
load	0xC07	32	RO	0x00000000	Loads
store	0xC08	32	RO	0x00000000	Stores
exception	0xC09	32	RO	0x00000000	Taken Exceptions
exception_ret	0xC0A	32	RO	0x00000000	Exception Return
branch_jump	0xC0B	32	RO	0x00000000	Software Change of PC
call	0xC0C	32	RO	0x00000000	Procedure Call
ret	0xC0D	32	RO	0x00000000	Procedure Return
mis_predict	0xC0E	32	RO	0x00000000	Branch mis-predicted
sb_full	0xC0F	32	RO	0x00000000	Scoreboard Full
if_empty	0xC10	32	RO	0x00000000	Instruction Fetch Queue Empty
cycleh	0xC80	32	RO	0x00000000	Upper 32-bits of Cycle counter
timeh	0xC81	32	RO	0x00000000	Upper 32-bit of Timer
instreth	0xC82	32	RO	0x00000000	Upper 32-bits of Instruction Retired counter
mvendorid	0xF11	32	RO	0x00000000	Machine Vendor ID
marchid	0xF12	32	RO	0x00000003	Machine Architecture ID
mimpid	0xF13	32	RO	0x00000000	Machine Implementation ID
mhartid	0xF14	32	RO	0x00000000	Machine Hardware Thread ID

Register Descriptions

Floating-Point Accrued Exceptions (fflags)

Address Offset
0x1

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The fields within the fcsr can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field frm and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in frm by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into frm. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field fflags.

Bits	Name	Display Name	Access Type	Reset
[31:5]	reserved_0	Reserved	RO	0b0
[4]	NV	Invalid Operation	RW	0b0
[3]	DZ	Divide by Zero	RW	0b0
[2]	OF	Overflow	RW	0b0
[1]	UF	Underflow	RW	0b0
[0]	NX	Inexact	RW	0b0

Invalid Operation (NV)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Divide by Zero (DZ)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Overflow (OF)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Underflow (UF)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Inexact (NX)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Floating-Point Dynamic Rounding Mode (frm)

Address Offset
0x2

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The fields within the fcsr can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field frm and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in frm by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into frm. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field fflags.

Bits	Name	Display Name	Access Type	Reset
[31:3]	`reserved_0`	Reserved	RO	0b0
[2:0]	`FRM`	Floating-Point Rounding Mode	RW	0b0

Floating-Point Rounding Mode (`FRM`)

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in the enumerated value. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception. Some instructions, including widening conversions, have the *rm* field but are nevertheless unaffected by the rounding mode; software should set their *rm* field to RNE (000).

Table 13: The following table shows the bitfield encoding

Value	Name	Description
0b000	RNE	Round to Nearest, ties to Even
0b001	RTZ	Round towards Zero
0b010	RDN	Round Down
0b011	RUP	Round Up
0b100	RMM	Round to Nearest, ties to Max Magnitude
0b101 - 0b110	INVALID	*Reserved for future use.*
0b111	DYN	**In instruction's *rm* field, selects dynamic rounding mode;** In Rounding Mode register, *Invalid*.

Floating-Point Control and Status Register (`fcsr`)

Address Offset

0x3

Width (bits)

32

Access Type

RW

Reset Value

0x00000000

Description

The floating-point control and status register, `fcsr`, is a RISC-V control and status register (CSR). It is a read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags.

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads `fcsr` by copying it into integer register *rd*. FSCSR swaps the value in `fcsr` by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into `fcsr`.

The fields within the `fcsr` can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field `frm` and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in frm by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into `frm`. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field `fflags`.

Bits	Name	Display Name	Access Type	Reset
[31:8]	`reserved_0`	Reserved	RO	0b0
[7:5]	`FRM`	Floating-Point Rounding Mode	RW	0b0
[4]	`NV`	Invalid Operation	RW	0b0
[3]	`DZ`	Divide by Zero	RW	0b0
[2]	`OF`	Overflow	RW	0b0
[1]	`UF`	Underflow	RW	0b0
[0]	`NX`	Inexact	RW	0b0

Floating-Point Rounding Mode (`FRM`)

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in the enumerated value. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception. Some instructions, including widening conversions, have the *rm* field but are nevertheless unaffected by the rounding mode; software should set their *rm* field to RNE (000).

Table 14: The following table shows the bitfield encoding

Value	Name	Description
0b000	RNE	Round to Nearest, ties to Even
0b001	RTZ	Round towards Zero
0b010	RDN	Round Down
0b011	RUP	Round Up
0b100	RMM	Round to Nearest, ties to Max Magnitude
0b101 - 0b110	INVALID	*Reserved for future use.*
0b111	DYN	**In instruction's *rm* field, selects dynamic rounding mode;** In Rounding Mode register, *Invalid*.

Invalid Operation (`NV`)

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Divide by Zero (`DZ`)

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Overflow (`OF`)

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Underflow (UF)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Inexact (NX)
The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Supervisor Status (sstatus)

Address Offset
0x100

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The sstatus register keeps track of the processor's current operating state.

The sstatus register is a subset of the mstatus register.

Bits	Name	Display Name	Access Type	Reset
[31]	SD	State Dirty	RO	0b0
[30:20]	reserved_0	Reserved	RO	0b0
[19]	MXR	Make eXecutable Readable	RW	0b0
[18]	SUM	Supervisor User Memory	RW	0b0
[17]	reserved_1	Reserved	RO	0b0
[16:15]	XS	Extension State	RO	0b0
[14:13]	FS	Floating-point unit State	RW	0b0
[12:9]	reserved_2	Reserved	RO	0b0
[8]	SPP	Supervisor mode Prior Privilege	RW	0b0
[7:6]	reserved_3	Reserved	RO	0b0
[5]	SPIE	Supervisor mode Prior Interrupt Enable	RW	0b0
[4]	UPIE		RW	0b0
[3:2]	reserved_4	Reserved	RO	0b0
[1]	SIE	Supervisor mode Interrupt Enable	RW	0b0
[0]	UIE		RW	0b0

State Dirty (SD)
The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero.

Make eXecutable Readable (MXR)
The MXR bit modifies the privilege with which loads access virtual memory. When MXR=0, only

loads from pages marked readable will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect.

Supervisor User Memory (`SUM`)

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it *is* in effect when MPRV=1 and MPP=S. SUM is read-only 0 if S-mode is not supported or if `satp`.MODE is read-only 0.

Extension State (`XS`)

The XS field is used to reduce the cost of context save and restore by setting and tracking the current state of the user-mode extensions. The XS field encodes the status of the additional user-mode extensions and associated state.

This field can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

Table 15: The following table shows the bitfield encoding

Value	Name	Description
0b00	Off	All off
0b01	Initial	None dirty or clean, some on
0b10	Clean	None dirty, some clean
0b11	Dirty	Some dirty

Floating-point unit State (`FS`)

The FS field is used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit. The FS field encodes the status of the floating-point unit state, including the floating-point registers `f0`-`f31` and the CSRs `fcsr`, `frm`, and `fflags`.

This field can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

Table 16: The following table shows the bitfield encoding

Value	Name	Description
0b00	Off	
0b01	Initial	
0b10	Clean	
0b11	Dirty	

Supervisor mode Prior Privilege (`SPP`)

SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

Supervisor mode Prior Interrupt Enable (`SPIE`)

The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1.

UPIE
When a URET instruction is executed, UIE is set to UPIE, and UPIE is set to 1.

Supervisor mode Interrupt Enable (SIE)
The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the `sie` CSR.

UIE
The UIE bit enables or disables user-mode interrupts.

Supervisor Interrupt Enable (sie)

Address Offset
0x104

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The `sie` is the register containing supervisor interrupt enable bits.

Bits	Name	Display Name	Access Type	Reset
[31:10]	reserved_0	Reserved	RO	0b0
[9]	SEIE	Supervisor-level External Interrupt Enable	RW	0b0
[8]	UEIE		RW	0b0
[7:6]	reserved_1	Reserved	RO	0b0
[5]	STIE	Supervisor-level Timer Interrupt Enable	RW	0b0
[4]	UTIE		RW	0b0
[3:2]	reserved_2	Reserved	RO	0b0
[1]	SSIE	Supervisor-level Software Interrupt Enable	RW	0b0
[0]	USIE		RW	0b0

Supervisor-level External Interrupt Enable (SEIE)
SEIE is the interrupt-enable bit for supervisor-level external interrupts.

