Design Documentation for CV32A65X architecture

Table of Contents

Editor: **Jean Roch Coulon**

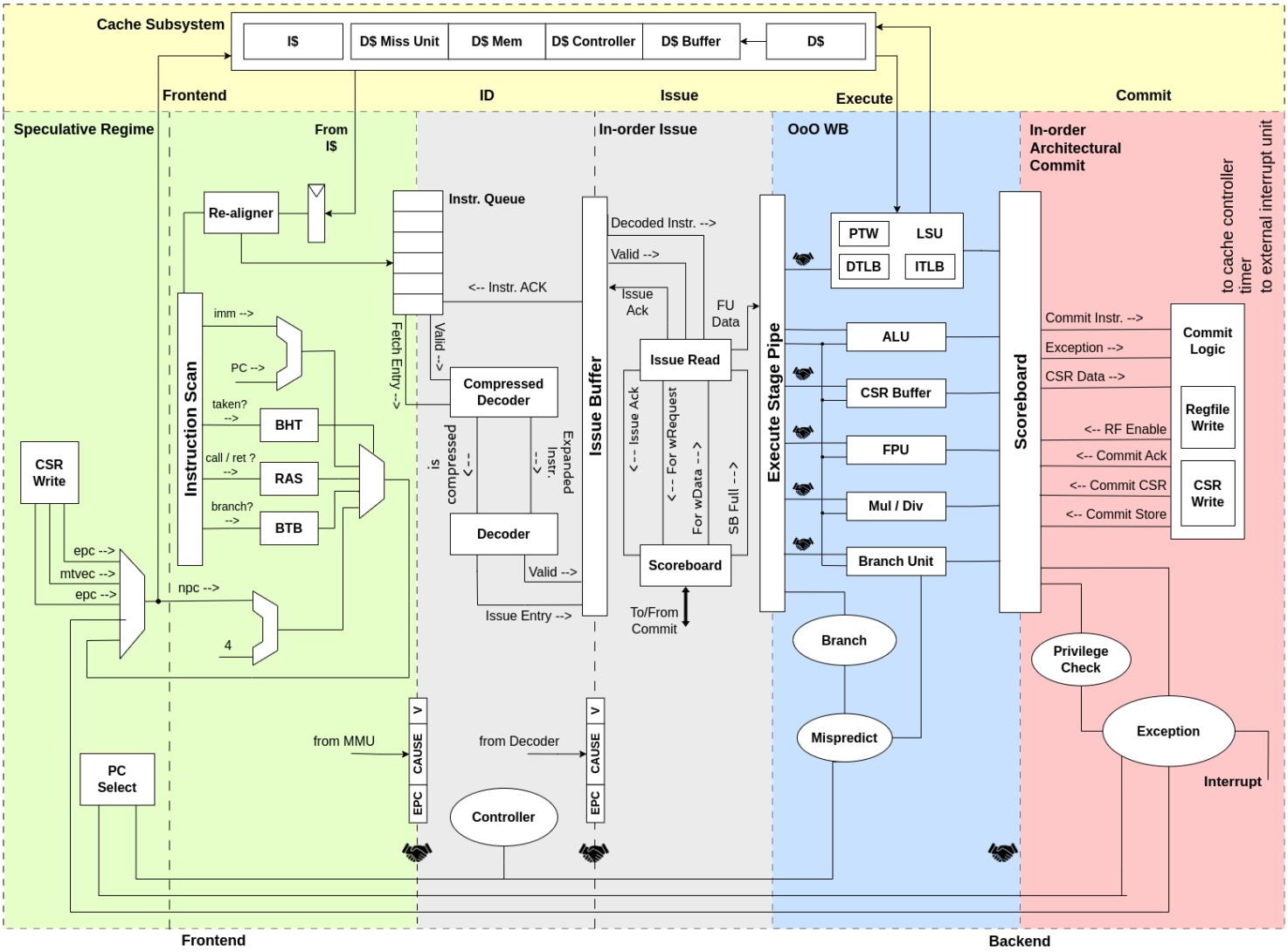
1. Introduction

The OpenHW Group uses semantic versioning (https://semver.org/) to describe the release status of its IP. This document describes the CV32A65X configuration version of CVA6. This intends to be the first formal release of CVA6.

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.

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.



1.1. License

1.2. 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].

1.3. 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

1.4. Contributors

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

[TO BE COMPLETED]

2. Subsystem

2.1. Global functionality

The CVA6 is a subsystem composed of the modules and protocol interfaces as illustrated 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 CVA6 implements a 6-stage pipeline composed of PC Generation, Instruction Fetch, Instruction Decode, Issue stage, Execute stage and Commit stage. At least 6 cycles are needed to execute one instruction.

2.2. Connection with other sub-systems

The submodule is connected to :

- NOC interconnect provides memory content
- COPROCESSOR connects through CV-X-IF coprocessor interface protocol
- TRACER provides support for verification
- TRAP provides traps inputs

2.3. Parameter configuration

Table 1. cv32a65x parameter configuration

Name	**description**	**description**
XLEN	General Purpose Register Size (in bits)	32
RVA	Atomic RISC-V extension	False
RVB	Bit manipulation RISC-V extension	True
RVV	Vector RISC-V extension	False
RVC	Compress RISC-V extension	True
RVH	Hypervisor RISC-V extension	False
RVZCB	Zcb RISC-V extension	True
RVZCMP	Zcmp RISC-V extension	False
RVZiCond	Zicond RISC-V extension	False
RVZicntr	Zicntr RISC-V extension	False
RVZihpm	Zihpm RISC-V extension	False
RVF	Floating Point	False
RVD	Floating Point	False
XF16	Non standard 16bits Floating Point extension	False
XF16ALT	Non standard 16bits Floating Point Alt extension	False
XF8	Non standard 8bits Floating Point extension	False
XFVec	Non standard Vector Floating Point extension	False
PerfCounterEn	Perf counters	False
MmuPresent	MMU	False
RVS	Supervisor mode	False
RVU	User mode	False

Name	description	description
DebugEn	Debug support	False
DmBaseAddress	Base address of the debug module	0x0
HaltAddress	Address to jump when halt request	0x800
ExceptionAddress	Address to jump when exception	0x808
TvalEn	Tval Support Enable	False
DirectVecOnly	MTVEC CSR supports only direct mode	True
NrPMPEntries	PMP entries number	8
PMPCfgRstVal	PMP CSR configuration reset values	[0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
PMPAddrRstVal	PMP CSR address reset values	[0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
PMPEntryReadOnly	PMP CSR read-only bits	0
NrNonIdempotentRules	PMA non idempotent rules number	0
NonIdempotentAddrBase	PMA NonIdempotent region base address	[0b0, 0b0]
NonIdempotentLength	PMA NonIdempotent region length	[0b0, 0b0]
NrExecuteRegionRules	PMA regions with execute rules number	0
ExecuteRegionAddrBase	PMA Execute region base address	[0x80000000, 0x10000, 0x0]
ExecuteRegionLength	PMA Execute region address base	[0x40000000, 0x10000, 0x1000]
NrCachedRegionRules	PMA regions with cache rules number	1
CachedRegionAddrBase	PMA cache region base address	[0x80000000]
CachedRegionLength	PMA cache region rules	[0x40000000]
CvxifEn	CV-X-IF coprocessor interface enable	True
NOCType	NOC bus type	config_pkg::NOC_TYPE_AXI4_ATOP
AxiAddrWidth	AXI address width	64
AxiDataWidth	AXI data width	64
AxiIdWidth	AXI ID width	4
AxiUserWidth	AXI User width	32
AxiBurstWriteEn	AXI burst in write	False
MemTidWidth	TODO	4
IcacheByteSize	Instruction cache size (in bytes)	2048

Name	description	description
IcacheSetAssoc	Instruction cache associativity (number of ways)	2
IcacheLineWidth	Instruction cache line width	128
DCacheType	Cache Type	config_pkg::HPDCACHE
DcacheIdWidth	Data cache ID	1
DcacheByteSize	Data cache size (in bytes)	2028
DcacheSetAssoc	Data cache associativity (number of ways)	2
DcacheLineWidth	Data cache line width	128
DataUserEn	User field on data bus enable	1
WtDcacheWbufDepth	Write-through data cache write buffer depth	8
FetchUserEn	User field on fetch bus enable	1
FetchUserWidth	Width of fetch user field	32
FpgaEn	Is FPGA optimization of CV32A6	False
TechnoCut	Is Techno Cut instanciated	True
SuperscalarEn	Enable superscalar* with 2 issue ports and 2 commit ports.	True
NrCommitPorts	Number of commit ports. Forced to 2 if SuperscalarEn.	1
NrLoadPipeRegs	Load cycle latency number	0
NrStorePipeRegs	Store cycle latency number	0
NrScoreboardEntries	Scoreboard length	8
NrLoadBufEntries	Load buffer entry buffer	2
MaxOutstandingStores	Maximum number of outstanding stores	7
RASDepth	Return address stack depth	2
BTBEntries	Branch target buffer entries	0
BHTEntries	Branch history entries	32
InstrTlbEntries	MMU instruction TLB entries	2
DataTlbEntries	MMU data TLB entries	2
UseSharedTlb	MMU option to use shared TLB	True
SharedTlbDepth	MMU depth of shared TLB	64

2.4. IO ports

*Table 2. **cva6 module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
boot_addr_i	in	Reset boot address	SUBSYSTEM	logic[CVA6Cfg.VLEN-1:0]
hart_id_i	in	Hard ID reflected as CSR	SUBSYSTEM	logic[CVA6Cfg.XLEN-1:0]
irq_i	in	Level sensitive (async) interrupts	SUBSYSTEM	logic[1:0]
ipi_i	in	Inter-processor (async) interrupt	SUBSYSTEM	logic
time_irq_i	in	Timer (async) interrupt	SUBSYSTEM	logic
cvxif_req_o	out	CVXIF request	SUBSYSTEM	cvxif_req_t
cvxif_resp_i	in	CVXIF response	SUBSYSTEM	cvxif_resp_t
noc_req_o	out	noc request, can be AXI or OpenPiton	SUBSYSTEM	noc_req_t
noc_resp_i	in	noc response, can be AXI or OpenPiton	SUBSYSTEM	noc_resp_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As DebugEn = False,

- `debug_req_i` input is tied to 0

As IsRVFI = 0,

- `rvfi_probes_o` output is tied to 0

3. Functionality

3.1. Instructions

The next subchapter lists the extensions implemented in CV32A65X. By configuration, we can enable/disable the extensions. CV32A65X supports the extensions described in the next subchapters.

3.2. isa

3.2.1. Instructions

Subset Name	Name	Description
I	RV32I Base Integer Instructions	the base integer instruction set, also known as the 'RV32I' or 'RV64I' instruction set , depending on the address space size, provides the core functionality required for general-purpose computing .it includes instructions for arithmetic, logical, and control operations, as well as memory accessand manipulation
M	RV32M Multiplication and Division Instructions	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.
C	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
Zicsr	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.
Zifencei	RVZifencei Instruction Fetch Fence	FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart.Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to it instruction fetches.

Subset Name	Name	Description
Zcb	RV32Zcb Code Size Reduction Instructions	Zcb belongs to the group of extensions called RISC-V Code Size Reduction Extension (Zc*). Zc* has become the superset of the Standard C extension adding more 16-bit instructions to the ISA. Zcb includes the 16-bit version of additional Integer (I), Multiply (M), and Bit-Manipulation (Zbb) Instructions. All the Zcb instructions require at least standard C extension support as a prerequisite, along with M and Zbb extensions for the 16-bit version of the respective instructions.
Zba	RVZba Address generation instructions	The Zba instructions can be used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and XLEN-sized indices: a shifted index is added to a base address. The shift and add instructions do a left shift of 1, 2, or 3 because these are commonly found in real-world code and because they can be implemented with a minimal amount of additional hardware beyond that of the simple adder. This avoids lengthening the critical path in implementations. While the shift and add instructions are limited to a maximum left shift of 3, the slli instruction (from the base ISA) can be used to perform similar shifts for indexing into arrays of wider elements. The slli.uw added in this extension can be used when the index is to be interpreted as an unsigned word.
Zbb	RVZbb Basic bit-manipulation	The bit-manipulation (bitmanip) extension collection is comprised of several component extensions to the base RISC-V architecture that are intended to provide some combination of code size reduction, performance improvement, and energy reduction. While the instructions are intended to have general use, some instructions are more useful in some domains than others. Hence, several smaller bitmanip extensions are provided. Each of these smaller extensions is grouped by common function and use case, and each has its own Zb*-extension name.
Zbc	RVZbc Carry-less multiplication	Carry-less multiplication is the multiplication in the polynomial ring over GF(2).clmul produces the lower half of the carry-less product and clmulh produces the upper half of the 2×XLEN carry-less product.clmulr produces bits 2×XLEN-2:XLEN-1 of the 2×XLEN carry-less product.
Zbs	RVZbs Single bit Instructions	The single-bit instructions provide a mechanism to set, clear, invert, or extract a single bit in a register. The bit is specified by its index.
Zicntr	Zicntr	No info found yet for extension Zicntr

3.2.2. RV32I Base Integer Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
ADDI	addi rd, rs1, imm[11:0]	x[rd] = x[rs1] + sext(imm[11:0])	NONE	NONE	add sign-extended 12-bit immediate to register rs1, and store the result in register rd.	Integer_Register_Immediate_Operations
ANDI	andi rd, rs1, imm[11:0]	x[rd] = x[rs1] & sext(imm[11:0])	NONE	NONE	perform bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.	Integer_Register_Immediate_Operations
ORI	ori rd, rs1, imm[11:0]	x[rd] = x[rs1] | sext(imm[11:0])	NONE	NONE	perform bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.	Integer_Register_Immediate_Operations
XORI	xori rd, rs1, imm[11:0]	x[rd] = x[rs1] ^ sext(imm[11:0])	NONE	NONE	perform bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.	Integer_Register_Immediate_Operations
SLTI	slti rd, rs1, imm[11:0]	if (x[rs1] < sext(imm[11:0])) x[rd] = 1 else x[rd] = 0	NONE	NONE	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.	Integer_Register_Immediate_Operations
SLTIU	sltiu rd, rs1, imm[11:0]	if (x[rs1] <u sext(imm[11:0])) x[rd] = 1 else x[rd] = 0	NONE	NONE	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."	Integer_Register_Immediate_Operations
SLLI	slli rd, rs1, imm[4:0]	x[rd] = x[rs1] << imm[4:0]	NONE	NONE	logical left shift (zeros are shifted into the lower bits).	Integer_Register_Immediate_Operations
SRLI	srli rd, rs1, imm[4:0]	x[rd] = x[rs1] >> imm[4:0]	NONE	NONE	logical right shift (zeros are shifted into the upper bits).	Integer_Register_Immediate_Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
SRAI	srai rd, rs1, imm[4:0]	x[rd] = x[rs1] >>s imm[4:0]	NONE	NONE	arithmetic right shift (the original sign bit is copied into the vacated upper bits).	Integer_Register_Immediate_Operations
LUI	lui rd, imm[19:0]	x[rd] = sext(imm[31:12] << 12)	NONE	NONE	place the immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros.	Integer_Register_Immediate_Operations
AUIPC	auipc rd, imm[19:0]	x[rd] = pc + sext(immediate[31:12] << 12)	NONE	NONE	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.	Integer_Register_Immediate_Operations
ADD	add rd, rs1, rs2	x[rd] = x[rs1] + x[rs2]	NONE	NONE	add rs2 to register rs1, and store the result in register rd.	Integer_Register_Register_Operations
SUB	sub rd, rs1, rs2	x[rd] = x[rs1] - x[rs2]	NONE	NONE	subtract rs2 from register rs1, and store the result in register rd.	Integer_Register_Register_Operations
AND	and rd, rs1, rs2	x[rd] = x[rs1] & x[rs2]	NONE	NONE	perform bitwise AND on register rs1 and rs2 and place the result in rd.	Integer_Register_Register_Operations
OR	or rd, rs1, rs2	x[rd] = x[rs1] | x[rs2]	NONE	NONE	perform bitwise OR on register rs1 and rs2 and place the result in rd.	Integer_Register_Register_Operations
XOR	xor rd, rs1, rs2	x[rd] = x[rs1] ^ x[rs2]	NONE	NONE	perform bitwise XOR on register rs1 and rs2 and place the result in rd.	Integer_Register_Register_Operations
SLT	slt rd, rs1, rs2	if (x[rs1] < x[rs2]) x[rd] = 1 else x[rd] = 0	NONE	NONE	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.	Integer_Register_Register_Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
SLTU	sltu rd, rs1, rs2	if (x[rs1] <u x[rs2]) x[rd] = 1 else x[rd] = 0	NONE	NONE	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.	Integer_Register_Register_Operations
SLL	sll rd, rs1, rs2	x[rd] = x[rs1] << x[rs2]	NONE	NONE	logical left shift (zeros are shifted into the lower bits).	Integer_Register_Register_Operations
SRL	srl rd, rs1, rs2	x[rd] = x[rs1] >> x[rs2]	NONE	NONE	logical right shift (zeros are shifted into the upper bits).	Integer_Register_Register_Operations
SRA	sra rd, rs1, rs2	x[rd] = x[rs1] >>s x[rs2]	NONE	NONE	arithmetic right shift (the original sign bit is copied into the vacated upper bits).	Integer_Register_Register_Operations
JAL	jal rd, imm[20:1]	x[rd] = pc+4; pc += sext(imm[20:1])	NONE	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	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.	Control_Transfer_Operations-Unconditional_Jumps
JALR	jalr rd, rs1, imm[11:0]	t = pc+4; pc = (x[rs1]+sext(imm[11:0]))&~1 ; x[rd] = t	NONE	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	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.	Control_Transfer_Operations-Unconditional_Jumps