UEIE
User-level external interrupts are disabled when the UEIE bit in the sie register is clear.

Supervisor-level Timer Interrupt Enable (STIE)
STIE is the interrupt-enable bit for supervisor-level timer interrupts.

UTIE
User-level timer interrupts are disabled when the UTIE bit in the sie register is clear.

Supervisor-level Software Interrupt Enable (SSIE)
SSIE is the interrupt-enable bit for supervisor-level software interrupts.

USIE
User-level software interrupts are disabled when the USIE bit in the sie register is clear

Supervisor Trap Vector Base Address (stvec)

Address Offset
0x105

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The `stvec` register holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

Bits	Name	Display Name	Access Type	Reset
[31:2]	BASE		RW	0b0
[1:0]	MODE		RW	0b0

BASE
The BASE field in stvec is a WARL field that can hold any valid virtual or physical address, subject to the following alignment constraints: the address must be 4-byte aligned, and MODE settings other than Direct might impose additional alignment constraints on the value in the BASE field.

MODE
When MODE=Direct, all traps into supervisor mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into supervisor mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number.

Table 17: The following table shows the bitfield encoding

Value	Name	Description
0b00	Direct	All exceptions set `pc` to BASE.
0b01	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
0b10 - 0b11	Reserved	*Reserved*

Supervisor Counter Enable (scounteren)

Address Offset
0x106

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The counter-enable register `scounteren` controls the availability of the hardware performance monitoring counters to U-mode.

Bits	Name	Display Name	Access Type	Reset
[31:3]	HPMn	Hpmcountern	RW	0b0
[2]	IR	Instret	RW	0b0
[1]	TM	Time	RW	0b0
[0]	CY	Cycle	RW	0b0

Hpmcountern (HPMn)
When HPMn is clear, attempts to read the hpmcountern register while executing in U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted.

Instret (IR)
When IR is clear, attempts to read the instret register while executing in U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted.

Time (TM)
When TM is clear, attempts to read the time register while executing in U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted.

Cycle (CY)
When CY is clear, attempts to read the cycle register while executing in U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted.

Supervisor Scratch (sscratch)

Address Offset
0x140

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The sscratch register is dedicated for use by the supervisor.

Bits	Name	Display Name	Access Type	Reset
[31:0]	SSCRATCH	Supervisor Scratch	RW	0b0

Supervisor Scratch (SSCRATCH)
Typically, sscratch is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, sscratch is swapped with a user register to provide an initial working register.

Supervisor Exception Program Counter (sepc)

Address Offset
0x141

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
When a trap is taken into S-mode, `sepc` is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, `sepc` is never written by the implementation, though it may be explicitly written by software.

Bits	Name	Display Name	Access Type	Reset
[31:0]	SEPC	Supervisor Exception Program Counter	RW	0b0

Supervisor Exception Program Counter (SEPC)
The low bit of SEPC (SEPC[0]) is always zero. On implementations that support only IALIGN=32, the two low bits (SEPC[1:0]) are always zero.

Supervisor Cause (scause)

Address Offset
0x142

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
When a trap is taken into S-mode, `scause` is written with a code indicating the event that caused the trap. Otherwise, `scause` is never written by the implementation, though it may be explicitly written by software.

Supervisor cause register (`scause`) values after trap are shown in the following table.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2-4	*Reserved*
1	5	Supervisor timer interrupt
1	6-8	*Reserved*
1	9	Supervisor external interrupt
1	10-15	*Reserved*
1	16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16-23	*Reserved*
0	24-31	*Designated for custom use*
0	32-47	*Reserved*
0	48-63	*Designated for custom use*
0	64	*Reserved*

Bits	Name	Display Name	Access Type	Reset
[31]	`Interrupt`		RW	0b0
[30:0]	`Exception_Code`	Exception Code	RW	0b0

`Interrupt`
The Interrupt bit in the `scause` register is set if the trap was caused by an interrupt.

Exception Code (`Exception_Code`)
The Exception Code field contains a code identifying the last exception or interrupt.

Supervisor Trap Value (`stval`)

Address Offset
0x143

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
When a trap is taken into S-mode, `stval` is written with exception-specific information to assist software in handling the trap. Otherwise, `stval` is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set `stval` informatively and which may unconditionally set it to zero.

Bits	Name	Display Name	Access Type	Reset
[31:0]	STVAL	Supervisor Trap Value	RW	0b0

Supervisor Trap Value (STVAL)
If `stval` is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then `stval` will contain the faulting virtual address.

If `stval` is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then `stval` will contain the virtual address of the portion of the access that caused the fault.

If `stval` is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a system with variable-length instructions, then `stval` will contain the virtual address of the portion of the instruction that caused the fault, while `sepc` will point to the beginning of the instruction.

The `stval` register can optionally also be used to return the faulting instruction bits on an illegal instruction exception (`sepc` points to the faulting instruction in memory). If `stval` is written with a nonzero value when an illegal-instruction exception occurs, then `stval` will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first SXLEN bits of the faulting instruction

The value loaded into `stval` on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero. For other traps, `stval` is set to zero, but a future standard may redefine `stval's` setting for other traps.

Supervisor Interrupt Pending (sip)

Address Offset
0x144

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The `sip` register contains information on pending interrupts.

Bits	Name	Display Name	Access Type	Reset
[31:10]	reserved_0	Reserved	RO	0b0
[9]	SEIP	Supervisor-level External Interrupt Pending	RO	0b0
[8]	UEIP		RW	0b0
[7:6]	reserved_1	Reserved	RO	0b0
[5]	STIP	Supervisor-level Timer Interrupt Pending	RO	0b0
[4]	UTIP		RW	0b0
[3:2]	reserved_2	Reserved	RO	0b0
[1]	SSIP	Supervisor-level Software Interrupt Pending	RO	0b0
[0]	USIP		RW	0b0

Supervisor-level External Interrupt Pending (SEIP)
SEIP is the interrupt-pending bit for supervisor-level external interrupts.

UEIP
UEIP may be written by S-mode software to indicate to U-mode that an external interrupt is pending.

Supervisor-level Timer Interrupt Pending (STIP)
SEIP is the interrupt-pending bit for supervisor-level timer interrupts.

UTIP
A user-level timer interrupt is pending if the UTIP bit in the sip register is set

Supervisor-level Software Interrupt Pending (SSIP)
SSIP is the interrupt-pending bit for supervisor-level software interrupts.

USIP
A user-level software interrupt is triggered on the current hart by riting 1 to its user software interrupt-pending (USIP) bit

Supervisor Address Translation and Protection (satp)

Address Offset
0x180

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The satp register controls supervisor-mode address translation and protection.

The satp register is considered active when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of satp when satp is active.

Note: Writing satp does not imply any ordering constraints between page-table updates and subsequent address translations, nor does it imply any invalidation of address-translation caches. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an SFENCE.VMA instruction after, or in some cases before, writing satp.

Bits	Name	Display Name	Access Type	Reset
[31]	MODE	Mode	RW	0b0
[30:22]	ASID	Address Space Identifier	RW	0b0
[21:0]	PPN	Physical Page Number	RW	0b0

Mode (MODE)
This bitfield selects the current address-translation scheme.

When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme.

To select MODE=Bare, software must write zero to the remaining fields of satp (bits 30–0). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an unspecified effect on the value that the remaining fields assume and an unspecified effect on address translation and protection behavior.

Table 18: The following table shows the bitfield encoding

Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

Address Space Identifier (ASID)
This bitfield facilitates address-translation fences on a per-address-space basis.

Physical Page Number (PPN)
This bitfield holds the root page table, i.e., its supervisor physical address divided by 4 KiB.

Machine Status (mstatus)

Address Offset
0x300

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The mstatus register keeps track of and controls the hart's current operating state.