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BEQ	beq rs1, rs2, imm[12:1]	if (x[rs1] == x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 and rs2 are equal.	Control_Transfer_Operations-Conditional_Branches
BNE	bne rs1, rs2, imm[12:1]	if (x[rs1] != x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 and rs2 are not equal.	Control_Transfer_Operations-Conditional_Branches

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BLT	blt rs1, rs2, imm[12:1]	if (x[rs1] < x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 less than rs2 (using signed comparison).	Control_Transfer_Operations-Conditional_Branches
BLTU	bltu rs1, rs2, imm[12:1]	if (x[rs1] <u x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 less than rs2 (using unsigned comparison).	Control_Transfer_Operations-Conditional_Branches

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BGE	bge rs1, rs2, imm[12:1]	if (x[rs1] >= x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 is greater than or equal rs2 (using signed comparison).	Control_Transfer_Operations-Conditional_Branches
BGEU	bgeu rs1, rs2, imm[12:1]	if (x[rs1] >=u x[rs2]) pc += sext({imm[12:1], 1'b0}) else pc += 4	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	takes the branch (pc is calculated using signed arithmetic) if registers rs1 is greater than or equal rs2 (using unsigned comparison).	Control_Transfer_Operations-Conditional_Branches
LB	lb rd, imm(rs1)	x[rd] = sext(M[x[rs1] + sext(imm[11:0])][7:0])	NONE	loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.	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.	Load_and_Store_Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
LH	lh rd, imm(rs1)	x[rd] = sext(M[x[rs1] + sext(imm[11:0])][15:0])	NONE	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).	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.	Load_and_Store_Instructions
LW	lw rd, imm(rs1)	x[rd] = sext(M[x[rs1] + sext(imm[11:0])][31:0])	NONE	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).	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.	Load_and_Store_Instructions
LBU	lbu rd, imm(rs1)	x[rd] = zext(M[x[rs1] + sext(imm[11:0])][7:0])	NONE	loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.	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.	Load_and_Store_Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
LHU	lhu rd, imm(rs1)	x[rd] = zext(M[x[rs1] + sext(imm[11:0])][15:0])	NONE	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).	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.	Load_and_Store_Instructions
SB	sb rs2, imm(rs1)	M[x[rs1] + sext(imm[11:0])][7:0] = x[rs2][7:0]	NONE	NONE	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.	Load_and_Store_Instructions
SH	sh rs2, imm(rs1)	M[x[rs1] + sext(imm[11:0])][15:0] = x[rs2][15:0]	NONE	an exception is raised if the memory address isn't aligned (2-byte boundary).	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.	Load_and_Store_Instructions
SW	sw rs2, imm(rs1)	M[x[rs1] + sext(imm[11:0])][31:0] = x[rs2][31:0]	NONE	an exception is raised if the memory address isn't aligned (4-byte boundary).	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.	Load_and_Store_Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
FENCE	fence pre, succ	No operation (nop)	NONE	NONE	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 ®, 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.	Memory_Ordering
ECALL	ecall	RaiseException(EnvironmentCall)	NONE	Raise an Environment Call exception.	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.	Environment_Call_and_Breakpoints

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
EBREAK	ebreak	x[8 + rd'] = sext(x[8 + rd'][7:0])	NONE	NONE	This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits. It also requires Bit-Manipulation (Zbb) extension support.	Environment_Call_and_Breakpoints

3.2.3. RV32M Multiplication and Division Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
MUL	mul rd, rs1, rs2	x[rd] = x[rs1] * x[rs2]	NONE	NONE	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).	Multiplication Operations
MULH	mulh rd, rs1, rs2	x[rd] = (x[rs1] s*s x[rs2]) >>s 32	NONE	NONE	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).	Multiplication Operations
MULHU	mulhu rd, rs1, rs2	x[rd] = (x[rs1] u*u x[rs2]) >>u 32	NONE	NONE	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).	Multiplication Operations
MULHSU	mulhsu rd, rs1, rs2	x[rd] = (x[rs1] s*u x[rs2]) >>s 32	NONE	NONE	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).	Multiplication Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
DIV	div rd, rs1, rs2	x[rd] = x[rs1] /s x[rs2]	NONE	NONE	perform signed integer division of 32 bits by 32 bits (rounding towards zero).	Division Operations
DIVU	divu rd, rs1, rs2	x[rd] = x[rs1] /u x[rs2]	NONE	NONE	perform unsigned integer division of 32 bits by 32 bits (rounding towards zero).	Division Operations
REM	rem rd, rs1, rs2	x[rd] = x[rs1] %s x[rs2]	NONE	NONE	provide the remainder of the corresponding division operation DIV (the sign of rd equals the sign of rs1).	Division Operations
REMU	rem rd, rs1, rs2	x[rd] = x[rs1] %u x[rs2]	NONE	NONE	provide the remainder of the corresponding division operation DIVU.	Division Operations

3.2.4. RV32C Compressed Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.LI	c.li rd, imm[5:0]	x[rd] = sext(imm[5:0])	rd = x0	NONE	loads the sign-extended 6-bit immediate, imm, into register rd.	Integer Computational Instructions
C.LUI	c.lui rd, nzimm[17:12]	x[rd] = sext(nzimm[17:12] << 12)	rd = x0 & rd = x2 & nzimm = 0	NONE	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.	Integer Computational Instructions
C.ADDI	c.addi rd, nzimm[5:0]	x[rd] = x[rd] + sext(nzimm[5:0])	rd = x0 & nzimm = 0	NONE	adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd.	Integer Computational Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.ADDI16SP	c.addi16sp nzimm[9:4]	x[2] = x[2] + sext(nzimm[9:4])	rd != x2 & nzimm = 0	NONE	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.	Integer Computational Instructions
C.ADDI4SPN	c.addi4spn rd', nzimm[9:2]	x[8 + rd'] = x[2] + zext(nzimm[9:2])	nzimm = 0	NONE	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.	Integer Computational Instructions
C.SLLI	c.slli rd, uimm[5:0]	x[rd] = x[rd] << uimm[5:0]	rd = x0 & uimm[5] = 0	NONE	performs a logical left shift (zeros are shifted into the lower bits).	Integer Computational Instructions
C.SRLI	c.srli rd', uimm[5:0]	x[8 + rd'] = x[8 + rd'] >> uimm[5:0]	uimm[5] = 0	NONE	performs a logical right shift (zeros are shifted into the upper bits).	Integer Computational Instructions
C.SRAI	c.srai rd', uimm[5:0]	x[8 + rd'] = x[8 + rd'] >>s uimm[5:0]	uimm[5] = 0	NONE	performs an arithmetic right shift (sign bits are shifted into the upper bits).	Integer Computational Instructions
C.ANDI	c.andi rd', imm[5:0]	x[8 + rd'] = x[8 + rd'] & sext(imm[5:0])	NONE	NONE	computes the bitwise AND of the value in register rd', and the sign-extended 6-bit immediate, then writes the result to rd'.	Integer Computational Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.ADD	c.add rd, rs2	x[rd] = x[rd] + x[rs2]	rd = x0 & rs2 = x0	NONE	adds the values in registers rd and rs2 and writes the result to register rd.	Integer Computational Instructions
C.MV	c.mv rd, rs2	x[rd] = x[rs2]	rd = x0 & rs2 = x0	NONE	copies the value in register rs2 into register rd.	Integer Computational Instructions
C.AND	c.and rd', rs2'	x[8 + rd'] = x[8 + rd'] & x[8 + rs2']	NONE	NONE	computes the bitwise AND of of the value in register rd', and register rs2', then writes the result to rd'.	Integer Computational Instructions
C.OR	c.or rd', rs2'	x[8 + rd'] = x[8 + rd'] | x[8 + rs2']	NONE	NONE	computes the bitwise OR of of the value in register rd', and register rs2', then writes the result to rd'.	Integer Computational Instructions
C.XOR	c.and rd', rs2'	x[8 + rd'] = x[8 + rd'] ^ x[8 + rs2']	NONE	NONE	computes the bitwise XOR of of the value in register rd', and register rs2', then writes the result to rd'.	Integer Computational Instructions
C.SUB	c.sub rd', rs2'	x[8 + rd'] = x[8 + rd'] - x[8 + rs2']	NONE	NONE	subtracts the value in registers rs2' from value in rd' and writes the result to register rd'.	Integer Computational Instructions
C.EBREAK	c.ebreak	RaiseException(Breakpoint)	NONE	Raise a Breakpoint exception.	cause control to be transferred back to the debugging environment.	Integer Computational Instructions
C.J	c.j imm[11:1]	pc += sext(imm[11:1])	NONE	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	performs an unconditional control transfer. The offset is sign-extended and added to the pc to form the jump target address.	Control Transfer Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.JAL	c.jal imm[11:1]	x[1] = pc+2; pc += sext(imm[11:1])	NONE	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	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.	Control Transfer Instructions
C.JR	c.jr rs1	pc = x[rs1]	rs1 = x0	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	performs an unconditional control transfer to the address in register rs1.	Control Transfer Instructions
C.JALR	c.jalr rs1	t = pc+2; pc = x[rs1]; x[1] = t	rs1 = x0	jumps to an unaligned address (4-byte or 2-byte boundary) will usually raise an exception.	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.	Control Transfer Instructions
C.BEQZ	c.beqz rs1', imm[8:1]	if (x[8+rs1'] == 0) pc += sext(imm[8:1])	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	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.	Control Transfer Instructions
C.BNEZ	c.bnez rs1', imm[8:1]	if (x[8+rs1'] != 0) pc += sext(imm[8:1])	NONE	no instruction fetch misaligned exception is generated for a conditional branch that is not taken. An Instruction address misaligned exception is raised if the target address is not aligned on 4-byte or 2-byte boundary, because the core supports compressed instructions.	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.	Control Transfer Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.LWSP	c.lwsp rd, uimm(x2)	x[rd] = M[x[2] + zext(uimm[7:2])][31:0]	rd = x0	loads with a destination of x0 must still raise any exceptions, also an exception if the memory address isn’t aligned (4-byte boundary).	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.	Load and Store Instructions
C.SWSP	c.swsp rd, uimm(x2)	M[x[2] + zext(uimm[7:2])][31:0] = x[rs2]	NONE	an exception raised if the memory address isn’t aligned (4-byte boundary).	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.	Load and Store Instructions
C.LW	c.lw rd', uimm(rs1')	x[8+rd'] = M[x[8+rs1'] + zext(uimm[6:2])][31:0])	NONE	an exception raised if the memory address isn’t aligned (4-byte boundary).	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'.	Load and Store Instructions
C.SW	c.sw rs2', uimm(rs1')	M[x[8+rs1'] + zext(uimm[6:2])][31:0] = x[8+rs2']	NONE	an exception raised if the memory address isn’t aligned (4-byte boundary).	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'.	Load and Store Instructions

3.2.5. RV32Zicsr Control and Status Register Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
CSRRW	csrrw rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = x[rs1]; x[rd] = t	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations
CSRRS	csrrs rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = t | x[rs1]; x[rd] = t	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
CSRRC	csrrc rd, csr, rs1	t = CSRs[csr]; CSRs[csr] = t & ~x[rs1]; x[rd] = t	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations
CSRRWI	csrrwi rd, csr, uimm[4:0]	x[rd] = CSRs[csr]; CSRs[csr] = zext(uimm[4:0])	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
CSRRSI	csrrsi rd, csr, uimm[4:0]	t = CSRs[csr]; CSRs[csr] = t | zext(uimm[4:0]); x[rd] = t	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
CSRRCI	csrrci rd, csr, uimm[4:0]	t = CSRs[csr]; CSRs[csr] = t & ~zext(uimm[4:0]); x[rd] = t	NONE	Attempts to access a non-existent CSR raise an illegal instruction exception. Attempts to access a CSR without appropriate privilege level or to write a read-only register also raise illegal instruction exceptions.	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.	Control and Status Register Operations

3.2.6. RVZifencei Instruction Fetch Fence

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
FENCE.I	fence.i	Fence(Store, Fetch)	NONE	NONE	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.	Fetch Fence Operations

3.2.7. RV32Zcb Code Size Reduction Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.ZEXT.B	c.zext.b rd'	x[8 + rd'] = zext(x[8 + rd'][7:0])	NONE	NONE	This instruction takes a single source/destination operand. It zero-extends the least-significant byte of the operand by inserting zeros into all of the bits more significant than 7.	Code Size Reduction Operations
C.SEXT.B	c.sext.b rd'	x[8 + rd'] = sext(x[8 + rd'][7:0])	NONE	NONE	This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits. It also requires Bit-Manipulation (Zbb) extension support.	Code Size Reduction Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.ZEXT.H	c.zext.h rd'	x[8 + rd'] = zext(x[8 + rd'][15:0])	NONE	NONE	This instruction takes a single source/destination operand. It zero-extends the least-significant halfword of the operand by inserting zeros into all of the bits more significant than 15. It also requires Bit-Manipulation (Zbb) extension support.	Code Size Reduction Operations
C.SEXT.H	c.sext.h rd'	x[8 + rd'] = sext(x[8 + rd'][15:0])	NONE	NONE	This instruction takes a single source/destination operand. It sign-extends the least-significant halfword in the operand by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits. It also requires Bit-Manipulation (Zbb) extension support.	Code Size Reduction Operations
C.NOT	c.not rd'	x[8 + rd'] = x[8 + rd'] ^ -1	NONE	NONE	This instruction takes the one’s complement of rd'/rs1' and writes the result to the same register.	Code Size Reduction Operations
C.MUL	c.mul rd', rs2'	x[8 + rd'] = (x[8 + rd'] * x[8 + rs2']) [31:0]	NONE	NONE	performs a 32-bit × 32-bit multiplication and places the lower 32 bits in the destination register (Both rd' and rs2' treated as signed numbers). It also requires M extension support.	Code Size Reduction Operations
C.LHU	c.lhu rd', uimm(rs1')	x[8+rd'] = zext(M[x[8+rs1'] + zext(uimm[1])] [15:0])	NONE	an exception raised if the memory address isn’t aligned (2-byte boundary).	This instruction loads a halfword from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting halfword is zero extended and is written to rd'.	Code Size Reduction Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
C.LH	c.lh rd', uimm(rs1')	x[8+rd'] = sext(M[x[8+rs1'] + zext(uimm[1])][15:0])	NONE	an exception raised if the memory address isn't aligned (2-byte boundary).	This instruction loads a halfword from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting halfword is sign extended and is written to rd'.	Code Size Reduction Operations
C.LBU	c.lbu rd', uimm(rs1')	x[8+rd'] = zext(M[x[8+rs1'] + zext(uimm[1:0])][7:0])	NONE	NONE	This instruction loads a byte from the memory address formed by adding rs1' to the zero extended immediate uimm. The resulting byte is zero extended and is written to rd'.	Code Size Reduction Operations
C.SH	c.sh rs2', uimm(rs1')	M[x[8+rs1'] + zext(uimm[1])][15:0] = x[8+rs2']	NONE	an exception raised if the memory address isn't aligned (2-byte boundary).	This instruction stores the least significant halfword of rs2' to the memory address formed by adding rs1' to the zero extended immediate uimm.	Code Size Reduction Operations
C.SB	c.sb rs2', uimm(rs1')	M[x[8+rs1'] + zext(uimm[1:0])][7:0] = x[8+rs2']	NONE	NONE	This instruction stores the least significant byte of rs2' to the memory address formed by adding rs1' to the zero extended immediate uimm.	Code Size Reduction Operations

3.2.8. RVZba Address generation instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
ADD.UW	add.uw rd, rs1, rs2	X(rd) = rs2 + EXTZ(X(rs1)[31..0])	NONE	NONE	This instruction performs an XLEN-wide addition between rs2 and the zero-extended least-significant word of rs1.	Address generation instructions
SH1ADD	sh1add rd, rs1, rs2	X(rd) = X(rs2) + (X(rs1) << 1)	NONE	NONE	This instruction shifts rs1 to the left by 1 bit and adds it to rs2.	Address generation instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
SH1ADD.UW	sh1add.uw rd, rs1, rs2	X(rd) = rs2 + (EXTZ(X(rs1)[31..0]) << 1)	NONE	NONE	This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 place.	Address generation instructions
SH2ADD	sh2add rd, rs1, rs2	X(rd) = X(rs2) + (X(rs1) << 2)	NONE	NONE	This instruction shifts rs1 to the left by 2 bit and adds it to rs2.	Address generation instructions
SH2ADD.UW	sh2add.uw rd, rs1, rs2	X(rd) = rs2 + (EXTZ(X(rs1)[31..0]) << 2)	NONE	NONE	This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 2 places.	Address generation instructions
SH3ADD	sh3add rd, rs1, rs2	X(rd) = X(rs2) + (X(rs1) << 3)	NONE	NONE	This instruction shifts rs1 to the left by 3 bit and adds it to rs2.	Address generation instructions
SH3ADD.UW	sh3add.uw rd, rs1, rs2	X(rd) = rs2 + (EXTZ(X(rs1)[31..0]) << 3)	NONE	NONE	This instruction performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 places.	Address generation instructions
SLLI.UW	slli.uw rd, rs1, imm	X(rd) = (EXTZ(X(rs)[31..0]) << imm)	NONE	NONE	This instruction takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.	Address generation instructions

3.2.9. RVZbb Basic bit-manipulation

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
ANDN	andn rd, rs1, rs2	X(rd) = X(rs1) & ~X(rs2)	NONE	NONE	Performs bitwise AND operation between rs1 and bitwise inversion of rs2.	Logical_with_negate
ORN	orn rd, rs1, rs2	X(rd) = X(rs1) | ~X(rs2)	NONE	NONE	Performs bitwise OR operation between rs1 and bitwise inversion of rs2.	Logical_with_negate
XNOR	xnor rd, rs1, rs2	X(rd) = ~(X(rs1) ^ X(rs2))	NONE	NONE	Performs bitwise XOR operation between rs1 and rs2, then complements the result.	Logical_with_negate
CLZ	clz rd, rs	if [x[i]] == 1 then return(i) else return -1	NONE	NONE	Counts leading zero bits in rs.	Count_leading_trailing_zero_bits
CTZ	ctz rd, rs	if [x[i]] == 1 then return(i) else return xlen;	NONE	NONE	Counts trailing zero bits in rs.	Count_leading_trailing_zero_bits
CLZW	clzw rd, rs	if [x[i]] == 1 then return(i) else return -1	NONE	NONE	Counts leading zero bits in the least-significant word of rs.	Count_leading_trailing_zero_bits
CTZW	ctzw rd, rs	if [x[i]] == 1 then return(i) else return 32;	NONE	NONE	Counts trailing zero bits in the least-significant word of rs.	Count_leading_trailing_zero_bits
CPOP	cpop rd, rs	if rs[i] == 1 then bitcount = bitcount + 1 else ()	NONE	NONE	Counts set bits in rs.	Count_population
CPOPW	cpopw rd, rs	if rs[i] == 0b1 then bitcount = bitcount + 1 else ()	NONE	NONE	Counts set bits in the least-significant word of rs.	Count_population
MAX	max rd, rs1, rs2	if rs1_val <_s rs2_val then rs2_val else rs1_val	NONE	NONE	Returns the larger of two signed integers.	Integer_minimum_maximum
MAXU	maxu rd, rs1, rs2	if rs1_val <_u rs2_val then rs2_val else rs1_val	NONE	NONE	Returns the larger of two unsigned integers.	Integer_minimum_maximum
MIN	min rd, rs1, rs2	if rs1_val <_s rs2_val then rs1_val else rs2_val	NONE	NONE	Returns the smaller of two signed integers.	Integer_minimum_maximum

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
MINU	minu rd, rs1, rs2	if rs1_val <_u rs2_val then rs1_val else rs2_val	NONE	NONE	Returns the smaller of two unsigned integers.	Integer_minimum_maximum
SEXT.B	sext.b rd, rs	X(rd) = EXTS(X(rs)[7..0])	NONE	NONE	Sign-extends the least-significant byte in the source to XLEN.	Sign_and_zero_extension
SEXT.H	sext.h rd, rs	X(rd) = EXTS(X(rs)[15..0])	NONE	NONE	Sign-extends the least-significant halfword in rs to XLEN.	Sign_and_zero_extension
ZEXT.H	zext.h rd, rs	X(rd) = EXTZ(X(rs)[15..0])	NONE	NONE	Zero-extends the least-significant halfword of the source to XLEN.	Sign_and_zero_extension
ROL	rol rd, rs1, rs2	(X(rs1) << log2(XLEN)) | (X(rs1) >> (xlen - log2(XLEN)))	NONE	NONE	Performs a rotate left of rs1 by the amount in least-significant log2(XLEN) bits of rs2.	Bitwise_rotation
ROR	ror rd, rs1, rs2	(X(rs1) >> log2(XLEN)) | (X(rs1) << (xlen - log2(XLEN)))	NONE	NONE	Performs a rotate right of rs1 by the amount in least-significant log2(XLEN) bits of rs2.	Bitwise_rotation
RORI	rori rd, rs1, shamt	(X(rs1) >> log2(XLEN)) | (X(rs1) << (xlen - log2(XLEN)))	NONE	NONE	Performs a rotate right of rs1 by the amount in least-significant log2(XLEN) bits of shamt.	Bitwise_rotation
ROLW	rolw rd, rs1, rs2	EXTSrs1 << X(rs2)[4..0]) | (rs1)	NONE	NONE	Performs a rotate left on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2.	Bitwise_rotation
RORIW	roriw rd, rs1, shamt	(rs1_data >> shamt[4..0]) | (rs1_data << (32 - shamt[4..0]))	NONE	NONE	Performs a rotate right on the least-significant word of rs1 by the amount in least-significant log2(XLEN) bits of shamt.	Bitwise_rotation

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
RORW	rorw rd, rs1, rs2	(rs1 >> X(rs2)[4..0]) | (rs1 << (32 - X(rs2)[4..0]))	NONE	NONE	Performs a rotate right on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2.	Bitwise_rotation
ORC.b	orc.b rd, rs	if { input[(i + 7)..i] == 0 then 0b00000000 else 0b11111111	NONE	NONE	Sets the bits of each byte in rd to all zeros if no bit within the respective byte of rs is set, or to all ones if any bit within the respective byte of rs is set.	OR_Combine
REV8	rev8 rd, rs	output[i..(i + 7)] = input[(j - 7)..j]	NONE	NONE	Reverses the order of the bytes in rs.	Byte_reverse

3.2.10. RVZbc Carry-less multiplication

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
CLMUL	clmul rd, rs1, rs2	foreach (i from 1 to xlen by 1) { output = if ((rs2 >> i) & 1) then output ^ (rs1 << i); else output;}	NONE	NONE	clmul produces the lower half of the 2.XLEN carry-less product.	Carry-less multiplication Operations
CLMULH	clmulh rd, rs1, rs2	foreach (i from 1 to xlen by 1) { output = if rs2_val else output}	NONE	NONE	clmulh produces the upper half of the 2.XLEN carry-less product.	Carry-less multiplication Operations
CLMULR	clmulr rd, rs1, rs2	foreach (i from 0 to (xlen - 1) by 1) { output = if rs2_val else output}	NONE	NONE	clmulr produces bits 2.XLEN-2:XLEN-1 of the 2.XLEN carry-less product.	Carry-less multiplication Operations

3.2.11. RVZbs Single bit Instructions

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BCLR	bclr rd, rs1, rs2	X(rd) = X(rs1) & ~(1 << (X(rs2) & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit cleared at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.	Single_bit_Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BCLRI	bclri rd, rs1, shamt	X(rd) = X(rs1) & ~(1 << (shamt & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit cleared at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.	Single_bit_Operations
BEXT	bext rd, rs1, rs2	X(rd) = (X(rs1) >> (X(rs2) & (XLEN - 1))) & 1	NONE	NONE	This instruction returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.	Single_bit_Operations
BEXTI	bexti rd, rs1, shamt	X(rd) = (X(rs1) >> (shamt & (XLEN - 1))) & 1	NONE	NONE	This instruction returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.	Single_bit_Operations
BINV	binv rd, rs1, rs2	X(rd) = X(rs1) ^ (1 << (X(rs2) & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit inverted at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.	Single_bit_Operations
BINVI	binvi rd, rs1, shamt	X(rd) = X(rs1) ^ (1 << (shamt & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit inverted at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.	Single_bit_Operations

Name	Format	Pseudocode	Invalid_values	Exception_raised	Description	Op Name
BSET	bset rd, rs1, rs2	X(rd) = X(rs1) | (1 << (X(rs2) & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit set at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.	Single_bit_Operations
BSETI	bseti rd, rs1, shamt	X(rd) = X(rs1) | (1 << (shamt & (XLEN - 1)))	NONE	NONE	This instruction returns rs1 with a single bit set at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.	Single_bit_Operations

3.3. Traps, Interrupts, Exceptions

Traps are composed of interrupts and exceptions. Interrupts are asynchronous events whereas exceptions are synchronous ones. On one hand, interrupts are occuring independently of the instructions (mainly raised by peripherals or debug module). On the other hand, an instruction may raise exceptions synchronously.

3.3.1. Raising Traps

When a trap is raised, the behaviour of the CVA6 core depends on several CSRs and some CSRs are modified.

3.3.1.1. Configuration CSRs

CSRs having an effect on the core behaviour when a trap occurs are:

- `mstatus` and `sstatus` : several fields control the core behaviour like interrupt enable (`MIE` , `SIE`)
- `mtvec` and `stvec` : specifies the address of trap handler.
- `medeleg` : specifies which exceptions can be handled by a lower privileged mode (S-mode)
- `mideleg` : specifies which interrupts can be handled by a lower privileged mode (S-mode)

3.3.1.2. Modified CSRs

CSRs (or fields) updated by the core when a trap occurs are:

- `mstatus` or `sstatus` : several fields are updated like previous privilege mode (`MPP` , `SPP`), previous interrupt enabled (`MPIE` , SPIE``)
- `mepc` or `sepc` : updated with the virtual address of the interrupted instruction or the instruction raising the exception.
- `mcause` or `scause` : updated with a code indicating the event causing the trap.
- `mtval` or `stval` : updated with exception specific information like the faulting virtual address

3.3.1.3. Supported exceptions

The following exceptions are supported by the CVA6:

- instruction address misaligned
  - control flow instruction with misaligned target
- instruction access fault
  - access to PMP region without execute permissions
- illegal instruction:
  - unimplemented CSRs
  - unsupported extensions

- breakpoint (`EBREAK`)
- load address misaligned:
  - `LH` at 2n+1 address
  - `LW` at 4n+1, 4n+2, 4n+3 address
- load access fault
  - access to PMP region without read permissions
- store/AMO address misaligned
  - `SH` at 2n+1 address
  - `SW` at 4n+1, 4n+2, 4n+3 address
- store/AMO access fault
  - access to PMP region without write permissions
- environment call (`ECALL`) from U-mode
- environment call (`ECALL`) from S-mode
- environment call (`ECALL`) from M-mode
- instruction page fault
- load page fault
  - access to effective address without read permissions
- store/AMO page fault
  - access to effective address without write permissions
- debug request (custom) via debug interface

Note: all exceptions are supported except the ones linked to the hypervisor extension

3.3.2. Trap return

Trap handler ends with trap return instruction (`MRET` , `SRET`). The behaviour of the CVA6 core depends on several CSRs.

3.3.2.1. Configuration CSRs

CSRs having an effect on the core behaviour when returning from a trap are:

- `mstatus` : several fields control the core behaviour like previous privilege mode (`MPP` , `SPP`), previous interrupt enabled (`MPIE` , `SPIE`)

3.3.2.2. Modified CSRs

CSRs (or fields) updated by the core when returning from a trap are:

- `mstatus` : several fields are updated like interrupt enable (`MIE` , `SIE`), modify privilege (`MPRV`)

3.3.3. Interrupts

- external interrupt: `irq_i` signal
- software interrupt (inter-processor interrupt): `ipi_i` signal
- timer interrupt: `time_irq_i` signal
- debug interrupt: `debug_req_i` signal

These signals are level sensitive. It means the interrupt is raised until it is cleared.

The exception code field (`mcause` CSR) depends on the interrupt source.

3.3.4. Wait for Interrupt

- CVA6 implementation: `WFI` stalls the core. The instruction is not available in U-mode (raise illegal instruction exception). Such exception is also raised when `TW=1` in `mstatus` .

3.4. csr

3.4.1. Conventions

In the subsequent sections, register fields are labeled with one of the following abbreviations:

- WPRI (Writes Preserve Values, Reads Ignore Values): read/write field reserved for future use. For forward compatibility, implementations that do not furnish these fields must make them read-only zero.
- WLRL (Write/Read Only Legal Values): read/write CSR field that specifies behavior for only a subset of possible bit encodings, with other bit encodings reserved.
- WARL (Write Any Values, Reads Legal Values): read/write CSR fields which are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read.
- ROCST (Read-Only Constant): A special case of WARL field which admits only one legal value, and therefore, behaves as a constant field that silently ignores writes.
- ROVAR (Read-Only Variable): A special case of WARL field which can take multiple legal values but cannot be modified by software and depends only on the architectural state of the hart.

In particular, a register that is not internally divided into multiple fields can be considered as containing a single field of XLEN bits. This allows to clearly represent read-write registers holding a single legal value (typically zero).

3.4.2. Register Summary

Address	Register Name	Privilege	Description
0x300	`MSTATUS`	MRW	The mstatus register keeps track of and controls the hart's current operating state.
0x301	`MISA`	MRW	misa is a read-write register reporting the ISA supported by the hart.
0x304	`MIE`	MRW	The mie register is an MXLEN-bit read/write register containing interrupt enable bits.
0x305	`MTVEC`	MRW	MXLEN-bit read/write register that holds trap vector configuration.
0x310	`MSTATUSH`	MRW	The mstatush register keeps track of and controls the hart's current operating state.
0x323-0x33f	`MHPMEVENT[3-31]`	MRW	The mhpmevent is a MXLEN-bit event register which controls mhpmcounter3.
0x340	`MSCRATCH`	MRW	The mscratch register is an MXLEN-bit read/write register dedicated for use by machine mode.
0x341	`MEPC`	MRW	The mepc is a warl register that must be able to hold all valid physical and virtual addresses.
0x342	`MCAUSE`	MRW	The mcause register stores the information regarding the trap.
0x343	`MTVAL`	MRW	The mtval is a warl register that holds the address of the instruction which caused the exception.
0x344	`MIP`	MRW	The mip register is an MXLEN-bit read/write register containing information on pending interrupts.
0x3a0-0x3a1	`PMPCFG[0-1]`	MRW	PMP configuration register
0x3a2-0x3af	`PMPCFG[2-15]`	MRW	PMP configuration register

Address	Register Name	Privilege	Description
0x3b0-0x3b7	PMPADDR[0-7]	MRW	Physical memory protection address register
0x3b8-0x3ef	PMPADDR[8-63]	MRW	Physical memory protection address register
0x7c0	ICACHE	MRW	the register controls the operation of the i-cache unit.
0x7c1	DCACHE	MRW	the register controls the operation of the d-cache unit.
0xb00	MCYCLE	MRW	Counts the number of clock cycles executed from an arbitrary point in time.
0xb02	MINSTRET	MRW	Counts the number of instructions completed from an arbitrary point in time.
0xb03-0xb1f	MHPMCOUNTER[3-31]	MRW	The mhpmcounter is a 64-bit counter. Returns lower 32 bits in RV32I mode.
0xb80	MCYCLEH	MRW	upper 32 bits of mcycle
0xb82	MINSTRETH	MRW	Upper 32 bits of minstret.
0xb83-0xb9f	MHPMCOUNTER[3-31]H	MRW	The mhpmcounterh returns the upper half word in RV32I systems.
0xf11	MVENDORID	MRO	32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core.
0xf12	MARCHID	MRO	MXLEN-bit read-only register encoding the base microarchitecture of the hart.
0xf13	MIMPID	MRO	Provides a unique encoding of the version of the processor implementation.
0xf14	MHARTID	MRO	MXLEN-bit read-only register containing the integer ID of the hardware thread running the code.
0xf15	MCONFIGPTR	MRO	MXLEN-bit read-only register that holds the physical address of a configuration data structure.