Bits	Name	Display Name	Access Type	Reset
[31]	SD	State Dirty	RO	0b0
[30:23]	reserved_0	Reserved	RO	0b0
[22]	TSR	Trap SRET	RW	0b0
[21]	TW	Timeout Wait	RW	0b0
[20]	TVM	Trap Virtual Memory	RW	0b0
[19]	MXR	Make eXecutable Readable	RW	0b0
[18]	SUM	Supervisor User Memory	RW	0b0
[17]	MPRV	Modify Privilege	RW	0b0
[16:15]	XS	Extension State	RO	0b0
[14:13]	FS	Floating-point unit State	RW	0b0
[12:11]	MPP	Machine mode Prior Privilege	RW	0b0
[10:9]	reserved_1	Reserved	RO	0b0
[8]	SPP	Supervisor mode Prior Privilege	RW	0b0
[7]	MPIE	Machine mode Prior Interrupt Enable	RW	0b0
[6]	reserved_2	Reserved	RO	0b0
[5]	SPIE	Supervisor mode Prior Interrupt Enable	RW	0b0
[4]	UPIE		RW	0b0
[3]	MIE	Machine mode Interrupt Enable	RW	0b0
[2]	reserved_3	Reserved	RO	0b0
[1]	SIE	Supervisor mode Interrupt Enable	RW	0b0
[0]	UIE		RW	0b0

State Dirty (SD)

The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero.

Trap SRET (TSR)

The TSR bit supports intercepting the supervisor exception return instruction, SRET. When TSR=1, attempts to execute SRET while executing in S-mode will raise an illegal instruction exception. When TSR=0, this operation is permitted in S-mode.

Timeout Wait (TW)

The TW bit supports intercepting the WFI instruction. When TW=0, the WFI instruction may execute in lower privilege modes when not prevented for some other reason. When TW=1, then if WFI is executed in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit, the WFI instruction causes an illegal instruction exception. The time limit may always be 0, in which case WFI always causes an illegal instruction exception in less-privileged modes when TW=1.

Trap Virtual Memory (TVM)

The TVM bit supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the satp CSR or execute an SFENCE.VMA or SINVAL.VMA instruction while executing in S-mode will raise an illegal instruction exception. When TVM=0, these operations are permitted in S-mode.

Make eXecutable Readable (MXR)

The MXR bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect.

Supervisor User Memory (SUM)

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode

loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it is in effect when MPRV=1 and MPP=S.

Modify Privilege (`MPRV`)
The MPRV (Modify PRiVilege) bit modifies the effective privilege mode, i.e., the privilege level at which loads and stores execute. When MPRV=0, loads and stores behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV.

Extension State (`XS`)
The XS field is used to reduce the cost of context save and restore by setting and tracking the current state of the user-mode extensions. The XS field encodes the status of the additional user-mode extensions and associated state.

This field can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

Table 19: The following table shows the bitfield encoding

Value	Name	Description
0b00	Off	All off
0b01	Initial	None dirty or clean, some on
0b10	Clean	None dirty, some clean
0b11	Dirty	Some dirty

Floating-point unit State (`FS`)
The FS field is used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit. The FS field encodes the status of the floating-point unit state, including the floating-point registers `f0-f31` and the CSRs `fcsr`, `frm`, and `fflags`.

This field can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.

Table 20: The following table shows the bitfield encoding

Value	Name	Description
0b00	Off	
0b01	Initial	
0b10	Clean	
0b11	Dirty	

Machine mode Prior Privilege (`MPP`)
Holds the previous privilege mode for machine mode.

Supervisor mode Prior Privilege (`SPP`)
Holds the previous privilege mode for supervisor mode.

Machine mode Prior Interrupt Enable (`MPIE`)
Indicates whether machine interrupts were enabled prior to trapping into machine mode.

Supervisor mode Prior Interrupt Enable (`SPIE`)
Indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode.

UPIE
indicates whether user-level interrupts were enabled prior to taking a user-level trap

Machine mode Interrupt Enable (MIE)
Global interrupt-enable bit for Machine mode.

Supervisor mode Interrupt Enable (SIE)
Global interrupt-enable bit for Supervisor mode.

UIE
Global interrupt-enable bits

Machine ISA (misa)

Address Offset
0x301

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
The misa CSR is reporting the ISA supported by the hart.

Bits	Name	Display Name	Access Type	Reset
[31:30]	MXL	Machine XLEN	RW	0b0
[29:26]	reserved_0	Reserved	RO	0b0
[25:0]	Extensions	Extensions	RW	0b0

Machine XLEN (MXL)
The MXL field encodes the native base integer ISA width.

Table 21: The following table shows the bitfield encoding

Value	Name	Description
0b01	XLEN_32	
0b10	XLEN_64	
0b11	XLEN_128	

Extensions (Extensions)
The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet.

Table 22: The following table shows the bitfield encoding

Value	Name	Description
0b00000000000000000000000000000001	A	Atomic extension.
0b00000000000000000000000000000010	B	*Tentatively reserved for Bit-Manipulation extension.*
0b00000000000000000000000000000100	C	Compressed extension.
0b00000000000000000000000000001000	D	Double-precision floating-point extension.
0b00000000000000000000000000010000	E	RV32E base ISA.
0b00000000000000000000000000100000	F	Single-precision floating-point extension.
0b00000000000000000000000001000000	G	*Reserved.*
0b00000000000000000000000010000000	H	Hypervisor extension.
0b00000000000000000000000100000000	I	RV32I/64I/128I base ISA.
0b00000000000000000000001000000000	J	*Tentatively reserved for Dynamically Translated Languages extension.*
0b00000000000000000000010000000000	K	*Reserved.*
0b00000000000000000000100000000000	L	*Reserved.*
0b00000000000000000001000000000000	M	Integer Multiply/Divide extension.
0b00000000000000000010000000000000	N	*Tentatively reserved for User-Level Interrupts extension.*
0b00000000000000000100000000000000	O	*Reserved.*
0b00000000000000001000000000000000	P	*Tentatively reserved for Packed-SIMD extension.*
0b00000000000000010000000000000000	Q	Quad-precision floating-point extension.
0b00000000000000100000000000000000	R	*Reserved.*
0b00000000000001000000000000000000	S	Supervisor mode implemented.
0b00000000000010000000000000000000	T	*Reserved.*
0b00000000000100000000000000000000	U	User mode implemented.
0b00000000001000000000000000000000	V	*Tentatively reserved for Vector extension.*
0b00000000010000000000000000000000	W	*Reserved.*
0b00000000100000000000000000000000	X	Non-standard extensions present.
0b00000001000000000000000000000000	Y	*Reserved.*
0b00000010000000000000000000000000	Z	*Reserved.*

Machine Exception Delegation (`medeleg`)

Address Offset
0x302

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Provides individual read/write bits to indicate that certain exceptions should be processed directly by a lower privilege level.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`Synchronous_Exceptions`	Synchronous Exceptions	RW	0b0

Synchronous Exceptions (`Synchronous_Exceptions`)
There is a bit position allocated for every synchronous exception, with the index of the bit position

equal to the value returned in the `mcause` register.

Machine Interrupt Delegation (`mideleg`)

Address Offset
0x303

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Provides individual read/write bits to indicate that certain interrupts should be processed directly by a lower privilege level.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`Interrupts`	Interrupts	RW	0b0

Interrupts (`Interrupts`)
This bitfield holds trap delegation bits for individual interrupts, with the layout of bits matching those in the `mip` register.

Machine Interrupt Enable (`mie`)

Address Offset
0x304

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register contains machine interrupt enable bits.

Bits	Name	Display Name	Access Type	Reset
[31:12]	reserved_0	Reserved	RO	0b0
[11]	MEIE	M-mode External Interrupt Enable	RW	0b0
[10]	reserved_1	Reserved	RO	0b0
[9]	SEIE	S-mode External Interrupt Enable	RW	0b0
[8]	UEIE		RW	0b0
[7]	MTIE	M-mode Timer Interrupt Enable	RW	0b0
[6]	reserved_2	Reserved	RO	0b0
[5]	STIE	S-mode Timer Interrupt Enable	RW	0b0
[4]	UTIE		RW	0b0
[3]	MSIE	M-mode Software Interrupt Enable	RW	0b0
[2]	reserved_3	Reserved	RO	0b0
[1]	SSIE	S-mode Software Interrupt Enable	RW	0b0
[0]	USIE		RW	0b0

M-mode External Interrupt Enable (MEIE)
Enables machine mode external interrupts.

S-mode External Interrupt Enable (SEIE)
Enables supervisor mode external interrupts.

UEIE
enables U-mode external interrupts

M-mode Timer Interrupt Enable (MTIE)
Enables machine mode timer interrupts.

S-mode Timer Interrupt Enable (STIE)
Enables supervisor mode timer interrupts.

UTIE
timer interrupt-enable bit for U-mode

M-mode Software Interrupt Enable (MSIE)
Enables machine mode software interrupts.

S-mode Software Interrupt Enable (SSIE)
Enables supervisor mode software interrupts.