3.4.3. Register Description

3.4.3.1. MSTATUS

Address

0x300

Reset Value

0x00001800

Privilege

MRW

Description

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

Bits	Field Name	Reset Value	Type	Legal Values	Description
0	UIE	0x0	ROCST	0x0	Stores the state of the user mode interrupts.
1	SIE	0x0	ROCST	0x0	Stores the state of the supervisor mode interrupts.
2	RESERVED_2	0x0	WPRI		*Reserved*
3	MIE	0x0	WLRL	0x0 - 0x1	Stores the state of the machine mode interrupts.
4	UPIE	0x0	ROCST	0x0	Stores the state of the user mode interrupts prior to the trap.
5	SPIE	0x0	ROCST	0x0	Stores the state of the supervisor mode interrupts prior to the trap.
6	UBE	0x0	ROCST	0x0	control the endianness of memory accesses other than instruction fetches for user mode
7	MPIE	0x0	WLRL	0x0 - 0x1	Stores the state of the machine mode interrupts prior to the trap.
8	SPP	0x0	ROCST	0x0	Stores the previous priority mode for supervisor.
[10:9]	RESERVED_9	0x0	WPRI		*Reserved*
[12:11]	MPP	0x3	WARL	0x3	Stores the previous priority mode for machine.
[14:13]	FS	0x0	ROCST	0x0	Encodes the status of the floating-point unit, including the CSR fcsr and floating-point data registers.
[16:15]	XS	0x0	ROCST	0x0	Encodes the status of additional user-mode extensions and associated state.
17	MPRV	0x0	ROCST	0x0	Modifies the privilege level at which loads and stores execute in all privilege modes.

Bits	Field Name	Reset Value	Type	Legal Values	Description
18	SUM	0x0	ROCST	0x0	Modifies the privilege with which S-mode loads and stores access virtual memory.
19	MXR	0x0	ROCST	0x0	Modifies the privilege with which loads access virtual memory.
20	TVM	0x0	ROCST	0x0	Supports intercepting supervisor virtual-memory management operations.
21	TW	0x0	ROCST	0x0	Supports intercepting the WFI instruction.
22	TSR	0x0	ROCST	0x0	Supports intercepting the supervisor exception return instruction.
23	SPELP	0x0	ROCST	0x0	Supervisor mode previous expected-landing-pad (ELP) state.
[30:24]	RESERVED_24	0x0	WPRI		*Reserved*
31	SD	0x0	ROCST	0x0	Read-only bit that summarizes whether either the FS field or XS field signals the presence of some dirty state.

3.4.3.2. MISA

Address

0x301

Reset Value

0x40001106

Privilege

MRW

Description

misa is a read-write register reporting the ISA supported by the hart.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[25:0]	EXTENSIONS	0x1106	ROCST	0x1106	Encodes the presence of the standard extensions, with a single bit per letter of the alphabet.
[29:26]	RESERVED_26	0x0	WPRI		*Reserved*
[31:30]	MXL	0x1	WARL	0x1	Encodes the native base integer ISA width.

3.4.3.3. MIE

Address

0x304

Reset Value

0x00000000

Privilege

MRW

Description

The mie register is an MXLEN-bit read/write register containing interrupt enable bits.

Bits	Field Name	Reset Value	Type	Legal Values	Description
0	USIE	0x0	ROCST	0x0	User Software Interrupt enable.
1	SSIE	0x0	ROCST	0x0	Supervisor Software Interrupt enable.
2	VSSIE	0x0	ROCST	0x0	VS-level Software Interrupt enable.
3	MSIE	0x0	ROCST	0x0	Machine Software Interrupt enable.
4	UTIE	0x0	ROCST	0x0	User Timer Interrupt enable.
5	STIE	0x0	ROCST	0x0	Supervisor Timer Interrupt enable.
6	VSTIE	0x0	ROCST	0x0	VS-level Timer Interrupt enable.
7	MTIE	0x0	WLRL	0x0 - 0x1	Machine Timer Interrupt enable.
8	UEIE	0x0	ROCST	0x0	User External Interrupt enable.
9	SEIE	0x0	ROCST	0x0	Supervisor External Interrupt enable.
10	VSEIE	0x0	ROCST	0x0	VS-level External Interrupt enable.
11	MEIE	0x0	WLRL	0x0 - 0x1	Machine External Interrupt enable.
12	SGEIE	0x0	ROCST	0x0	HS-level External Interrupt enable.
[31:13]	RESERVED_13	0x0	WPRI		*Reserved*

3.4.3.4. MTVEC

Address

0x305

Reset Value

0x80010000

Privilege

MRW

Description

MXLEN-bit read/write register that holds trap vector configuration.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[1:0]	MODE	0x0	WARL	0x0	Vector mode.
[31:2]	BASE	0x20004000	WARL	0x00000000 - 0x3FFFFFFF	Vector base address.

3.4.3.5. MSTATUSH

Address

0x310

Reset Value

0x00000000

Privilege

MRW

Description

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

Bits	Field Name	Reset Value	Type	Legal Values	Description
[3:0]	RESERVED_0	0x0	WPRI		*Reserved*
4	SBE	0x0	ROCST	0x0	control the endianness of memory accesses other than instruction fetches for supervisor mode
5	MBE	0x0	ROCST	0x0	control the endianness of memory accesses other than instruction fetches for machine mode
6	GVA	0x0	ROCST	0x0	Stores the state of the supervisor mode interrupts.
7	MPV	0x0	ROCST	0x0	Stores the state of the user mode interrupts.
8	RESERVED_8	0x0	WPRI		*Reserved*
9	MPELP	0x0	ROCST	0x0	Machine mode previous expected-landing-pad (ELP) state.
[31:10]	RESERVED_10	0x0	WPRI		*Reserved*

3.4.3.6. MHPMEVENT[3-31]

Address

0x323-0x33f

Reset Value

0x00000000

Privilege

MRW

Description

The mhpmevent is a MXLEN-bit event register which controls mhpmcounter3.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MHPMEVENT[I]	0x00000000	ROCST	0x0	The mhpmevent is a MXLEN-bit event register which controls mhpmcounter3.

3.4.3.7. MSCRATCH

Address

0x340

Reset Value

0x00000000

Privilege

MRW

Description

The mscratch register is an MXLEN-bit read/write register dedicated for use by machine mode.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MSCRATCH	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	The mscratch register is an MXLEN-bit read/write register dedicated for use by machine mode.

3.4.3.8. MEPC

Address

0x341

Reset Value

0x00000000

Privilege

MRW

Description

The mepc is a warl register that must be able to hold all valid physical and virtual addresses.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MEPC	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	The mepc is a warl register that must be able to hold all valid physical and virtual addresses.

3.4.3.9. MCAUSE

Address

0x342

Reset Value

0x00000000

Privilege

MRW

Description

The mcause register stores the information regarding the trap.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[30:0]	EXCEPTION_CODE	0x0	WLRL	0x0 - 0x8, 0xb	Encodes the exception code.
31	INTERRUPT	0x0	WLRL	0x0 - 0x1	Indicates whether the trap was due to an interrupt.

3.4.3.10. MTVAL

Address

0x343

Reset Value

0x00000000

Privilege

MRW

Description

The mtval is a warl register that holds the address of the instruction which caused the exception.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MTVAL	0x00000000	ROCST	0x0	The mtval is a warl register that holds the address of the instruction which caused the exception.

3.4.3.11. MIP

Address

0x344

Reset Value

0x00000000

Privilege

MRW

Description

The mip register is an MXLEN-bit read/write register containing information on pending interrupts.

Bits	Field Name	Reset Value	Type	Legal Values	Description
0	USIP	0x0	ROCST	0x0	User Software Interrupt Pending.
1	SSIP	0x0	ROCST	0x0	Supervisor Software Interrupt Pending.
2	VSSIP	0x0	ROCST	0x0	VS-level Software Interrupt Pending.
3	MSIP	0x0	ROCST	0x0	Machine Software Interrupt Pending.

Bits	Field Name	Reset Value	Type	Legal Values	Description
4	UTIP	0x0	ROCST	0x0	User Timer Interrupt Pending.
5	STIP	0x0	ROCST	0x0	Supervisor Timer Interrupt Pending.
6	VSTIP	0x0	ROCST	0x0	VS-level Timer Interrupt Pending.
7	MTIP	0x0	ROVAR	0x0 - 0x1	Machine Timer Interrupt Pending.
8	UEIP	0x0	ROCST	0x0	User External Interrupt Pending.
9	SEIP	0x0	ROCST	0x0	Supervisor External Interrupt Pending.
10	VSEIP	0x0	ROCST	0x0	VS-level External Interrupt Pending.
11	MEIP	0x0	ROVAR	0x0 - 0x1	Machine External Interrupt Pending.
12	SGEIP	0x0	ROCST	0x0	HS-level External Interrupt Pending.
[31:13]	RESERVED_13	0x0	WPRI		*Reserved*

3.4.3.12. PMPCFG[0-1]

Address

0x3a0-0x3a1

Reset Value

0x00000000

Privilege

MRW

Description

PMP configuration register

Bits	Field Name	Reset Value	Type	Legal Values	Description
[7:0]	PMP[I*4 +0]CFG	0x0	WARL	masked: & 0x8f | 0x0	pmp configuration bits
[15:8]	PMP[I*4 +1]CFG	0x0	WARL	masked: & 0x8f | 0x0	pmp configuration bits
[23:16]	PMP[I*4 +2]CFG	0x0	WARL	masked: & 0x8f | 0x0	pmp configuration bits
[31:24]	PMP[I*4 +3]CFG	0x0	WARL	masked: & 0x8f | 0x0	pmp configuration bits

3.4.3.13. PMPCFG[2-15]

Address

0x3a2-0x3af

Reset Value

0x00000000

Privilege

MRW

Description

PMP configuration register

Bits	Field Name	Reset Value	Type	Legal Values	Description
[7:0]	PMP[I*4 +0]CFG	0x0	ROCST	0x0	pmp configuration bits
[15:8]	PMP[I*4 +1]CFG	0x0	ROCST	0x0	pmp configuration bits
[23:16]	PMP[I*4 +2]CFG	0x0	ROCST	0x0	pmp configuration bits
[31:24]	PMP[I*4 +3]CFG	0x0	ROCST	0x0	pmp configuration bits

3.4.3.14. PMPADDR[0-7]

Address

0x3b0-0x3b7

Reset Value

0x00000000

Privilege

MRW

Description

Physical memory protection address register

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	PMPADDR[I]	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	Physical memory protection address register

3.4.3.15. PMPADDR[8-63]

Address

0x3b8-0x3ef

Reset Value

0x00000000

Privilege

MRW

Description

Physical memory protection address register

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	PMPADDR[I]	0x00000000	ROCST	0x0	Physical memory protection address register

3.4.3.16. ICACHE

Address

0x7c0

Reset Value

0x00000001

Privilege

MRW

Description

the register controls the operation of the i-cache unit.

Bits	Field Name	Reset Value	Type	Legal Values	Description
0	ICACHE	0x1	RW	0x1	bit for cache-enable of instruction cache
[31:1]	RESERVED_1	0x0	WPRI		*Reserved*

3.4.3.17. DCACHE

Address

0x7c1

Reset Value

0x00000001

Privilege

MRW

Description

the register controls the operation of the d-cache unit.

Bits	Field Name	Reset Value	Type	Legal Values	Description
0	DCACHE	0x1	RW	0x1	bit for cache-enable of data cache
[31:1]	RESERVED_1	0x0	WPRI		*Reserved*

3.4.3.18. MCYCLE

Address

0xb00

Reset Value

0x00000000

Privilege

MRW

Description

Counts the number of clock cycles executed from an arbitrary point in time.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MCYCLE	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	Counts the number of clock cycles executed from an arbitrary point in time.

3.4.3.19. MINSTRET

Address

0xb02

Reset Value

0x00000000

Privilege

MRW

Description

Counts the number of instructions completed from an arbitrary point in time.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MINSTRET	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	Counts the number of instructions completed from an arbitrary point in time.

3.4.3.20. MHPMCOUNTER[3-31]

Address

0xb03-0xb1f

Reset Value

0x00000000

Privilege

MRW

Description

The mhpmcounter is a 64-bit counter. Returns lower 32 bits in RV32I mode.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MHPMCOUNTER[I]	0x00000000	ROCST	0x0	The mhpmcounter is a 64-bit counter. Returns lower 32 bits in RV32I mode.

3.4.3.21. MCYCLEH

Address

0xb80

Reset Value

0x00000000

Privilege

MRW

Description

upper 32 bits of mcycle

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MCYCLEH	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	upper 32 bits of mcycle

3.4.3.22. MINSTRETH

Address

0xb82

Reset Value

0x00000000

Privilege

MRW

Description

Upper 32 bits of minstret.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MINSTRETH	0x00000000	WARL	0x00000000 - 0xFFFFFFFF	Upper 32 bits of minstret.

3.4.3.23. MHPMCOUNTER[3-31]H

Address

0xb83-0xb9f

Reset Value

0x00000000

Privilege

MRW

Description

The mhpmcounterh returns the upper half word in RV32I systems.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MHPMCOUNTER[I]H	0x00000000	ROCST	0x0	The mhpmcounterh returns the upper half word in RV32I systems.

3.4.3.24. MVENDORID

Address

0xf11

Reset Value

0x00000602

Privilege

MRO

Description

32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MVENDORID	0x00000602	ROCST	0x602	32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core.

3.4.3.25. MARCHID

Address

0xf12

Reset Value

0x00000003

Privilege

MRO

Description

MXLEN-bit read-only register encoding the base microarchitecture of the hart.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MARCHID	0x00000003	ROCST	0x3	MXLEN-bit read-only register encoding the base microarchitecture of the hart.

3.4.3.26. MIMPID

Address

0xf13

Reset Value

0x00000000

Privilege

MRO

Description

Provides a unique encoding of the version of the processor implementation.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MIMPID	0x00000000	ROCST	0x0	Provides a unique encoding of the version of the processor implementation.

3.4.3.27. MHARTID

Address

0xf14

Reset Value

0x00000000

Privilege

MRO

Description

MXLEN-bit read-only register containing the integer ID of the hardware thread running the code.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MHARTID	0x00000000	ROCST	0x0	MXLEN-bit read-only register containing the integer ID of the hardware thread running the code.

3.4.3.28. MCONFIGPTR

Address

0xf15

Reset Value

0x00000000

Privilege

MRO

Description

MXLEN-bit read-only register that holds the physical address of a configuration data structure.

Bits	Field Name	Reset Value	Type	Legal Values	Description
[31:0]	MCONFIGPTR	0x00000000	ROCST	0x0	MXLEN-bit read-only register that holds the physical address of a configuration data structure.

3.5. AXI

3.5.1. 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.

Applicability of this chapter to configurations:

Configuration	Implementation
CV32A60AX	AXI included
CV32A60X	AXI included

3.5.1.1. 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

3.5.1.2. 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.

3.5.2. 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.

3.5.2.1. 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.

3.5.2.2. 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_label.
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_label.
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 Normal Non-cacheable Non-bufferable. (AWCACHE = 0b0010)
AWPROT	M	Yes	Protection attributes of a write transaction: privilege, security level, and access type. The value of AWPROT is always 0b000.
AWQOS	M	No (optional)	Quality of Service identifier for a write transaction. AWQOS = 0b0000

Signal	Src	Support	Description
AWREGION	M	No (optional)	Region indicator for a write transaction. AWREGION = 0b0000
AWUSER	M	No (optional)	User-defined extension for the write address channel. AWUSER = 0b00
AWATOP	M	Yes (optional)	AWATOP indicates the Properties of the Atomic Operation used for a write transaction. See atomic_transactions_label.
AWVALID	M	Yes	Indicates that the write address channel signals are valid.
AWREADY	S	Yes	Indicates that a transfer on the write address channel can be accepted.

3.5.2.3. 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
WDATA	M	Yes	Write data.
WSTRB	M	Yes (optional)	Write strobes, indicate which byte lanes hold valid data See data_read_and_write_structure_label.
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.

3.5.2.4. 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_label.
BRESP	S	Yes	Write response, indicates the status of a write transaction. See read_and_write_response_structure_label.
BUSER	S	No (optional)	User-defined extension for the write response channel. Not supported.
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.

3.5.2.5. 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_label.
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 have a length equal to 0, ICACHE_LINE_WIDTH/64 or DCACHE_LINE_WIDTH/64.
ARSIZE	M	Yes (optional)	Size, the number of bytes in each data transfer in a read transaction See address_structure_label.
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 Normal Non-cacheable Non-bufferable. (ARCACHE = 0b0010)
ARPROT	M	Yes	Protection attributes of a read transaction: privilege, security level, and access type. The value of ARPROT is always 0b000.
ARQOS	M	No (optional)	Quality of Service identifier for a read transaction. ARQOS= 0b00
ARREGION	M	No (optional)	Region indicator for a read transaction. ARREGION= 0b00
ARUSER	M	No (optional)	User-defined extension for the read address channel. ARUSER= 0b00
ARVALID	M	Yes (optional)	Indicates that the read address channel signals are valid.
ARREADY	S	Yes (optional)	Indicates that a transfer on the read address channel can be accepted.

3.5.2.6. 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_label.
RDATA	S	Yes	Read data.

Signal	Src	Support	Description
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.
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.

3.5.3. 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

3.5.3.1. 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 equal to 0, ICACHE_LINE_WIDTH/64 or DCACHE_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

The maximum value can be taking by AxSIZE is log2(AXI DATA WIDTH/8) (8 bytes by transfer). If(RV32) AWSIZE < 3 (The maximum store size is 4 bytes)

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)

3.5.3.2. Data read and write structure: (Section A3.4.4)

Write strobes

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)]` .

Write Strobe width is equal to (AXI_DATA_WIDTH/8) (n = (AXI_DATA_WIDTH/8)-1).

The size of transactions performed by cva6 is equal to the number of data byte lanes containing valid information. This means 1, 2, 4, ... or (AXI_DATA_WIDTH/8) byte lanes containing valid information. CVA6 doesn't perform unaligned memory acces, therefore the WSTRB take only combination of aligned access If(RV32) WSTRB < 255 (Since AWSIZE lower than 3, so the data bus cannot have more than 4 valid byte lanes)

Unaligned transfers

For any burst that is made up of data transfers wider than 1 byte, the first bytes accessed might be unaligned with the natural address boundary. For example, a 32-bit data packet that starts at a byte address of 0x1002 is not aligned to the natural 32-bit transfer size.

CVA6 does not perform Unaligned transfers.

3.5.3.3. 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).

3.5.4. 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 takeq 0b0010. The subordinate should be a Normal Non-cacheable Non-bufferable.

The required behavior for Normal Non-cacheable 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 modifiable.
- Writes can be merged.

3.5.5. 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. (0 for instruction fetch and 1 for data)
- 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.
- For Exclusive transaction, id = 3.

3.5.6. AXI Ordering Model (Section A6)

3.5.6.1. 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.

3.5.6.2. 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.

3.5.6.3. 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 Normal, because `AxCACHE[1]` is asserted.

Normal transactions are used to access Memory locations and are not expected to be used to access Peripheral regions.

A Normal access to a Peripheral region must complete in a protocol-compliant manner, but the result is IMPLEMENTATION DEFINED.

A write transaction performed by CVA6 is Non-bufferable (It is not possible to send an early response before the transaction reach the final destination), because `AxCACHE[0]` is deasserted.

3.5.6.4. 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.

3.5.7. 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.

If(RVA) AWATOP = 0 (If AMO instructions are not supported, CVA6 cannot perform Atomic transaction)

CVA6 supports just the AtomicLoad and AtomicSwap transaction. So `AWATOP[5:4]` *can be 00, 10 or 11.*

CVA6 performs only little-endian operation. So `AWATOP[3]` *= 0.*

For AtomicLoad, CVA6 supports all arithmetic operations encoded on the lower-order `AWATOP[2:0]` *signals.*

3.5.8. CVA6 Constraints

This section describes cross-cases between several features that are not supported by CVA6.

- ARID = 0 && ARSIZE = log(AXI_DATA_WIDTH/8), CVA6 always requests max number of words in case of read transaction with ID 0 (instruction fetch)
- if(RV32) ARSIZE != 3 && ARLEN = 0 && ARID = 1, the maximum load instruction size is 4 bytes
- if(!RVA) AxLOCK = 0, if AMO instructions are not supported, CVA6 cannot perform exclusive transaction
- if(RVA) AxLOCK = 1 ⇒ AxSIZE > 1, CVA6 doesn't perform exclusive transaction with size lower than 4 bytes

3.6. CV-X-IF Interface and Coprocessor

The CV-X-IF interface of CVA6 allows to extend its supported instruction set with external coprocessors.

Applicability of this chapter to configurations:

Configuration	Implementation
CV32A60AX	CV-X-IF included
CV32A60X	CV-X-IF included
CV64A6_MMU	CV-X-IF included

3.6.1. CV-X-IF interface specification

3.6.1.1. Description

This design specification presents global functionalities of Core-V-eXtension-Interface (XIF, CVXIF, CV-X-IF, X-interface) in the CVA6 core.

SOURCECODE

```
The CORE-V X-Interface is a RISC-V eXtension interface that provides a
generalized framework suitable to implement custom coprocessors and ISA
extensions for existing RISC-V processors.

--core-v-xif Readme, https://github.com/openhwgroup/core-v-xif
```

The specification of the CV-X-IF bus protocol can be found at [CV-X-IF].

CV-X-IF aims to:

- Create interfaces to connect a coprocessor to the CVA6 to execute instructions.
- Offload CVA6 illegal instrutions to the coprocessor to be executed.
- Get the results of offloaded instructions from the coprocessor so they are written back into the CVA6 register file.
- Add standard RISC-V instructions unsupported by CVA6 or custom instructions and implement them in a coprocessor.
- Kill offloaded instructions to allow speculative execution in the coprocessor. (Unsupported in CVA6 yet)
- Connect the coprocessor to memory via the CVA6 Load and Store Unit. (Unsupported in CVA6 yet)

The coprocessor operates like another functional unit so it is connected to the CVA6 in the execute stage.

Only the 3 mandatory interfaces from the CV-X-IF specification (issue, commit and result) have been implemented. Compressed interface, Memory Interface and Memory result interface are not yet implemented in the CVA6.

3.6.1.2. Supported Parameters

The following table presents CVXIF parameters supported by CVA6.

Signal	Value	Description
X_NUM_RS	int: 2 or 3 (configurable)	Number of register file read ports that can be used by the eXtension interface
	X_ID_WIDTH	int: 3
Identification width for the eXtension interface		**X_MEM_WIDTH**
n/a (feature not supported)	Memory access width for loads/stores via the eXtension interface	
X_RFR_WIDTH	int: `XLEN` (32 or 64)	Register file read access width for the eXtension interface
	X_RFW_WIDTH	int: `XLEN` (32 or 64)

Signal	Value	Description
Register file write access width for the eXtension interface		**X_MISA**
logic[31:0]: 0x0000_0000	MISA extensions implemented on the eXtension interface	

3.6.1.3. CV-X-IF Enabling

CV-X-IF can be enabled or disabled via the `CVA6ConfigCvxifEn` parameter in the SystemVerilog source code.

3.6.1.4. Illegal instruction decoding

The CVA6 decoder module detects illegal instructions for the CVA6, prepares exception field with relevant information (exception code "ILLEGAL INSTRUCTION", instruction value).

The exception valid flag is raised in CVA6 decoder when CV-X-IF is disabled. Otherwise it is not raised at this stage because the decision belongs to the coprocessor after the offload process.

3.6.1.5. RS3 support

The number of source registers used by the CV-X-IF coprocessor is configurable with 2 or 3 source registers.

If CV-X-IF is enabled and configured with 3 source registers, a third read port is added to the CVA6 general purpose register file.

3.6.1.6. Description of interface connections between CVA6 and Coprocessor

In CVA6 execute stage, there is a new functional unit dedicated to drive the CV-X-IF interfaces. Here is *how* and *to what* CV-X-IF interfaces are connected to the CVA6.

- Issue interface::
  - Request;;
    - [verse] — Operands are connected to `issue_req.rs` signals —
    - [verse] — Scoreboard transaction id is connected to `issue_req.id` signal. Therefore scoreboard ids and offloaded instruction ids are linked together (equal in this implementation). It allows the CVA6 to do out of order execution with the coprocessor in the same way as other functional units. —
    - [verse] — Undecoded instruction is connected to `issue_req.instruction` —
    - [verse] — Valid signal for CVXIF functional unit is connected to `issue_req.valid` —
    - [verse] — All `issue_req.rs_valid` signals are set to 1. The validity of source registers is assured by the validity of valid signal sent from issue stage. —
  - Response;;
    - [verse] — If `issue_resp.accept` is set during a transaction (i.e. issue valid and ready are set), the offloaded instruction is accepted by the coprocessor and a result transaction will happen. —
    - [verse] — If `issue_resp.accept` is not set during a transaction, the offloaded instruction is illegal and an illegal instruction exception will be raised as soon as no result transaction are written on the writeback bus. —
- Commit interface::
  - [verse] — Valid signal of commit interface is connected to the valid signal of issue interface. —
  - [verse] — Id signal of commit interface is connected to issue interface id signal (i.e. scoreboard id). —
  - [verse] — Killing of offload instruction is never set. (Unsupported feature) —
  - [verse] — Therefore all accepted offloaded instructions are commited to their execution and no killing of instruction is possible in this implementation. —
- Result interface::
  - Request;;
    - [verse] — Ready signal of result interface is always set as CVA6 is always ready to take a result from coprocessor for an accepted offloaded instruction. —
  - Response;;
    - [verse] — Result response is directly connected to writeback bus of the CV-X-IF functionnal unit. —
    - [verse] — Valid signal of result interface is connected to valid signal of writeback bus. —
    - [verse] — Id signal of result interface is connected to scoreboard id of writeback bus. —

- [verse] — Write enable signal of result interface is connected to a dedicated CV-X-IF WE signal in CVA6 which signals scoreboard if a writeback should happen or not to the CVA6 register file. —
- [verse] — `exccode` and `exc` signal of result interface are connected to exception signals of writeback bus. Exception from coprocessor does not write the `tval` field in exception signal of writeback bus. —
- [verse] — Three registers are added to hold illegal instruction information in case a result transaction and a non-accepted issue transaction happen in the same cycle. Result transactions will be written to the writeback bus in this case having priority over the non-accepted instruction due to being linked to an older offloaded instruction. Once the writeback bus is free, an illegal instruction exception will be raised thanks to information held in these three registers. —

3.6.2. Coprocessor recommendations for use with CVA6's CV-X-IF

CVA6 supports all coprocessors supporting the CV-X-IF specification with the exception of :

- Coprocessor requiring the Memory interface and Memory result interface (not implemented in CVA6 yet).::
  - All memory transaction should happen via the Issue interface, i.e. Load into CVA6 register file then initialize an issue transaction.
- Coprocessor requiring the Compressed interface (not implemented in CVA6 yet).::
  - RISC-V Compressed extension (RVC) is already implemented in CVA6 User Space for custom compressed instruction is not big enough to have RVC and a custom compressed extension.
- Stateful coprocessors.::
  - CVA6 will commit on the Commit interface all its issue transactions. Speculation informations are only kept in the CVA6 and speculation process is only done in CVA6. The coprocessor shall be stateless otherwise it will not be able to revert its state if CVA6 kills an in-flight instruction (in case of mispredict or flush).

3.6.3. How to use CVA6 without CV-X-IF interface

Select a configuration with `CVA6ConfigCvxifEn` parameter disabled or change it for your configuration.

Never let the CV-X-IF interface unconnected with the `CVA6ConfigCvxifEn` parameter enabled.

3.6.4. How to design a coprocessor for the CV-X-IF interface

The team is looking for a contributor to write this section.

3.6.5. How to program a CV-X-IF coprocessor

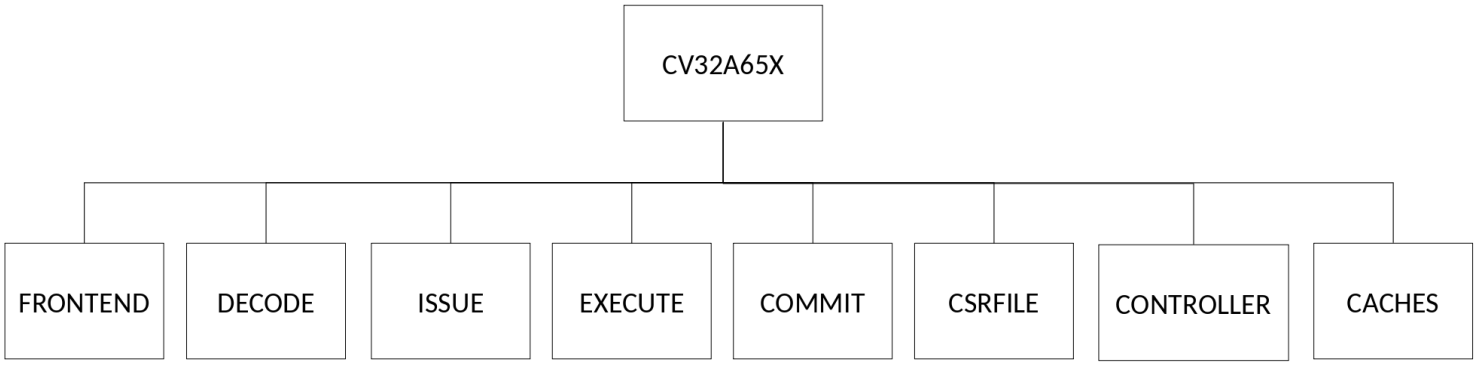
The team is looking for a contributor to write this section.

4. Architecture and Modules

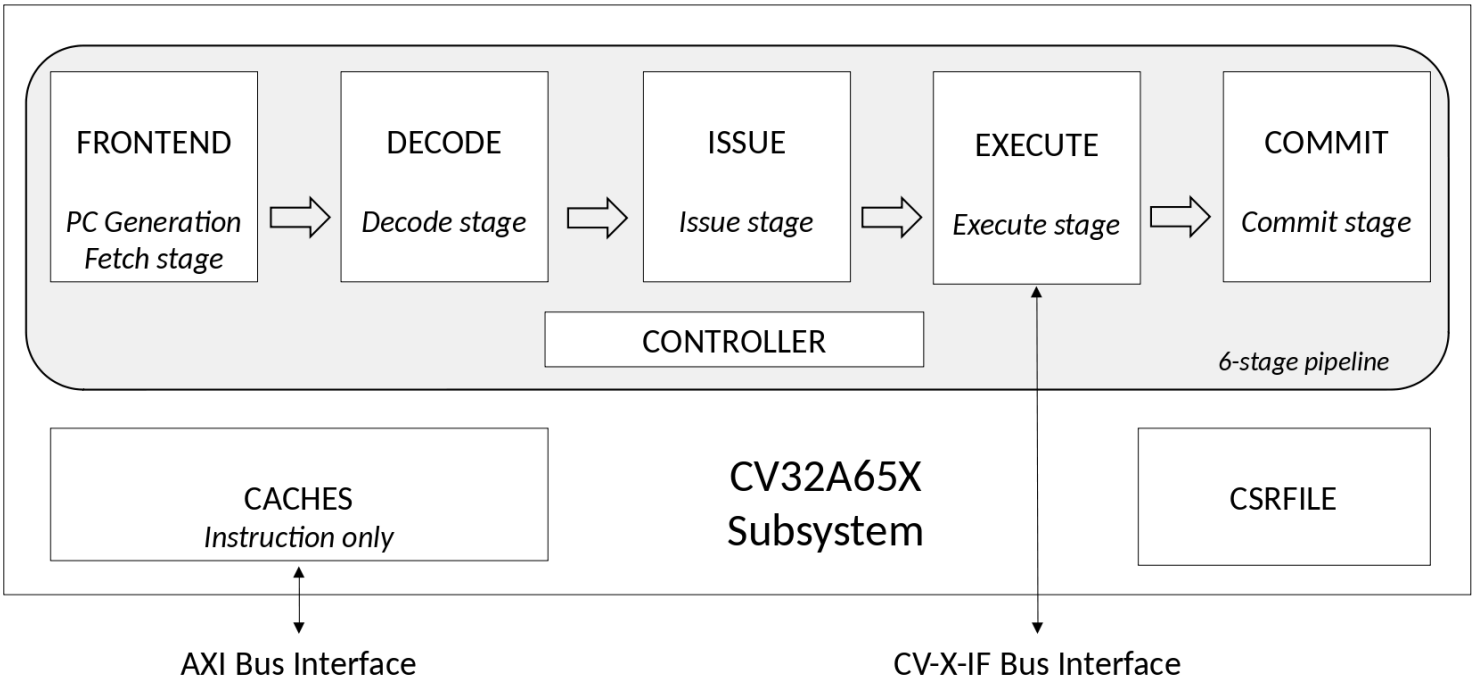
The CV32A65X is fully synthesizable. It has been designed mainly for ASIC designs, but FPGA synthesis is supported as well.

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.

The CV32A65X 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.



4.1. FRONTEND Module

4.1.1. 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. It hosts a Branch Target Buffer (BTB), a Branch History Table (BHT) and a Return Address Stack (RAS) to speculate on control flow instructions.

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 enabled, but DECODE module decodes up to one instruction per cycles.

The module is connected to:

- CACHES module provides fethed instructions to FRONTEND.

- DECODE module receives instructions from FRONTEND.
- CONTROLLER module can order to flush and to halt FRONTEND PC gen stage
- EXECUTE, CONTROLLER, CSR and COMMIT modules trigger PC jumping due to a branch misprediction, an exception, a return from an exception, a debug entry or a pipeline flush. They provides the PC next value.
- CSR module states about debug mode.