USIE
enable U-mode software interrrupts

Machine Trap Vector (mtvec)

Address Offset
0x305

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register holds trap vector configuration, consisting of a vector base address and a vector mode.

Bits	Name	Display Name	Access Type	Reset
[31:2]	BASE		RW	0b0
[1:0]	MODE		RW	0b0

BASE
Holds the vector base address. The value in the BASE field must always be aligned on a 4-byte boundary.

MODE
Imposes additional alignment constraints on the value in the BASE field.

Table 23: The following table shows the bitfield encoding

Value	Name	Description
0b00	Direct	All exceptions set pc to BASE.
0b01	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
0b10-0b11	Reserved	*Reserved.*

Machine Counter Enable (mcountern)

Address Offset
0x306

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register controls the availability of the hardware performance-monitoring counters to the next-lowest privileged mode.

Bits	Name	Display Name	Access Type	Reset
[31:3]	HPMn	Hpmcountern	RW	0b0
[2]	IR	Instret	RW	0b0
[1]	TM	Time	RW	0b0
[0]	CY	Cycle	RW	0b0

Hpmcountern (HPMn)
When HPMn is clear, attempts to read the hpmcountern register while executing in S-mode or U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted in the next implemented privilege mode.

Instret (IR)
When IR is clear, attempts to read the instret register while executing in S-mode or U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted in the next implemented privilege mode.

Time (TM)
When TM is clear, attempts to read the time register while executing in S-mode or U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted in the next implemented privilege mode.

Cycle (CY)
When CY is clear, attempts to read the `cycle` register while executing in S-mode or U-mode will cause an illegal instruction exception. When this bit is set, access to the corresponding register is permitted in the next implemented privilege mode.

Hardware Performance-Monitoring Event Selector (`hpmevent[6]`)

Address Offset
0x323 [+ i*0x1]

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register controls which event causes the corresponding counter to increment.

Bits	Name	Display Name	Access Type	Reset
[31:5]	`reserved_0`	Reserved	RO	0b0
[4:0]	`mhpmevent`		RW	0b0

`mhpmevent`
event selector CSRs

Machine Scratch (`mscratch`)

Address Offset
0x340

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`mscratch`	Machine Scratch	RW	0b0

Machine Scratch (`mscratch`)
Holds a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.

Machine Exception Program Counter (mepc)

Address Offset
0x341

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register must be able to hold all valid virtual addresses.

Bits	Name	Display Name	Access Type	Reset
[31:0]	mepc	Machine Exception Program Counter	RW	0b0

Machine Exception Program Counter (mepc)
When a trap is taken into M-mode, mepc is written with the virtual address of the instruction that was interrupted or that encountered the exception.

Machine Cause (mcause)

Address Offset
0x342

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
When a trap is taken into M-mode, mcause is written with a code indicating the event that caused the trap.

Machine cause register (mcause) values after trap are shown in the following table.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2-4	*Reserved*
1	5	Supervisor timer interrupt
1	6-8	*Reserved*
1	9	Supervisor external interrupt
1	10-15	*Reserved*
1	16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16-23	*Reserved*
0	24-31	*Designated for custom use*
0	32-47	*Reserved*
0	48-63	*Designated for custom use*
0	64	*Reserved*

Bits	Name	Display Name	Access Type	Reset
[31]	`Interrupt`	Interrupt	RW	0b0
[30:0]	`exception_code`	Exception Code	RW	0b0

Interrupt (`Interrupt`)
This bit is set if the trap was caused by an interrupt.

Exception Code (`exception_code`)
This field contains a code identifying the last exception or interrupt.

Machine Trap Value (`mtval`)

Address Offset
0x343

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
When a trap is taken into M-mode, mtval is either set to zero or written with exception-specific information to assist software in handling the trap.

Bits	Name	Display Name	Access Type	Reset
[31:0]	mtval	Machine Trap Value	RW	0b0

Machine Trap Value (mtval)
If mtval is written with a nonzero value when a breakpoint, address-misaligned, access-fault, or page-fault exception occurs on an instruction fetch, load, or store, then mtval will contain the faulting virtual address.

If mtval is written with a nonzero value when a misaligned load or store causes an access-fault or page-fault exception, then mtval will contain the virtual address of the portion of the access that caused the fault.

If mtval is written with a nonzero value when an instruction access-fault or page-fault exception occurs on a system with variable-length instructions, then mtval will contain the virtual address of the portion of the instruction that caused the fault, while mepc will point to the beginning of the instruction.

Machine Interrupt Pending (mip)

Address Offset
0x344

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register contains machine interrupt pending bits.

Bits	Name	Display Name	Access Type	Reset
[31:12]	reserved_0	Reserved	RO	0b0
[11]	MEIP	M-mode External Interrupt Pending	RO	0b0
[10]	reserved_1	Reserved	RO	0b0
[9]	SEIP	S-mode External Interrupt Pending	RW	0b0
[8]	UEIP		RW	0b0
[7]	MTIP	M-mode Timer Interrupt Pending	RO	0b0
[6]	reserved_2	Reserved	RO	0b0
[5]	STIP	S-mode Timer Interrupt Pending	RW	0b0
[4]	UTIP		RW	0b0
[3]	MSIP	M-mode Software Interrupt Pending	RO	0b0
[2]	reserved_3	Reserved	RO	0b0
[1]	SSIP	S-mode Software Interrupt Pending	RW	0b0
[0]	USIP		RW	0b0

M-mode External Interrupt Pending (MEIP)
The interrupt-pending bit for machine-level external interrupts.

S-mode External Interrupt Pending (SEIP)
The interrupt-pending bit for supervisor-level external interrupts.

UEIP
enables external interrupts

M-mode Timer Interrupt Pending (MTIP)
The interrupt-pending bit for machine-level timer interrupts.

S-mode Timer Interrupt Pending (STIP)
The interrupt-pending bit for supervisor-level timer interrupts.

UTIP
Correspond to timer interrupt-pending bits for user interrupt

M-mode Software Interrupt Pending (MSIP)
The interrupt-pending bit for machine-level software interrupts.

S-mode Software Interrupt Pending (SSIP)
The interrupt-pending bit for supervisor-level software interrupts.

USIP
A hart to directly write its own USIP bits when running in the appropriate mode

Physical Memory Protection Config 0 (pmpcfg0)

Address Offset
0x3A0

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Holds configuration 0-3.

Bits	Name	Display Name	Access Type	Reset
[31:24]	pmp3cfg	Physical Memory Protection 3 Config	RW	0b0
[23:16]	pmp2cfg	Physical Memory Protection 2 Config	RW	0b0
[15:8]	pmp1cfg	Physical Memory Protection 1 Config	RW	0b0
[7:0]	pmp0cfg	Physical Memory Protection 0 Config	RW	0b0

Physical Memory Protection 3 Config (pmp3cfg)
Holds the configuration.

Physical Memory Protection 2 Config (pmp2cfg)
Holds the configuration.

Physical Memory Protection 1 Config (pmp1cfg)
Holds the configuration.

Physical Memory Protection 0 Config (pmp0cfg)
Holds the configuration.

Physical Memory Protection Config 1 (pmpcfg1)

Address Offset
0x3A1

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Holds configuration 4-7.

Bits	Name	Display Name	Access Type	Reset
[31:24]	pmp7cfg	Physical Memory Protection 7 Config	RW	0b0
[23:16]	pmp6cfg	Physical Memory Protection 6 Config	RW	0b0
[15:8]	pmp5cfg	Physical Memory Protection 5 Config	RW	0b0
[7:0]	pmp4cfg	Physical Memory Protection 4 Config	RW	0b0

Physical Memory Protection 7 Config (pmp7cfg)
Holds the configuration.

Physical Memory Protection 6 Config (pmp6cfg)
Holds the configuration.

Physical Memory Protection 5 Config (pmp5cfg)
Holds the configuration.

Physical Memory Protection 4 Config (pmp4cfg)
Holds the configuration.

Physical Memory Protection Config 2 (pmpcfg2)

Address Offset
0x3A2

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Holds configuration 8-11.

Bits	Name	Display Name	Access Type	Reset
[31:24]	pmp11cfg	Physical Memory Protection 11 Config	RW	0b0
[23:16]	pmp10cfg	Physical Memory Protection 10 Config	RW	0b0
[15:8]	pmp9cfg	Physical Memory Protection 9 Config	RW	0b0
[7:0]	pmp8cfg	Physical Memory Protection 8 Config	RW	0b0

Physical Memory Protection 11 Config (`pmp11cfg`)
Holds the configuration.