*Table 3. **frontend module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`boot_addr_i`	in	Next PC when reset	SUBSYSTEM	logic[CVA6Cfg.VLEN-1:0]
`flush_i`	in	Flush requested by FENCE, mis-predict and exception	CONTROLLER	logic
`halt_i`	in	Halt requested by WFI and Accelerate port	CONTROLLER	logic
`set_pc_commit_i`	in	Set COMMIT PC as next PC requested by FENCE, CSR side-effect and Accelerate port	CONTROLLER	logic
`pc_commit_i`	in	COMMIT PC	COMMIT	logic[CVA6Cfg.VLEN-1:0]
`ex_valid_i`	in	Exception event	COMMIT	logic
`resolved_branch_i`	in	Mispredict event and next PC	EXECUTE	bp_resolve_t
`eret_i`	in	Return from exception event	CSR	logic
`epc_i`	in	Next PC when returning from exception	CSR	logic[CVA6Cfg.VLEN-1:0]
`trap_vector_base_i`	in	Next PC when jumping into exception	CSR	logic[CVA6Cfg.VLEN-1:0]
`icache_dreq_o`	out	Handshake between CACHE and FRONTEND (fetch)	CACHES	icache_dreq_t
`icache_dreq_i`	in	Handshake between CACHE and FRONTEND (fetch)	CACHES	icache_drsp_t
`fetch_entry_o`	out	Handshake's data between fetch and decode	ID_STAGE	fetch_entry_t[CVA6Cfg.NrIssuePorts-1:0]
`fetch_entry_valid_o`	out	Handshake's valid between fetch and decode	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
`fetch_entry_ready_i`	in	Handshake's ready between fetch and decode	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

For any HW configuration,

- `flush_bp_i` input is tied to 0

As DebugEn = False,

- `set_debug_pc_i` input is tied to 0

- `debug_mode_i` input is tied to 0

4.1.2. Functionality

4.1.3. 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 Prediction:** The fetched instruction is predecoded by the instr_scan submodule. When the instruction is a control flow, three cases are considered:

  > 1. *When the instruction is a JALR which corresponds to a return (rs1 = x1 or rs1 = x5). RAS provides next PC as a prediction.*
  > 2. *When the instruction is a JALR which **does not** correspond to areturn. If BTB (Branch Target Buffer) returns a valid address, then BTBpredicts next PC. Else JALR is not considered as a control flow instruction, which will generate a mispredict.*
  > 3. *When the instruction is a conditional branch. If BHT (Branch History table) returns a valid address, then BHT predicts next PC. Else the prediction depends on the PC relative jump offset sign: if sign is negative the prediction is taken, otherwise the prediction is not taken.*

  Then the PC gen informs the Fetch stage that it performed a prediction on the PC.
- **Default:** The next 32-bit block is fetched. PC Gen fetches word boundary 32-bits block from CACHES module. And the fetch stage identifies the instructions from the 32-bits blocks.
- **Mispredict:** Misprediction are feedbacked by EX_STAGE module. 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 requests a return from an environment call, next PC takes the value of the PC of the instruction after the one pointed to by the mepc CSR.
- **Exception/Interrupt:** If an exception is triggered by CSR_REGISTER, next PC takes the value of the trap vector base address CSR.
- **Pipeline starting fetching from COMMIT PC:** When the commit stage is halted by a WFI instruction or when the pipeline has been flushed due to CSR change, next PC takes the value of the PC coming from the COMMIT submodule. As CSR instructions do not exist in a compressed form, PC is unconditionally incremented by 4.

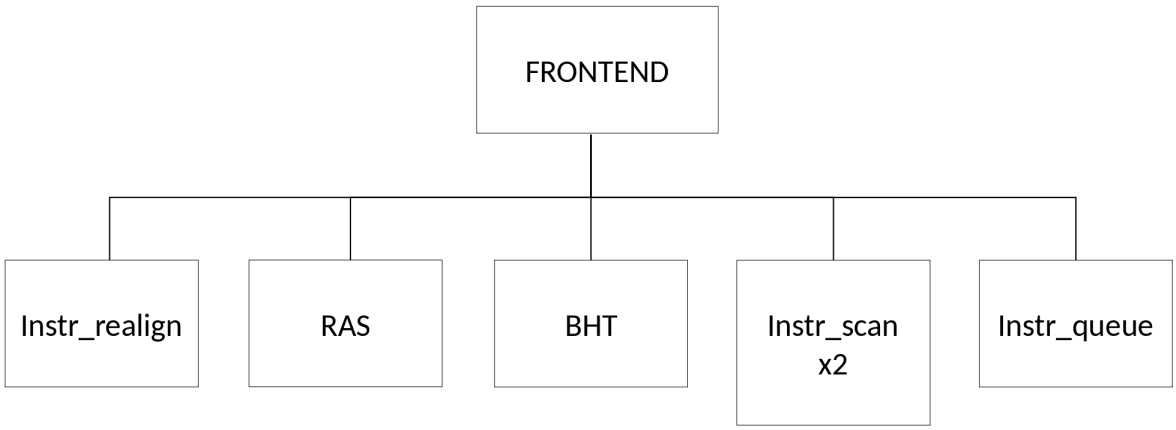
All program counters are logical addressed.

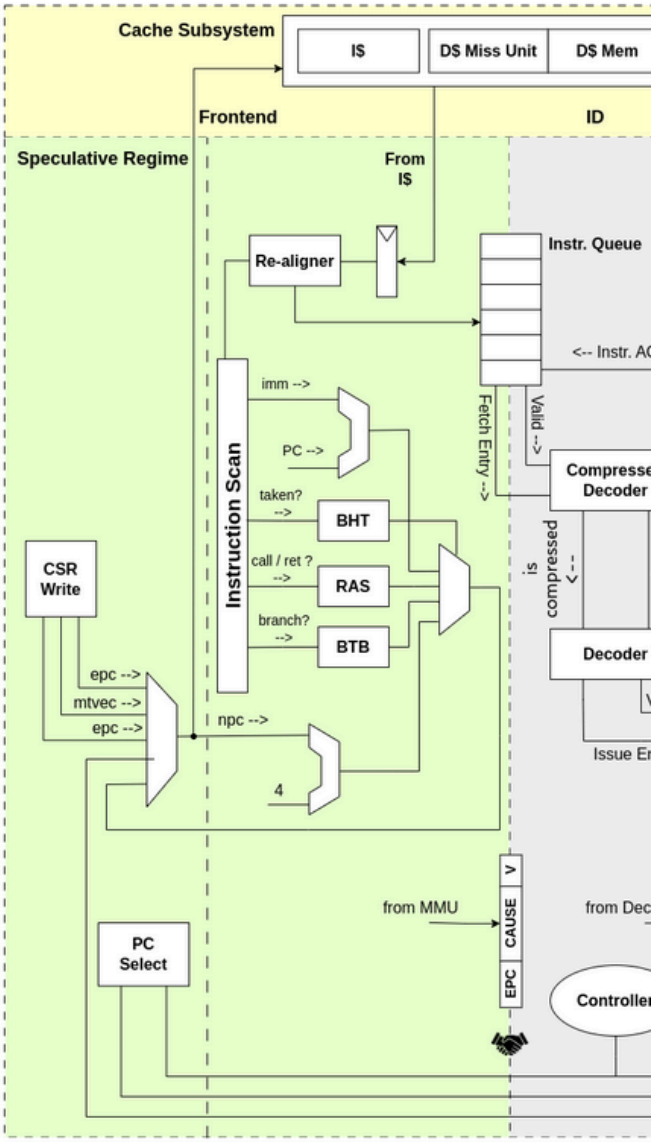
4.1.4. Fetch Stage

Fetch stage controls the CACHE module by a handshaking protocol. Fetched data is a 32-bit block with a word-aligned address. A granted fetch is processed by the 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 sends them to the DECODE module.

Memory can feedback potential exceptions which can be bus errors, invalid accesses or instruction page faults. The FRONTEND transmits the exception from CACHES to DECODE.

4.1.5. Submodules





4.1.5.1. Instr_realign submodule

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. It is possible to fetch up to two instructions per cycle when C extension is used. An not-compressed instruction can be misaligned on the block size, interleaved with two cache blocks. In that case, two cache accesses are needed to get the whole instruction. The instr_realign submodule provides at maximum two instructions per cycle when compressed extension is enabled, else one instruction per cycle. Incomplete instruction is stored in instr_realign submodule until its second half is fetched.

*Table 4. **instr_realign module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`flush_i`	in	Fetch flush request	CONTROLLER	logic
`valid_i`	in	32-bit block is valid	CACHE	logic
`serving_unaligned_o`	out	Instruction is unaligned	FRONTEND	logic
`address_i`	in	32-bit block address	CACHE	logic[CVA6Cfg.VLEN-1:0]
`data_i`	in	32-bit block	CACHE	logic[CVA6Cfg.FETCH_WIDTH-1:0]

Signal	IO	Description	connexion	Type
valid_o	out	instruction is valid	FRONTEND	logic[CVA6Cfg.INSTR_PER_FETCH-1:0]
addr_o	out	Instruction address	FRONTEND	logic[CVA6Cfg.INSTR_PER_FETCH-1:0][CVA6Cfg.VLEN-1:0]
instr_o	out	Instruction	instr_scan&instr_queue	logic[CVA6Cfg.INSTR_PER_FETCH-1:0][31:0]

4.1.5.2. Instr_queue submodule

The instr_queue receives mutliple instructions from instr_realign submodule 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. 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.

*Table 5. **instr_queue module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	Fetch flush request	CONTROLLER	logic
instr_i	in	Instruction	instr_realign	logic[CVA6Cfg.INSTR_PER_FETCH-1:0][31:0]
addr_i	in	Instruction address	instr_realign	logic[CVA6Cfg.INSTR_PER_FETCH-1:0] [CVA6Cfg.VLEN-1:0]
valid_i	in	Instruction is valid	instr_realign	logic[CVA6Cfg.INSTR_PER_FETCH-1:0]
ready_o	out	Handshake's ready with CACHE	CACHE	logic
consumed_o	out	Indicates instructions consummed, or popped by ID_STAGE	FRONTEND	logic[CVA6Cfg.INSTR_PER_FETCH-1:0]
exception_i	in	Exception (which is page-table fault)	CACHE	ariane_pkg::frontend_exception_t
exception_addr_i	in	Exception address	CACHE	logic[CVA6Cfg.VLEN-1:0]
predict_address_i	in	Branch predict	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
cf_type_i	in	Instruction predict address	FRONTEND	ariane_pkg::cf_t[CVA6Cfg.INSTR_PER_FETCH-1:0]
replay_o	out	Replay instruction because one of the FIFO was full	FRONTEND	logic
replay_addr_o	out	Address at which to replay the fetch	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
fetch_entry_o	out	Handshake's data with ID_STAGE	ID_STAGE	fetch_entry_t[CVA6Cfg.NrIssuePorts-1:0]

Signal	IO	Description	connexion	Type
fetch_entry_valid_o	out	Handshake's valid with ID_STAGE	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
fetch_entry_ready_i	in	Handshake's ready with ID_STAGE	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVH = False,

- exception_gpaddr_i input is tied to 0
- exception_tinst_i input is tied to 0
- exception_gva_i input is tied to 0

4.1.5.3. instr_scan submodule

As compressed extension is enabled, two instr_scan are instantiated to handle up to two instructions per cycle.

Each instr_scan submodule pre-decodes the fetched instructions coming from the instr_realign module, instructions could be compressed or not. The instr_scan submodule is a flox controler which provides the intruction type: branch, jump, return, jalr, imm, call or others. These outputs are used by the branch prediction feature.

*Table 6. **instr_scan module** IO ports*

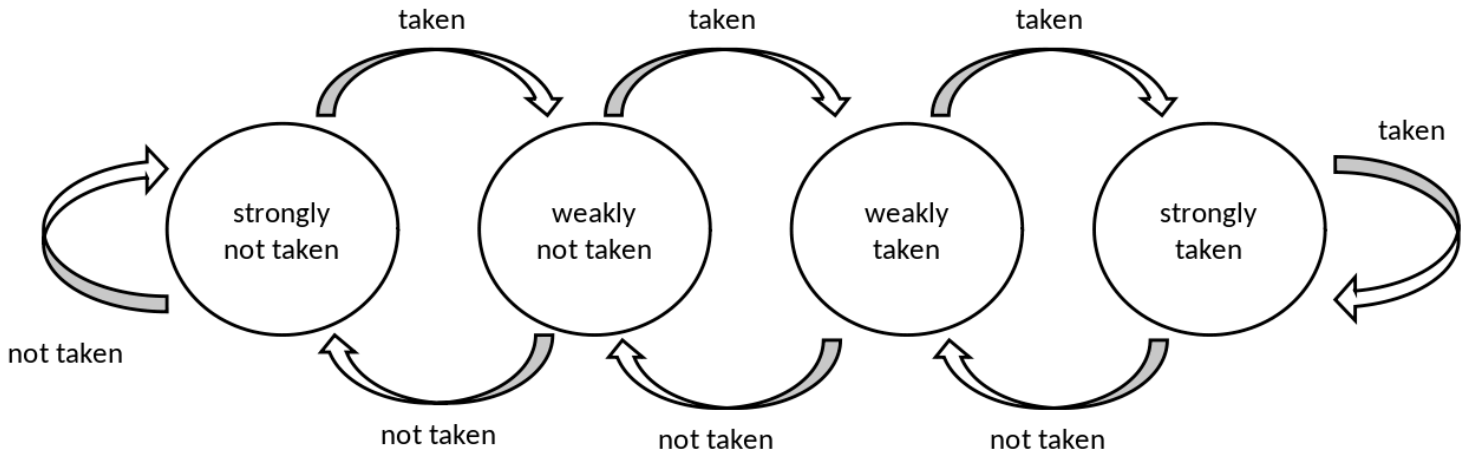
Signal	IO	Description	connexion	Type
instr_i	in	Instruction to be predecoded	instr_realign	logic[31:0]
rvi_return_o	out	Return instruction	FRONTEND	logic
rvi_call_o	out	JAL instruction	FRONTEND	logic
rvi_branch_o	out	Branch instruction	FRONTEND	logic
rvi_jalr_o	out	JALR instruction	FRONTEND	logic
rvi_jump_o	out	Unconditional jump instruction	FRONTEND	logic
rvi_imm_o	out	Instruction immediat	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
rvc_branch_o	out	Branch compressed instruction	FRONTEND	logic
rvc_jump_o	out	Unconditional jump compressed instruction	FRONTEND	logic
rvc_jr_o	out	JR compressed instruction	FRONTEND	logic
rvc_return_o	out	Return compressed instruction	FRONTEND	logic
rvc_jalr_o	out	JALR compressed instruction	FRONTEND	logic
rvc_call_o	out	JAL compressed instruction	FRONTEND	logic
rvc_imm_o	out	Instruction compressed immediat	FRONTEND	logic[CVA6Cfg.VLEN-1:0]

4.1.5.4. BHT (Branch History Table) submodule

BHT is implemented as a memory which is composed of **BHTDepth configuration parameter** entries. The lower address bits of the virtual address point to the memory entry.

When a branch instruction is resolved by the EX_STAGE module, the branch PC and the taken (or not taken) status information is stored in the Branch History Table.

The Branch History Table is a table of two-bit saturating counters 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 instructions as shown in the following figure.



When a branch instruction is pre-decoded by instr_scan submodule, the BHT valids whether the PC address is in the BHT and provides the taken or not prediction.

The BHT is never flushed.

*Table 7. **bht module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`vpc_i`	in	Virtual PC	CACHE	logic[CVA6Cfg.VLEN-1:0]
`bht_update_i`	in	Update bht with resolved address	EXECUTE	bht_update_t
`bht_prediction_o`	out	Prediction from bht	FRONTEND	ariane_pkg::bht_prediction_t[CVA6Cfg.INSTR_PER_FETCH-1:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

For any HW configuration,

- `flush_bp_i` input is tied to 0

As DebugEn = False,

- `debug_mode_i` input is tied to 0

4.1.5.5. BTB (Branch Target Buffer) submodule

BTB is implemented as an array which is composed of **BTBDepth configuration parameter** entries. The lower address bits of the virtual address point to the memory entry.

When an JALR instruction is found mispredicted by the EX_STAGE module, the JALR PC and the target address are stored into the BTB.

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

The BTB is never flushed.

*Table 8. **btb module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
vpc_i	in	Virtual PC	CACHE	logic[CVA6Cfg.VLEN-1:0]
btb_update_i	in	Update BTB with resolved address	EXECUTE	btb_update_t
btb_prediction_o	out	BTB Prediction	FRONTEND	btb_prediction_t[CVA6Cfg.INSTR_PER_FETCH-1:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

For any HW configuration,

- `flush_bp_i` input is tied to 0

As DebugEn = False,

- `debug_mode_i` input is tied to 0

4.1.5.6. RAS (Return Address Stack) submodule

RAS is implemented as a LIFO which is composed of **RASDepth configuration parameter** entries.

When a JAL instruction is pre-decoded by the instr_scan, the PC of the instruction following JAL instruction is pushed into the RAS when the JAL instruction is added to the instruction queue.

When a JALR instruction which corresponds to a return (rs1 = x1 or rs1 = x5) is pre-decoded by the instr_scan, the predicted return address is popped from the RAS when the JALR instruction is added to the instruction queue. If the predicted return address is wrong due for instance to speculation or RAS depth limitation, a mis-repdiction will be generated.

The RAS is never flushed.

*Table 9. **ras module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
push_i	in	Push address in RAS	FRONTEND	logic
pop_i	in	Pop address from RAS	FRONTEND	logic
data_i	in	Data to be pushed	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
data_o	out	Popped data	FRONTEND	ras_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

For any HW configuration,

- `flush_bp_i` input is tied to 0

4.2. ID_STAGE Module

4.2.1. Description

The ID_STAGE module implements the decode stage of the pipeline. Its main purpose is to decode RISC-V instructions coming from FRONTEND module (fetch stage) and send them to the ISSUE_STAGE module (issue stage).

The compressed_decoder module checks whether the incoming instruction is compressed and output the corresponding uncompressed instruction. Then the decoder module decodes the instruction and send it to the issue stage.

The module is connected to:

- CONTROLLER module can flush ID_STAGE decode stage
- FRONTEND module sends instrution to ID_STAGE module
- ISSUE module receives the decoded instruction from ID_STAGE module
- CSR_REGFILE module sends status information about privilege mode, traps, extension support.

*Table 10. **id_stage module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	Fetch flush request	CONTROLLER	logic
fetch_entry_i	in	Handshake's data between fetch and decode	FRONTEND	fetch_entry_t[CVA6Cfg.NrIssuePorts-1:0]
fetch_entry_valid_i	in	Handshake's valid between fetch and decode	FRONTEND	logic[CVA6Cfg.NrIssuePorts-1:0]
fetch_entry_ready_o	out	Handshake's ready between fetch and decode	FRONTEND	logic[CVA6Cfg.NrIssuePorts-1:0]
issue_entry_o	out	Handshake's data between decode and issue	ISSUE	scoreboard_entry_t[CVA6Cfg.NrIssuePorts-1:0]
orig_instr_o	out	Instruction value	ISSUE	logic[CVA6Cfg.NrIssuePorts-1:0][31:0]
issue_entry_valid_o	out	Handshake's valid between decode and issue	ISSUE	logic[CVA6Cfg.NrIssuePorts-1:0]
is_ctrl_flow_o	out	Report if instruction is a control flow instruction	ISSUE	logic[CVA6Cfg.NrIssuePorts-1:0]
issue_instr_ack_i	in	Handshake's acknowlege between decode and issue	ISSUE	logic[CVA6Cfg.NrIssuePorts-1:0]
irq_i	in	Level sensitive (async) interrupts	SUBSYSTEM	logic[1:0]
irq_ctrl_i	in	Interrupt control status	CSR_REGFILE	irq_ctrl_t
hart_id_i	in	none	none	logic[CVA6Cfg.XLEN-1:0]
compressed_ready_i	in	none	none	logic
compressed_resp_i	in	none	none	x_compressed_resp_t
compressed_valid_o	out	none	none	logic
compressed_req_o	out	none	none	x_compressed_req_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As DebugEn = False,

- `debug_req_i` input is tied to 0
- `debug_mode_i` input is tied to 0

As IsRVFI = 0,

- `rvfi_is_compressed_o` output is tied to 0

As PRIV = MachineOnly,

- `priv_lvl_i` input is tied to MachineMode

- `tvm_i` input is tied to 0
- `tw_i` input is tied to 0
- `tsr_i` input is tied to 0

As RVH = False,

- `v_i` input is tied to 0
- `vfs_i` input is tied to 0
- `vtw_i` input is tied to 0
- `hu_i` input is tied to 0

As RVF = 0,

- `fs_i` input is tied to 0
- `frm_i` input is tied to 0

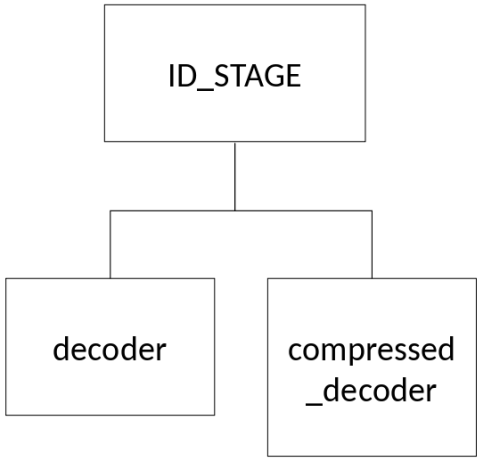
As RVV = False,

- `vs_i` input is tied to 0

4.2.2. Functionality

TO BE COMPLETED

4.2.3. Submodules



4.2.3.1. Compressed_decoder

The compressed_decoder module decompresses all the compressed instructions taking a 16-bit compressed instruction and expanding it to its 32-bit equivalent. All compressed instructions have a 32-bit equivalent.

*Table 11. **compressed_decoder module** IO ports*

Signal	IO	Description	connexion	Type
`instr_i`	in	Input instruction coming from fetch stage	FRONTEND	logic[31:0]
`instr_o`	out	Output instruction in uncompressed format	decoder	logic[31:0]
`illegal_instr_o`	out	Input instruction is illegal	decoder	logic
`is_macro_instr_o`	out	Output instruction is macro	decoder	logic
`is_compressed_o`	out	Output instruction is compressed	decoder	logic

4.2.3.2. Decoder

The decoder module takes the output of compressed_decoder module and decodes it. It transforms the instruction to the most fundamental control structure in pipeline, a scoreboard entry.

The scoreboard entry contains an exception entry which is composed of a valid field, a cause and a value called TVAL. As TVALEn configuration parameter is zero, the TVAL field is not implemented.

A potential illegal instruction exception can be detected during decoding. If no exception has happened previously in fetch stage, the decoder will valid the exception and add the cause and tval value to the scoreboard entry.

*Table 12. **decoder module** IO ports*

Signal	IO	Description	connexion	Type
`pc_i`	in	PC from fetch stage	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
`is_compressed_i`	in	Is a compressed instruction	compressed_decoder	logic
`compressed_instr_i`	in	Compressed form of instruction	FRONTEND	logic[15:0]
`is_illegal_i`	in	Illegal compressed instruction	compressed_decoder	logic
`instruction_i`	in	Instruction from fetch stage	FRONTEND	logic[31:0]
`is_macro_instr_i`	in	Is a macro instruction	macro_decoder	logic
`is_last_macro_instr_i`	in	Is a last macro instruction	macro_decoder	logic
`is_double_rd_macro_instr_i`	in	Is mvsa01/mva01s macro instruction	macro_decoder	logic
`branch_predict_i`	in	Is a branch predict instruction	FRONTEND	branchpredict_sbe_t
`ex_i`	in	If an exception occured in fetch stage	FRONTEND	exception_t
`irq_i`	in	Level sensitive (async) interrupts	SUBSYSTEM	logic[1:0]
`irq_ctrl_i`	in	Interrupt control status	CSR_REGFILE	irq_ctrl_t
`instruction_o`	out	Instruction to be added to scoreboard entry	ISSUE_STAGE	scoreboard_entry_t
`orig_instr_o`	out	Instruction	ISSUE_STAGE	logic[31:0]
`is_control_flow_instr_o`	out	Is a control flow instruction	ISSUE_STAGE	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As DebugEn = False,

- `debug_req_i` input is tied to 0
- `debug_mode_i` input is tied to 0

As PRIV = MachineOnly,

- `priv_lvl_i` input is tied to MachineMode
- `tvm_i` input is tied to 0
- `tw_i` input is tied to 0
- `tsr_i` input is tied to 0

As RVH = False,

- `v_i` input is tied to 0
- `vfs_i` input is tied to 0
- `vtw_i` input is tied to 0

- `hu_i` input is tied to 0

As RVF = 0,

- `fs_i` input is tied to 0
- `frm_i` input is tied to 0

As RVV = False,

- `vs_i` input is tied to 0

4.3. ISSUE_STAGE Module

4.3.1. Description

The execution can be roughly divided into four parts: issue(1), read operands(2), execute(3) and write-back(4). The ISSUE_STAGE module handles step one, two and four. The ISSUE_STAGE module receives the decoded instructions and issues them to the various functional units.

A data structure called scoreboard is used to keep track of data related to the issue instruction: which functional unit and which destination register they are. The scoreboard handle the write-back data received from the COMMIT_STAGE module.

Furthermore it contains the CPU's register file.

The module is connected to:

- TO BE COMPLETED

*Table 13. **issue_stage module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`flush_unissued_instr_i`	in	TO_BE_COMPLETED	CONTROLLER	logic
`flush_i`	in	TO_BE_COMPLETED	CONTROLLER	logic
`decoded_instr_i`	in	Handshake's data with decode stage	ID_STAGE	scoreboard_entry_t[CVA6Cfg.NrIssuePorts-1:0]
`orig_instr_i`	in	instruction value	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0][31:0]
`decoded_instr_valid_i`	in	Handshake's valid with decode stage	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
`is_ctrl_flow_i`	in	Is instruction a control flow instruction	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
`decoded_instr_ack_o`	out	Handshake's acknowlege with decode stage	ID_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
`rs1_forwarding_o`	out	rs1 forwarding	EX_STAGE	[CVA6Cfg.NrIssuePorts-1:0][CVA6Cfg.VLEN-1:0]
`rs2_forwarding_o`	out	rs2 forwarding	EX_STAGE	[CVA6Cfg.NrIssuePorts-1:0][CVA6Cfg.VLEN-1:0]
`fu_data_o`	out	FU data useful to execute instruction	EX_STAGE	fu_data_t[CVA6Cfg.NrIssuePorts-1:0]
`pc_o`	out	Program Counter	EX_STAGE	logic[CVA6Cfg.VLEN-1:0]
`is_compressed_instr_o`	out	Is compressed instruction	EX_STAGE	logic
`flu_ready_i`	in	Fixed Latency Unit is ready	EX_STAGE	logic

Signal	IO	Description	connexion	Type
alu_valid_o	out	ALU FU is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
resolve_branch_i	in	Signaling that we resolved the branch	EX_STAGE	logic
lsu_ready_i	in	Load store unit FU is ready	EX_STAGE	logic
lsu_valid_o	out	Load store unit FU is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
branch_valid_o	out	Branch unit is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
branch_predict_o	out	Information of branch prediction	EX_STAGE	branchpredict_sbe_t
mult_valid_o	out	Mult FU is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
alu2_valid_o	out	ALU2 FU is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
csr_valid_o	out	CSR is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
xfu_valid_o	out	CVXIF FU is valid	EX_STAGE	logic[CVA6Cfg.NrIssuePorts-1:0]
xfu_ready_i	in	CVXIF is FU ready	EX_STAGE	logic
x_off_instr_o	out	CVXIF offloader instruction value	EX_STAGE	logic[31:0]
hart_id_i	in	CVA6 Hart ID	SUBSYSTEM	logic[CVA6Cfg.XLEN-1:0]
x_issue_ready_i	in	none	none	logic
x_issue_resp_i	in	none	none	x_issue_resp_t
x_issue_valid_o	out	none	none	logic
x_issue_req_o	out	none	none	x_issue_req_t
x_register_ready_i	in	none	none	logic
x_register_valid_o	out	none	none	logic
x_register_o	out	none	none	x_register_t
x_commit_valid_o	out	none	none	logic
x_commit_o	out	none	none	x_commit_t
x_transaction_rejected_o	out	CVXIF Transaction rejected → instruction is illegal	EX_STAGE	logic
trans_id_i	in	Transaction ID	EX_STAGE	logic[CVA6Cfg.NrWbPorts-1:0] [CVA6Cfg.TRANS_ID_BITS-1:0]
resolved_branch_i	in	The branch engine uses the write back from the ALU	EX_STAGE	bp_resolve_t
wbdata_i	in	TO_BE_COMPLETED	EX_STAGE	logic[CVA6Cfg.NrWbPorts-1:0] [CVA6Cfg.XLEN-1:0]
ex_ex_i	in	exception from execute stage or CVXIF	EX_STAGE	exception_t[CVA6Cfg.NrWbPorts-1:0]
wt_valid_i	in	TO_BE_COMPLETED	EX_STAGE	logic[CVA6Cfg.NrWbPorts-1:0]

Signal	IO	Description	connexion	Type
x_we_i	in	CVXIF write enable	EX_STAGE	logic
x_rd_i	in	CVXIF destination register	ISSUE_STAGE	logic[4:0]
waddr_i	in	TO_BE_COMPLETED	EX_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0][4:0]
wdata_i	in	TO_BE_COMPLETED	EX_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0] [CVA6Cfg.XLEN-1:0]
we_gpr_i	in	GPR write enable	EX_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
commit_instr_o	out	Instructions to commit	COMMIT_STAGE	scoreboard_entry_t[CVA6Cfg.NrCommitPorts-1:0]
commit_drop_o	out	Instruction is cancelled	COMMIT_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
commit_ack_i	in	Commit acknowledge	COMMIT_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As PerfCounterEn = 0,

- `sb_full_o` output is tied to 0
- `stall_issue_o` output is tied to 0

As EnableAccelerator = 0,

- `stall_i` input is tied to 0
- `issue_instr_o` output is tied to 0
- `issue_instr_hs_o` output is tied to 0

As RVH = False,

- `tinst_o` output is tied to 0

As RVF = 0,

- `fpu_ready_i` input is tied to 0
- `fpu_valid_o` output is tied to 0
- `fpu_fmt_o` output is tied to 0
- `fpu_rm_o` output is tied to 0
- `we_fpr_i` input is tied to 0

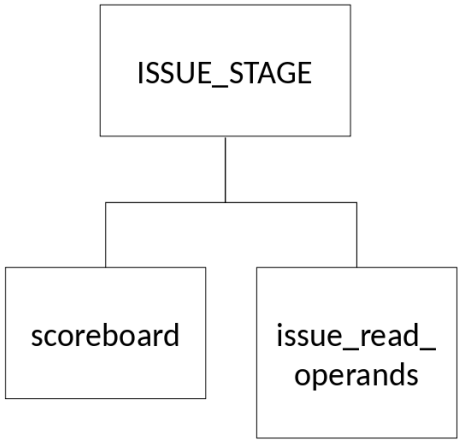
As IsRVFI = 0,

- `rvfi_issue_pointer_o` output is tied to 0
- `rvfi_commit_pointer_o` output is tied to 0

4.3.2. Functionality

TO BE COMPLETED

4.3.3. Submodules



4.3.3.1. Scoreboard

The scoreboard contains a FIFO to store the decoded instructions. Issued instruction is pushed to the FIFO if it is not full. It indicates which registers are going to be clobbered by a previously issued instruction.