Physical Memory Protection 10 Config (`pmp10cfg`)
Holds the configuration.

Physical Memory Protection 9 Config (`pmp9cfg`)
Holds the configuration.

Physical Memory Protection 8 Config (`pmp8cfg`)
Holds the configuration.

Physical Memory Protection Config 3 (`pmpcfg3`)

Address Offset
0x3A3

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Holds configuration 12-15.

Bits	Name	Display Name	Access Type	Reset
[31:24]	`pmp15cfg`	Physical Memory Protection 15 Config	RW	0b0
[23:16]	`pmp14cfg`	Physical Memory Protection 14 Config	RW	0b0
[15:8]	`pmp13cfg`	Physical Memory Protection 13 Config	RW	0b0
[7:0]	`pmp12cfg`	Physical Memory Protection 12 Config	RW	0b0

Physical Memory Protection 15 Config (`pmp15cfg`)
Holds the configuration.

Physical Memory Protection 14 Config (`pmp14cfg`)
Holds the configuration.

Physical Memory Protection 13 Config (`pmp13cfg`)
Holds the configuration.

Physical Memory Protection 12 Config (`pmp12cfg`)
Holds the configuration.

Physical Memory Protection Address (`pmpaddr[16]`)

Address Offset
0x3B0 [+ i*0x1]

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Address register for Physical Memory Protection.

Bits	Name	Display Name	Access Type	Reset
[31:0]	address	Address	RW	0b0

Address (address)
Encodes bits 33-2 of a 34-bit physical address.

Instuction Cache (icache)

Address Offset
0x700

Width (bits)
32

Access Type
RW

Reset Value
0x00000001

Description
Custom Register to enable/disable for Icache [bit 0]

Bits	Name	Display Name	Access Type	Reset
[31:1]	reserved_0	Reserved	RO	0b0
[0]	icache	Instruction Cache	RW	0b1

Instruction Cache (icache)
Custom Register

Data Cache (dcache)

Address Offset
0x701

Width (bits)
32

Access Type
RW

Reset Value
0x00000001

Description
Custom Register to enable/disable for Dcache [bit 0]

Bits	Name	Display Name	Access Type	Reset
[31:1]	reserved_0	Reserved	RO	0b0
[0]	dcache	Data Cache	RW	0b1

Data Cache (`dcache`)
Custom Register

Trigger Select (`tselect`)

Address Offset
0x7A0

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
This register determines which trigger is accessible through the other trigger registers.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`index`	Index	RW	0b0

Index (`index`)
The set of accessible triggers must start at 0, and be contiguous.

Writes of values greater than or equal to the number of supported triggers may result in a different value in this register than what was written. To verify that what they wrote is a valid index, debuggers can read back the value and check that `tselect` holds what they wrote.

Since triggers can be used both by Debug Mode and M-mode, the debugger must restore this register if it modifies it.

Trigger Data 1 (`tdata1`)

Address Offset
0x7A1

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Trigger-specific data.

Bits	Name	Display Name	Access Type	Reset
[31:28]	`type`	Type	RW	0b0
[27]	`dmode`	Debug Mode	RW	0b0
[26:0]	`data`	Data	RW	0b0

Type (`type`)
Type of trigger.

Table 24: The following table shows the bitfield encoding

Value	Name	Description
0b0000	no_trigger	There is no trigger at this `tselect`.
0b0001	legacy_address_match_trigger	The trigger is a legacy SiFive address match trigger. These should not be implemented and aren't further documented here.
0b0010	address_data_match_trigger	The trigger is an address/data match trigger. The remaining bits in this register act as described in `mcontrol`.
0b0011	instruction_count_trigger	The trigger is an instruction count trigger. The remaining bits in this register act as described in `icount`.
0b0100	interrupt_trigger	The trigger is an interrupt trigger. The remaining bits in this register act as described in `itrigger`.
0b0101	exception_trigger	The trigger is an exception trigger. The remaining bits in this register act as described in `etrigger`.
0b0110-0b1110	Reserved	*Reserved.*
0b1111	trigger_exists	This trigger exists (so enumeration shouldn't terminate), but is not currently available.

Debug Mode (`dmode`)
This bit is only writable from Debug Mode.

Table 25: The following table shows the bitfield encoding

Value	Name	Description
0	D_and_M_mode	Both Debug and M-mode can write the `tdata` registers at the selected `tselect`.
1	M_mode_only	Only Debug Mode can write the `tdata` registers at the selected `tselect`. Writes from other modes are ignored.

Data (`data`)
Trigger-specific data.

Trigger Data 2 (`tdata2`)

Address Offset
0x7A2

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Trigger-specific data.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`data`	Data	RW	0b0

Data (data)
Trigger-specific data.

Trigger Data 3 (tdata3)

Address Offset
0x7A3

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Trigger-specific data.

Bits	Name	Display Name	Access Type	Reset
[31:0]	data	Data	RW	0b0

Data (data)
Trigger-specific data.

Trigger Info (tinfo)

Address Offset
0x7A4

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Shows trigger information.

Bits	Name	Display Name	Access Type	Reset
[31:16]	reserved_0	Reserved	RO	0b0
[15:0]	info	Info	RO	0b0

Info (info)
One bit for each possible type enumerated in tdata1. Bit N corresponds to type N. If the bit is set, then that type is supported by the currently selected trigger.

If the currently selected trigger doesn't exist, this field contains 1.

If type is not writable, this register may be unimplemented, in which case reading it causes an illegal instruction exception. In this case the debugger can read the only supported type from tdata1.

Debug Control and Status (dcsr)

Address Offset
0x7B0

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Debug ontrol and status register.

Bits	Name	Display Name	Access Type	Reset
[31:28]	xdebugver	Debug Version	RO	0b0
[27:16]	reserved_0	Reserved	RO	0b0
[15]	ebreakm	Environment Breakpoint M-mode	RW	0b0
[14]	reserved_1	Reserved	RO	0b0
[13]	ebreaks	Environment Breakpoint S-mode	RW	0b0
[12]	ebreaku	Environment Breakpoint U-mode	RW	0b0
[11]	stepie	Stepping Interrupt Enable	RW	0b0
[10]	stopcount	Stop Counters	RW	0b0
[9]	stoptime	Stop Timers	RW	0b0
[8:6]	cause	Cause	RW	0b0
[5]	reserved_2	Reserved	RO	0b0
[4]	mprven	Modify Privilege Enable	RW	0b0
[3]	nmip	Non-Maskable Interrupt Pending	RO	0b0
[2]	step	Step	RW	0b0
[1:0]	prv	Privilege level	RW	0b0

Debug Version (xdebugver)
Shows the version of the debug support.

Table 26: The following table shows the bitfield encoding

Value	Name	Description
0b0000	no_ext_debug	There is no external debug support.
0b0100	ext_debug_spec	External debug support exists as it is described in the riscv-debug-release document.
0b1111	ext_debug_no_spec	There is external debug support, but it does not conform to any available version of the riscv-debug-release spec.

Environment Breakpoint M-mode (ebreakm)
Shows the behvior of the ebreak instruction in machine mode.

Table 27: The following table shows the bitfield encoding

Value	Name	Description
0	break_as_spec	ebreak instructions in M-mode behave as described in the Privileged Spec.
1	break_to_debug	ebreak instructions in M-mode enter Debug Mode.

Environment Breakpoint S-mode (`ebreaks`)
Shows the behvior of the `ebreak` instruction in supervisor mode.

Table 28: The following table shows the bitfield encoding

Value	Name	Description
0	break_as_spec	`ebreak` instructions in S-mode behave as described in the Privileged Spec.
1	break_to_debug	`ebreak` instructions in S-mode enter Debug Mode.

Environment Breakpoint U-mode (`ebreaku`)
Shows the behvior of the `ebreak` instruction in user mode.

Table 29: The following table shows the bitfield encoding

Value	Name	Description
0	break_as_spec	`ebreak` instructions in U-mode behave as described in the Privileged Spec.
1	break_to_debug	`ebreak` instructions in U-mode enter Debug Mode.

Stepping Interrupt Enable (`stepie`)
Enables/disables interrupts for single stepping.

The debugger must not change the value of this bit while the hart is running.

Table 30: The following table shows the bitfield encoding

Value	Name	Description
0	disabled	Interrupts are disabled during single stepping.
1	enabled	Interrupts are enabled during single stepping.

Stop Counters (`stopcount`)
Starts/stops incrementing counters in debug mode.

Table 31: The following table shows the bitfield encoding

Value	Name	Description
0	increment_counters	Increment counters as usual.
1	dont_increment_counters	Don't increment any counters while in Debug Mode or on `ebreak` instructions that cause entry into Debug Mode.

Stop Timers (`stoptime`)
Starts/stops incrementing timers in debug mode.

Table 32: The following table shows the bitfield encoding

Value	Name	Description
0	increment_timers	Increment timers as usual.
1	dont_increment_timers	Don't increment any hart-local timers while in Debug Mode.

Cause (`cause`)
Explains why Debug Mode was entered.

When there are multiple reasons to enter Debug Mode in a single cycle, hardware sets `cause` to the cause with the highest priority.

Table 33: The following table shows the bitfield encoding

Value	Name	Description
0b001	ebreak_instruction	An `ebreak` instruction was executed. (priority 3)
0b010	trigger_module	The Trigger Module caused a breakpoint exception. (priority 4, highest)
0b011	debugger_request	The debugger requested entry to Debug Mode using `haltreq`. (priority 1)
0b100	single_step	The hart single stepped because `step` was set. (priority 0, lowest)
0b101	reset_halt	The hart halted directly out of reset due to `resethaltreq`. It is also acceptable to report 3 when this happens. (priority 2)

Modify Privilege Enable (`mprven`)
Enables/disables the modify privilege setting in debug mode.

Table 34: The following table shows the bitfield encoding

Value	Name	Description
0	disable_mprv	MPRV in `mstatus` is ignored in Debug Mode.
1	enable_mprv	MPRV in `mstatus` takes effect in Debug Mode.

Non-Maskable Interrupt Pending (`nmip`)
When set, there is a Non-Maskable-Interrupt (NMI) pending for the hart.

Step (`step`)
When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. If the instruction does not complete due to an exception, the hart will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set. The debugger must not change the value of this bit while the hart is running.

Privilege level (`prv`)
Contains the privilege level the hart was operating in when Debug Mode was entered. A debugger can change this value to change the hart's privilege level when exiting Debug Mode.

Table 35: The following table shows the bitfield encoding

Value	Name	Description
0b00	User	
0b01	Supervisor	
0b11	Machine	

Debug PC (`dpc`)

Address Offset
0x7B1

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description

Upon entry to debug mode, dpc is updated with the virtual address of the next instruction to be executed.

When resuming, the hart's PC is updated to the virtual address stored in dpc. A debugger may write dpc to change where the hart resumes.

Bits	Name	Display Name	Access Type	Reset
[31:0]	dpc		RW	0b0

dpc

The dpc behavior is described in more detail in the table below.

Cause	Virtual Address in DPC
ebreak	Address of the ebreak instruction.
single step	Address of the instruction that would be executed next if no debugging was going on. Ie. pc + 4 for 32-bit instructions that don't change program flow, the destination PC on taken jumps/branches, etc.
trigger module	If timing is 0, the address of the instruction which caused the trigger to fire. If timing is 1, the address of the next instruction to be executed at the time that debug mode was entered.
halt request	Address of the next instruction to be executed at the time that debug mode was entered.

Debug Scratch Register (dscratch[2])

Address Offset

0x7B2 [+ i*0x1]

Width (bits)

32

Access Type

RW

Reset Value

0x00000000

Description

Optional scratch register. A debugger must not write to this register unless hartinfo explicitly mentions it.

Bits	Name	Display Name	Access Type	Reset
[31:0]	dscratch		RW	0b0

ftran

Address Offset
0x800

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Floating Point Custom CSR

Bits	Name	Display Name	Access Type	Reset
[31:7]	reserved_0	Reserved	RO	0b0
[6:0]	ftran		RW	0b0

ftran
Floating Point Custom CSR

M-mode Cycle counter (mcycle)

Address Offset
0xB00

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Counts the number of clock cycles executed by the processor core on which the hart is running.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Count (count)
Counts the number of clock cycles executed by the processor core.

Machine Instruction Retired counter (`minstret`)

Address Offset
0xB02

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Counts the number of instructions the hart has retired.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

Count (`count`)
Counts the number of instructions the hart has retired.

L1 Inst Cache Miss (`ml1_icache_miss`)

Address Offset
0xB03

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

L1 Data Cache Miss (`ml1_dcache_miss`)

Address Offset
0xB04

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

ITLB Miss (mitlb_miss)

Address Offset
0xB05

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

DTLB Miss (mdtlb_miss)

Address Offset
0xB06

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Loads (`mload`)

Address Offset
0xB07

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

Stores (`mstore`)

Address Offset
0xB08

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

Taken Exceptions (`mexception`)

Address Offset
0xB09

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Exception Return (mexception_ret)

Address Offset
0xB0A

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Software Change of PC (mbranch_jump)

Address Offset
0xB0B

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Procedure Call (mcall)

Address Offset
0xB0C

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Procedure Return (mret)

Address Offset
0xB0D

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Branch mis-predicted (mmis_predict)

Address Offset
0xB0E

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Scoreboard Full (`msb_full`)

Address Offset
0xB0F

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

Instruction Fetch Queue Empty (`mif_empty`)

Address Offset
0xB10

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RW	0b0

Upper 32-bits of M-mode Cycle counter (`mcycleh`)

Address Offset
0xB80

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Counts the number of clock cycles executed by the processor core on which the hart is running.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Count (count)
Counts the number of clock cycles executed by the processor core.

Upper 32-bits of Machine Instruction Retired counter (minstreth)

Address Offset
0xB82

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Counts the number of instructions the hart has retired.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Count (count)
Counts the number of instructions the hart has retired.

Upper 32-bits of Machine Hardware Performance Monitoring Counter (mhpmcounterh[6])

Address Offset
0xB83 [+ i*0x1]

Width (bits)
32

Access Type
RW

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RW	0b0

Cycle counter (cycle)

Address Offset
0xC00

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Cycle counter for RDCYCLE instruction.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Timer (time)

Address Offset
0xC01

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Timer for RDTIME instruction.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Instruction Retired counter (instret)

Address Offset
0xC02

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Instructions-retired counter for RDINSTRET instruction

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

L1 Inst Cache Miss (l1_icache_miss)

Address Offset
0xC03

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

L1 Data Cache Miss (l1_dcache_miss)

Address Offset
0xC04

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

ITLB Miss (itlb_miss)

Address Offset
0xC05

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

DTLB Miss (dtlb_miss)

Address Offset
0xC06

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Loads (load)

Address Offset
0xC07

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Stores (`store`)

Address Offset
0xC08

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RO	0b0

Taken Exceptions (`exception`)

Address Offset
0xC09

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RO	0b0

Exception Return (`exception_ret`)

Address Offset
0xC0A

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Software Change of PC (branch_jump)

Address Offset
0xC0B

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Procedure Call (call)

Address Offset
0xC0C

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Procedure Return (ret)

Address Offset
0xC0D

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Branch mis-predicted (mis_predict)

Address Offset
0xC0E

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Scoreboard Full (sb_full)

Address Offset
0xC0F

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Instruction Fetch Queue Empty (`if_empty`)

Address Offset
0xC10

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Hardware performance event counter.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RO	0b0

Upper 32-bits of Cycle counter (`cycleh`)

Address Offset
0xC80

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Cycle counter for RDCYCLE instruction.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`count`	Count	RO	0b0

Upper 32-bit of Timer (`timeh`)

Address Offset
0xC81

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Timer for RDTIME instruction.

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Upper 32-bits of Instruction Retired counter (instreth)

Address Offset
0xC82

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Instructions-retired counter for RDINSTRET instruction

Bits	Name	Display Name	Access Type	Reset
[31:0]	count	Count	RO	0b0

Machine Vendor ID (mvendorid)

Address Offset
0xF11

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
This register provids the JEDEC manufacturer ID of the provider of the core.

Bits	Name	Display Name	Access Type	Reset
[31:7]	bank	Bank	RO	0b0
[6:0]	offset	Offset	RO	0b0

Bank (bank)
Contain encoding for number of one-byte continuation codes discarding the parity bit.

Offset (offset)
Contain encording for the final byte discarding the parity bit.

Machine Architecture ID (marchid)

Address Offset
0xF12

Width (bits)
32

Access Type
RO

Reset Value
0x00000003

Description
This register encodes the base microarchitecture of the hart.

Bits	Name	Display Name	Access Type	Reset
[31:0]	architecture_id	Architecture ID	RO	0b11

Architecture ID (architecture_id)
Provide Encoding the base microarchitecture of the hart.

Machine Implementation ID (mimpid)

Address Offset
0xF13

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
Provides a unique encoding of the version of the processor implementation.

Bits	Name	Display Name	Access Type	Reset
[31:0]	implementation	Implementation	RO	0b0

Implementation (implementation)
Provides unique encoding of the version of the processor implementation.

Machine Hardware Thread ID (mhartid)

Address Offset
0xF14

Width (bits)
32

Access Type
RO

Reset Value
0x00000000

Description
This register contains the integer ID of the hardware thread running the code.

Bits	Name	Display Name	Access Type	Reset
[31:0]	`hart_id`	Hart ID	RO	0b0

Hart ID (`hart_id`)
Contains the integer ID of the hardware thread running the code.

2.3.6 AXI

Introduction

In this chapter, we describe in detail the restriction that apply to the supported features.

In order to understand how the AXI memory interface behaves in CVA6, it is necessary to read the AMBA AXI and ACE Protocol Specification (https://developer.arm.com/documentation/ihi0022/hc) and this chapter.

About the AXI4 protocol

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between Manager and Subordinate components.

The AXI protocol features are:

- It is suitable for high-bandwidth and low-latency designs.
- High-frequency operation is provided, without using complex bridges.
- The protocol meets the interface requirements of a wide range of components.
- It is suitable for memory controllers with high initial access latency.
- Flexibility in the implementation of interconnect architectures is provided.
- It is backward-compatible with AHB and APB interfaces.

The key features of the AXI protocol are:

- Separate address/control and data phases.
- Support for unaligned data transfers, using byte strobes.
- Uses burst-based transactions with only the start address issued.
- Separate read and write data channels, that can provide low-cost Direct Memory Access (DMA).
- Support for issuing multiple outstanding addresses.
- Support for out-of-order transaction completion.
- Permits easy addition of register stages to provide timing closure.

The present specification is based on :

https://developer.arm.com/documentation/ihi0022/hc

AXI4 and CVA6

The AXI bus protocol is used with the CVA6 processor as a memory interface. Since the processor is the one that initiates the connection with the memory, it will have a manager interface to send requests to the subordinate, which will be the memory.

Features supported by CVA6 are the ones in the AMBA AXI4 specification and the Atomic Operation feature from AXI5. With restriction that apply to some features.

This doesn't mean that all the full set of signals available on an AXI interface are supported by the CVA6. Nevertheless, all required AXI signals are implemented.

Supported AXI4 features are defined in AXI Protocol Specification sections: A3, A4, A5, A6 and A7.

Supported AXI5 feature are defined in AXI Protocol Specification section: E1.1.

Signal Description (Section A2)

This section introduces the AXI memory interface signals of CVA6. Most of the signals are supported by CVA6, the tables summarizing the signals identify the exceptions.

In the following tables, the *Src* column tells whether the signal is driven by Manager ou Subordinate.

The AXI required and optional signals, and the default signals values that apply when an optional signal is not implemented are defined in AXI Protocol Specification section A9.3.

Global signals (Section A2.1)

Table 2.1 shows the global AXI memory interface signals.

Signal	Src	Description
ACLK	Clock source	Global clock signal. Synchronous signals are sampled on the rising edge of the global clock.
WDATA	Reset source	Global reset signal. This signal is active-LOW.

Write address channel signals (Section A2.2)

Table 2.2 shows the AXI memory interface write address channel signals. Unless the description indicates otherwise, a signal can take any parameter if is supported.

Signal	Src	Support	Description
AWID	M	Yes (optional)	Identification tag for a write transaction. CVA6 gives the id depending on the type of transaction. See *Transaction Identifiers (Section A5)*.
AWADDR	M	Yes	The address of the first transfer in a write transaction.
AWLEN	M	Yes (optional)	Length, the exact number of data transfers in a write transaction. This information determines the number of data transfers associated with the address. All write transactions performed by CVA6 are of length 1. (AWLEN = 0b00000000)
AWSIZE	M	Yes (optional)	Size, the number of bytes in each data transfer in a write transaction See *Address structure (Section A3.4.1)*.
AWBURST	M	Yes (optional)	Burst type, indicates how address changes between each transfer in a write transaction. All write transactions performed by CVA6 are of burst type INCR. (AWBURST = 0b01)
AWLOCK	M	Yes (optional)	Provides information about the atomic characteristics of a write transaction.
AWCACHE	M	Yes (optional)	Indicates how a write transaction is required to progress through a system. The subordinate is always of type Device Non-bufferable. (AWCACHE = 0b0000)
AWPROT	M	Yes	Protection attributes of a write transaction: privilege, security level, and access type. The value of AWPROT is always 0b000.

Write data channel signals (Section A2.3)

Table 2.3 shows the AXI write data channel signals. Unless the description indicates otherwise, a signal can take any parameter if is supported.

Signal	Src	Support	Description
WID	M	Yes (optional)	The ID tag of the write data transfer. CVA6 gives the id depending on the type of transaction. See *Transaction Identifiers (Section A5)*.
WDATA	M	Yes	Write data.
WSTRB	M	Yes (optional)	Write strobes, indicate which byte lanes hold valid data See *Data read and write structure: Write strobes (Section A3.4.4)*.
WLAST	M	Yes	Indicates whether this is the last data transfer in a write transaction.
WUSER	M	Yes (optional)	User-defined extension for the write data channel.
WVALID	M	Yes	Indicates that the write data channel signals are valid.
WREADY	S	Yes	Indicates that a transfer on the write data channel can be accepted.

Write Response Channel signals (Section A2.4)

Table 2.4 shows the AXI write response channel signals. Unless the description indicates otherwise, a signal can take any parameter if is supported.

Signal	Src	Support	Description
BID	S	Yes (optional)	Identification tag for a write response. CVA6 gives the id depending on the type of transaction. See *Transaction Identifiers (Section A5)*.
BRESP	S	Yes	Write response, indicates the status of a write transaction. See *Read and write response structure (Section A3.4.5)*.
BUSER	S	No (optional)	User-defined extension for the write response channel. BUSER= 0b00
BVALID	S	Yes	Indicates that the write response channel signals are valid.
BREADY	M	Yes	Indicates that a transfer on the write response channel can be accepted.

Read address channel signals (Section A2.5)

Table 2.5 shows the AXI read address channel signals. Unless the description indicates otherwise, a signal can take any parameter if is supported.

Signal	Src	Support	Description
ARID	M	Yes (optional)	Identification tag for a read transaction. CVA6 gives the id depending on the type of transaction. See *Transaction Identifiers (Section A5)*.
ARADDR	M	Yes	The address of the first transfer in a read transaction.
ARLEN	M	Yes (optional)	Length, the exact number of data transfers in a read transaction. This information determines the number of data transfers associated with the address. All read transactions performed by CVA6 are of length less or equal to ICACHE_LINE_WIDTH/64.
ARSIZE	M	Yes (optional)	Size, the number of bytes in each data transfer in a read transaction See *Address structure (Section A3.4.1)*.
ARBURST	M	Yes (optional)	Burst type, indicates how address changes between each transfer in a read transaction. All Read transactions performed by CVA6 are of burst type INCR. (ARBURST = 0b01)
ARLOCK	M	Yes (optional)	Provides information about the atomic characteristics of a read transaction.
ARCACHE	M	Yes (optional)	Indicates how a read transaction is required to progress through a system. The memory is always of type Device Non-bufferable. (ARCACHE = 0b0000)
ARPROT	M	Yes	Protection attributes of a read transaction: privilege, security level, and access type. The value of ARPROT is always 0b000.
ARQOS	M		

Read data channel signals (Section A2.6)

Table 2.6 shows the AXI read data channel signals. Unless the description indicates otherwise, a signal can take any parameter if is supported.

Signal	Src	Support	Description
RID	S	Yes (optional)	The ID tag of the read data transfer. CVA6 gives the id depending on the type of transaction. See *Transaction Identifiers (Section A5)*.
RDATA	S	Yes	Read data.
RLAST	S	Yes	Indicates whether this is the last data transfer in a read transaction.
RUSER	S	Yes (optional)	User-defined extension for the read data channel. Not supported. (RUSER= 0b00)
RVALID	S	Yes	Indicates that the read data channel signals are valid.
RREADY	M	Yes	Indicates that a transfer on the read data channel can be accepted.

Single Interface Requirements: Transaction structure (Section A3.4)

This section describes the structure of transactions. The following sections define the address, data, and response structures

Address structure (Section A3.4.1)

The AXI protocol is burst-based. The Manager begins each burst by driving control information and the address of the first byte in the transaction to the Subordinate. As the burst progresses, the Subordinate must calculate the addresses of subsequent transfers in the burst.

Burst length

The burst length is specified by:

- **ARLEN[7:0]**, for read transfers
- **AWLEN[7:0]**, for write transfers

The burst length for AXI4 is defined as:

```
Burst_Length = AxLEN[3:0] + 1
```

CVA6 has some limitation governing the use of bursts:

- *All read transactions performed by CVA6 are of burst length less or equal to ICACHE_LINE_WIDTH/64.*
- *All write transactions performed by CVA6 are of burst length equal to 1.*

Burst size

The maximum number of bytes to transfer in each data transfer, or beat, in a burst, is specified by:

- **ARSIZE[2:0]**, for read transfers
- **AWSIZE[2:0]**, for write transfers

AXI DATA WIDTH used by CVA6 is 64-bit. For that, the maximum value can be taking by AXSIZE is 3 (8 bytes by transfer).

Burst type

The AXI protocol defines three burst types:

- **FIXED**
- **INCR**
- **WRAP**

The burst type is specified by:

- **ARBURST[1:0]**, for read transfers
- **AWBURST[1:0]**, for write transfers

All transactions performed by CVA6 are of burst type INCR. (AXBURST = 0b01)

Data read and write structure: Write strobes (Section A3.4.4)

The WSTRB[n:0] signals when HIGH, specify the byte lanes of the data bus that contain valid information. There is one write strobe for each 8 bits of the write data bus, therefore WSTRB[n] corresponds to WDATA[(8n)+7: (8n)].

AXI DATA WIDTH used by CVA6 is 64-bit. Therefore, Write Strobe width is equal to eight (n = 7).

Read and write response structure (Section A3.4.5)

The AXI protocol provides response signaling for both read and write transactions:

- For read transactions, the response information from the Subordinate is signaled on the read data channel.
- For write transactions, the response information is signaled on the write response channel.

CVA6 does not consider the responses sent by the memory except in the exclusive Access (XRESP[1:0] = 0b01).

Transaction Attributes: Memory types (Section A4)

This section describes the attributes that determine how a transaction should be treated by the AXI subordinate that is connected to the CVA6.

AXCACHE always take 0b0000. The subordinate should be a Device Non-bufferable.

The required behavior for Device Non-bufferable memory is:

- The write response must be obtained from the final destination.
- Read data must be obtained from the final destination.
- Transactions are Non-modifiable.
- Reads must not be prefetched. Writes must not be merged.

Transaction Identifiers (Section A5)

The AXI protocol includes AXI ID transaction identifiers. A Manager can use these to identify separate transactions that must be returned in order.

The CVA6 identify each type of transaction with a specific ID

For read transaction id can be 0 or 1.

For write transaction id = 1.

For Atomic operation id = 3. This ID must be sent in the write channels and also in the read channel if the transaction performed requires response data.

AXI Ordering Model (Section A6)

AXI ordering model overview (Section A6.1)

The AXI ordering model is based on the use of the transaction identifier, which is signaled on ARID or AWID.

Transaction requests on the same channel, with the same ID and destination are guaranteed to remain in order.

Transaction responses with the same ID are returned in the same order as the requests were issued.

Write transaction requests, with the same destination are guaranteed to remain in order. Because all write transaction performed by CVA6 have the same ID.

CVA6 can perform multiple outstanding write address transactions.

CVA6 cannot perform a Read transaction and a Write one at the same time. Therefore there no ordering problems between Read and write transactions.

The ordering model does not give any ordering guarantees between:

- Transactions from different Managers
- Read Transactions with different IDs
- Transactions to different Memory locations

If the CVA6 requires ordering between transactions that have no ordering guarantee, the Manager must wait to receive a response to the first transaction before issuing the second transaction.

Memory locations and Peripheral regions (Section A6.2)

The address map in AMBA is made up of Memory locations and Peripheral regions. But the AXI is associated to the memory interface of CVA6.

A Memory location has all of the following properties:

- A read of a byte from a Memory location returns the last value that was written to that byte location.
- A write to a byte of a Memory location updates the value at that location to a new value that is obtained by a subsequent read of that location.
- Reading or writing to a Memory location has no side-effects on any other Memory location.
- Observation guarantees for Memory are given for each location.
- The size of a Memory location is equal to the single-copy atomicity size for that component.

Transactions and ordering (Section A6.3)

A transaction is a read or a write to one or more address locations. The locations are determined by AxADDR and any relevant qualifiers such as the Non-secure bit in AxPROT.

- Ordering guarantees are given only between accesses to the same Memory location or Peripheral region.
- A transaction to a Peripheral region must be entirely contained within that region.
- A transaction that spans multiple Memory locations has multiple ordering guarantees.

Transaction performed by CVA6 is of type Device. Because AxCACHE[1] deasserted.

Device transactions can be used to access Peripheral regions or Memory locations.

A write transaction performed by CVA6 is Non-bufferable (It is possible to send an early response to Bufferable write). Because AxCACHE[0] deasserted.

Ordered write observation (Section A6.8)

To improve compatibility with interface protocols that support a different ordering model, a Subordinate interface can give stronger ordering guarantees for write transactions. A stronger ordering guarantee is known as Ordered Write Observation.

The CVA6 AXI interface exhibits Ordered Write Observation, so the Ordered_Write_Observation property is True.

An interface that exhibits Ordered Write Observation gives guarantees for write transactions that are not dependent on the destination or address:

- A write W1 is guaranteed to be observed by a write W2, where W2 is issued after W1, from the same Manager, with the same ID.

Atomic transactions (Section E1.1)

AMBA 5 introduces Atomic transactions, which perform more than just a single access and have an operation that is associated with the transaction. Atomic transactions enable sending the operation to the data, permitting the operation to be performed closer to where the data is located. Atomic transactions are suited to situations where the data is located a significant distance from the agent that must perform the operation.

CVA6 support just the AtomicLoad and AtomicSwap transaction. So AWATOP[5:4] can be 00, 10 or 11

CVA6 perform only little-endian operation. So AWATOP[3] = 0

For AtomicLoad, CVA6 support all arithmetic operations encoded on the lower-order AWATOP[2:0] signals

2.3.7 Glossary

- **VLEN**: Virtual address lengh
- **XLEN**: RISC-V processor data lengh
- **ALU**: Arithmetic/Logic Unit
- **ASIC**: Application-Specific Integrated Circuit
- **Byte**: 8-bit data item
- **CPU**: Central Processing Unit, processor
- **CSR**: Control and Status Register
- **Custom extension**: Non-Standard extension to the RISC-V base instruction set (RISC-V Instruction Set Manual, Volume I: User-Level ISA)
- **EXE**: Instruction Execute
- **FPGA**: Field Programmable Gate Array
- **FPU**: Floating Point Unit
- **Halfword**: 16-bit data item
- **Halfword aligned address**: An address is halfword aligned if it is divisible by 2
- **ID**: Instruction Decode
- **IF**: Instruction Fetch

- **ISA**: Instruction Set Architecture
- **KGE**: kilo gate equivalents (NAND2)
- **LSU**: Load Store Unit
- **M-Mode**: Machine Mode (RISC-V Instruction Set Manual, Volume II: Privileged Architecture)
- **OBI**: Open Bus Interface
- **PC**: Program Counter
- **PULP platform**: Parallel Ultra Low Power Platform (<https://pulp-platform.org>)
- **RV32C**: RISC-V Compressed (C extension)
- **RV32F**: RISC-V Floating Point (F extension)
- **SIMD**: Single Instruction/Multiple Data
- **Standard extension**: Standard extension to the RISC-V base instruction set (RISC-V Instruction Set Manual, Volume I: User-Level ISA)
- **WARL**: Write Any Values, Reads Legal Values
- **WB**: Write Back of instruction results
- **WLRL**: Write/Read Only Legal Values
- **Word**: 32-bit data item
- **Word aligned address**: An address is word aligned if it is divisible by 4
- **WPRI**: Reserved Writes Preserve Values, Reads Ignore Values