*Table 14. **scoreboard module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
sb_full_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
flush_unissued_instr_i	in	Flush only un-issued instructions	TO_BE_COMPLETED	logic
flush_i	in	Flush whole scoreboard	TO_BE_COMPLETED	logic
rd_clobber_gpr_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	ariane_pkg::fu_t[2**ariane_pkg::REG_ADDR_SIZE-1:0]
rd_clobber_fpr_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	ariane_pkg::fu_t[2**ariane_pkg::REG_ADDR_SIZE-1:0]
x_transaction_accepted_i	in	none	none	logic
x_issue_writeback_i	in	none	none	logic
x_id_i	in	none	none	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
rs1_i	in	rs1 operand address	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0] [ariane_pkg::REG_ADDR_SIZE-1:0]
rs1_o	out	rs1 operand	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0][CVA6Cfg.XLEN-1:0]
rs1_valid_o	out	rs1 operand is valid	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0]
rs2_i	in	rs2 operand address	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0] [ariane_pkg::REG_ADDR_SIZE-1:0]
rs2_o	out	rs2 operand	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0][CVA6Cfg.XLEN-1:0]
rs2_valid_o	out	rs2 operand is valid	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0]
rs3_i	in	rs3 operand address	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0] [ariane_pkg::REG_ADDR_SIZE-1:0]

Signal	IO	Description	connexion	Type
rs3_o	out	rs3 operand	issue_read_operands	rs3_len_t[CVA6Cfg.NrIssuePorts-1:0]
rs3_valid_o	out	rs3 operand is valid	issue_read_operands	logic[CVA6Cfg.NrIssuePorts-1:0]
commit_instr_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	scoreboard_entry_t[CVA6Cfg.NrCommitPorts-1:0]
commit_drop_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrCommitPorts-1:0]
commit_ack_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrCommitPorts-1:0]
decoded_instr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	scoreboard_entry_t[CVA6Cfg.NrIssuePorts-1:0]
orig_instr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0][31:0]
decoded_instr_valid_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
decoded_instr_ack_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
orig_instr_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0][31:0]
issue_instr_valid_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
issue_ack_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
resolved_branch_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	bp_resolve_t
trans_id_i	in	Transaction ID at which to write the result back	TO_BE_COMPLETED	logic[CVA6Cfg.NrWbPorts-1:0] [CVA6Cfg.TRANS_ID_BITS-1:0]
wbdata_i	in	Results to write back	TO_BE_COMPLETED	logic[CVA6Cfg.NrWbPorts-1:0][CVA6Cfg.XLEN-1:0]
ex_i	in	Exception from a functional unit (e.g.: ld/st exception)	TO_BE_COMPLETED	exception_t[CVA6Cfg.NrWbPorts-1:0]
wt_valid_i	in	Indicates valid results	TO_BE_COMPLETED	logic[CVA6Cfg.NrWbPorts-1:0]
x_we_i	in	Cvxif we for writeback	TO_BE_COMPLETED	logic
x_rd_i	in	CVXIF destination register	ISSUE_STAGE	logic[4:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As EnableAccelerator = 0,

- `issue_instr_o` output is tied to 0

As IsRVFI = 0,

- `rvfi_issue_pointer_o` output is tied to 0
- `rvfi_commit_pointer_o` output is tied to 0

4.3.3.2. Issue_read_operands

TO BE COMPLETED

*Table 15. **issue_read_operands module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
flush_i	in	Flush	CONTROLLER	logic
issue_instr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	scoreboard_entry_t[CVA6Cfg.NrIssuePorts-1:0]
orig_instr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0][31:0]
issue_instr_valid_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
issue_ack_o	out	Issue stage acknowledge	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
rs1_o	out	rs1 operand address	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0] [REG_ADDR_SIZE-1:0]
rs1_i	in	rs1 operand	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0] [CVA6Cfg.XLEN-1:0]
rs1_valid_i	in	rs1 operand is valid	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0]
rs2_o	out	rs2 operand address	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0] [REG_ADDR_SIZE-1:0]
rs2_i	in	rs2 operand	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0] [CVA6Cfg.XLEN-1:0]
rs2_valid_i	in	rs2 operand is valid	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0]
rs3_o	out	rs3 operand address	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0] [REG_ADDR_SIZE-1:0]
rs3_i	in	rs3 operand	scoreboard	rs3_len_t[CVA6Cfg.NrIssuePorts-1:0]
rs3_valid_i	in	rs3 operand is valid	scoreboard	logic[CVA6Cfg.NrIssuePorts-1:0]
rd_clobber_gpr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	fu_t[2**REG_ADDR_SIZE-1:0]
rd_clobber_fpr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	fu_t[2**REG_ADDR_SIZE-1:0]
fu_data_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	fu_data_t[CVA6Cfg.NrIssuePorts-1:0]
rs1_forwarding_o	out	Unregistered version of fu_data_o.operanda	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0] [CVA6Cfg.XLEN-1:0]
rs2_forwarding_o	out	Unregistered version of fu_data_o.operandb	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0] [CVA6Cfg.XLEN-1:0]
pc_o	out	Instruction pc	TO_BE_COMPLETED	logic[CVA6Cfg.VLEN-1:0]
is_compressed_instr_o	out	Is compressed instruction	TO_BE_COMPLETED	logic
flu_ready_i	in	Fixed Latency Unit ready to accept new request	TO_BE_COMPLETED	logic
alu_valid_o	out	ALU output is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
branch_valid_o	out	Branch instruction is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
branch_predict_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	branchpredict_sbe_t
lsu_ready_i	in	Load Store Unit is ready	TO_BE_COMPLETED	logic
lsu_valid_o	out	Load Store Unit result is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
mult_valid_o	out	Mult result is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]

Signal	IO	Description	connexion	Type
alu2_valid_o	out	ALU output is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
csr_valid_o	out	CSR result is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
cvxif_valid_o	out	CVXIF result is valid	TO_BE_COMPLETED	logic[CVA6Cfg.NrIssuePorts-1:0]
cvxif_ready_i	in	CVXIF is ready	TO_BE_COMPLETED	logic
cvxif_off_instr_o	out	CVXIF offloaded instruction	TO_BE_COMPLETED	logic[31:0]
hart_id_i	in	CVA6 Hart ID	SUBSYSTEM	logic[CVA6Cfg.XLEN-1:0]
x_issue_ready_i	in	none	none	logic
x_issue_resp_i	in	none	none	x_issue_resp_t
x_issue_valid_o	out	none	none	logic
x_issue_req_o	out	none	none	x_issue_req_t
x_register_ready_i	in	none	none	logic
x_register_valid_o	out	none	none	logic
x_register_o	out	none	none	x_register_t
x_commit_valid_o	out	none	none	logic
x_commit_o	out	none	none	x_commit_t
x_transaction_accepted_o	out	none	none	logic
x_transaction_rejected_o	out	none	none	logic
x_issue_writeback_o	out	none	none	logic
x_id_o	out	none	none	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
waddr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrCommitPorts-1:0][4:0]
wdata_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrCommitPorts-1:0] [CVA6Cfg.XLEN-1:0]
we_gpr_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic[CVA6Cfg.NrCommitPorts-1:0]
stall_issue_o	out	Stall signal, we do not want to fetch any more entries	TO_BE_COMPLETED	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As EnableAccelerator = 0,

- `stall_i` input is tied to 0

As RVH = False,

- `tinst_o` output is tied to 0

As RVF = 0,

- `fpu_ready_i` input is tied to 0
- `fpu_valid_o` output is tied to 0
- `fpu_fmt_o` output is tied to 0
- `fpu_rm_o` output is tied to 0

- `we_fpr_i` input is tied to 0

4.4. EX_STAGE Module

4.4.1. Description

The EX_STAGE module is a logical stage which implements the execute stage. It encapsulates the following functional units: ALU, Branch Unit, CSR buffer, Mult, load and store and CVXIF.

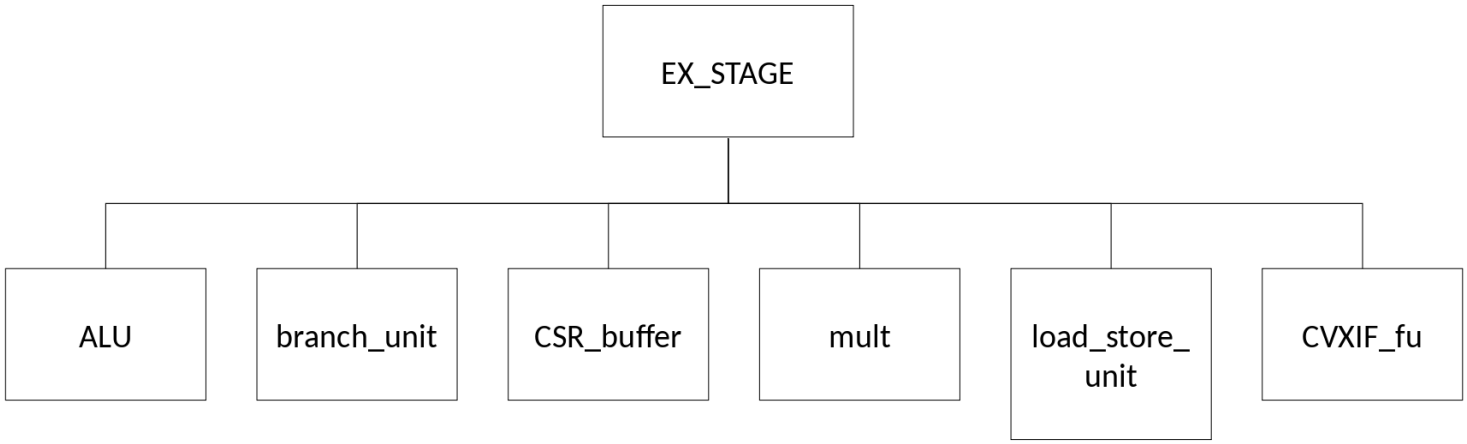
The module is connected to:

- ID_STAGE module provides scoreboard entry. *

4.4.2. Functionality

TO BE COMPLETED

4.4.3. Submodules



4.4.3.1. alu

The arithmetic logic unit (ALU) is a small piece of hardware which performs 32 and 64-bit arithmetic and bitwise operations: subtraction, addition, shifts, comparisons...It always completes its operation in a single cycle.

*Table 16. **alu module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`fu_data_i`	in	FU data needed to execute instruction	ISSUE_STAGE	fu_data_t
`result_o`	out	ALU result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
`alu_branch_res_o`	out	ALU branch compare result	branch_unit	logic

4.4.3.2. branch_unit

The branch unit module manages all kinds of control flow changes i.e.: conditional and unconditional jumps. It calculates the target address and decides whether to take the branch or not. It also decides if a branch was mis-predicted or not and reports corrective actions to the pipeline stages.

*Table 17. **branch_unit module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
fu_data_i	in	FU data needed to execute instruction	ISSUE_STAGE	fu_data_t
pc_i	in	Instruction PC	ISSUE_STAGE	logic[CVA6Cfg.VLEN-1:0]
is_compressed_instr_i	in	Instruction is compressed	ISSUE_STAGE	logic
branch_valid_i	in	Branch unit instruction is valid	ISSUE_STAGE	logic
branch_comp_res_i	in	ALU branch compare result	ALU	logic
branch_result_o	out	Brach unit result	ISSUE_STAGE	logic[CVA6Cfg.VLEN-1:0]
branch_predict_i	in	Information of branch prediction	ISSUE_STAGE	branchpredict_sbe_t
resolved_branch_o	out	Signaling that we resolved the branch	ISSUE_STAGE	bp_resolve_t
resolve_branch_o	out	Branch is resolved, new entries can be accepted by scoreboard	ID_STAGE	logic
branch_exception_o	out	Branch exception out	TO_BE_COMPLETED	exception_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVH = False,

- v_i input is tied to 0

As DebugEn = False,

- debug_mode_i input is tied to 0

4.4.3.3. CSR_buffer

The CSR buffer module stores the CSR address at which the instruction is going to read/write. As the CSR instruction alters the processor architectural state, this instruction has to be buffered until the commit stage decides to execute the instruction.

*Table 18. **csr_buffer module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	Flush CSR	CONTROLLER	logic
fu_data_i	in	FU data needed to execute instruction	ISSUE_STAGE	fu_data_t
csr_ready_o	out	CSR FU is ready	ISSUE_STAGE	logic
csr_valid_i	in	CSR instruction is valid	ISSUE_STAGE	logic
csr_result_o	out	CSR buffer result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
csr_commit_i	in	commit the pending CSR OP	TO_BE_COMPLETED	logic
csr_addr_o	out	CSR address to write	COMMIT_STAGE	logic[11:0]

4.4.3.4. mult

The multiplier module supports the division and multiplication operations.

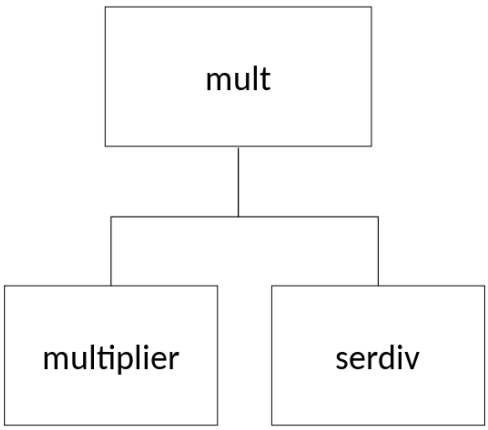


*Table 19. **mult module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	Flush	CONTROLLER	logic
fu_data_i	in	FU data needed to execute instruction	ISSUE_STAGE	fu_data_t
mult_valid_i	in	Mult instruction is valid	ISSUE_STAGE	logic
result_o	out	Mult result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
mult_valid_o	out	Mult result is valid	ISSUE_STAGE	logic
mult_ready_o	out	Mutl is ready	ISSUE_STAGE	logic
mult_trans_id_o	out	Mult transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]

4.4.3.4.1. multiplier

Multiplication is performed in two cycles and is fully pipelined.

*Table 20. **multiplier module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
trans_id_i	in	Multiplier transaction ID	Mult	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
mult_valid_i	in	Multiplier instruction is valid	Mult	logic
operation_i	in	Multiplier operation	Mult	fu_op
operand_a_i	in	A operand	Mult	logic[CVA6Cfg.XLEN-1:0]
operand_b_i	in	B operand	Mult	logic[CVA6Cfg.XLEN-1:0]
result_o	out	Multiplier result	Mult	logic[CVA6Cfg.XLEN-1:0]
mult_valid_o	out	Mutliplier result is valid	Mult	logic

Signal	IO	Description	connexion	Type
mult_ready_o	out	Multiplier FU is ready	Mult	logic
mult_trans_id_o	out	Multiplier transaction ID	Mult	logic[CVA6Cfg.TRANS_ID_BITS-1:0]

4.4.3.4.2. serdiv

The division is a simple serial divider which needs 64 cycles in the worst case.

*Table 21. **serdiv module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
id_i	in	Serdiv translation ID	Mult	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
op_a_i	in	A operand	Mult	logic[WIDTH-1:0]
op_b_i	in	B operand	Mult	logic[WIDTH-1:0]
rem	in	Serdiv operation	Mult	logic[1:0]opcode_i,//0:udiv,2:urem,1:div,3:
in_vld_i	in	Serdiv instruction is valid	Mult	logic
in_rdy_o	out	Serdiv FU is ready	Mult	logic
flush_i	in	Flush	CONTROLLER	logic
out_vld_o	out	Serdiv result is valid	Mult	logic
out_rdy_i	in	Serdiv is ready	Mult	logic
id_o	out	Serdiv transaction ID	Mult	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
res_o	out	Serdiv result	Mult	logic[WIDTH-1:0]

4.4.3.5. load_store_unit (LSU)

The load store module interfaces with the data cache (D$) to manage the load and store operations.

The LSU does not handle misaligned accesses. Misaligned accesses are double word accesses which are not aligned to a 64-bit boundary, word accesses which are not aligned to a 32-bit boundary and half word accesses which are not aligned on 16-bit boundary. If the LSU encounters a misaligned load or store, it throws a misaligned exception.

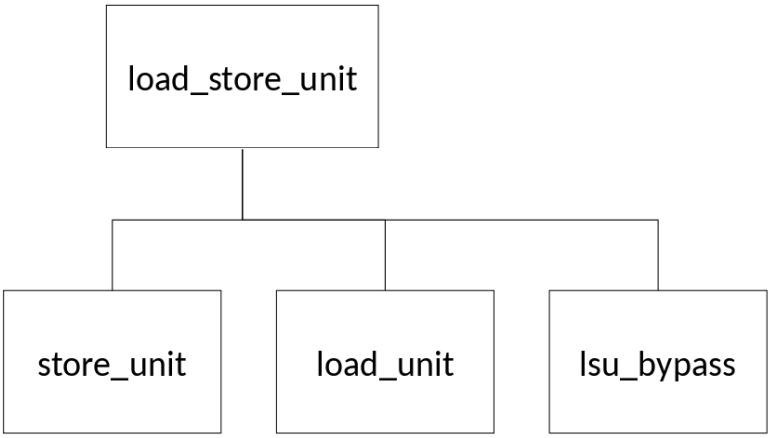


*Table 22. **load_store_unit module** IO ports*

Signal	IO	Description	connexion	Type

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
stall_st_pending_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
no_st_pending_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
fu_data_i	in	FU data needed to execute instruction	ISSUE_STAGE	fu_data_t
lsu_ready_o	out	Load Store Unit is ready	ISSUE_STAGE	logic
lsu_valid_i	in	Load Store Unit instruction is valid	ISSUE_STAGE	logic
load_trans_id_o	out	Load transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
load_result_o	out	Load result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
load_valid_o	out	Load result is valid	ISSUE_STAGE	logic
load_exception_o	out	Load exception	ISSUE_STAGE	exception_t
store_trans_id_o	out	Store transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
store_result_o	out	Store result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
store_valid_o	out	Store result is valid	ISSUE_STAGE	logic
store_exception_o	out	Store exception	ISSUE_STAGE	exception_t
commit_i	in	Commit the first pending store	TO_BE_COMPLETED	logic
commit_ready_o	out	Commit queue is ready to accept another commit request	TO_BE_COMPLETED	logic
commit_tran_id_i	in	Commit transaction ID	TO_BE_COMPLETED	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
icache_areq_i	in	Instruction cache input request	CACHES	icache_arsp_t
icache_areq_o	out	Instruction cache output request	CACHES	icache_areq_t
dcache_req_ports_i	in	Data cache request output	CACHES	dcache_req_o_t[2:0]
dcache_req_ports_o	out	Data cache request input	CACHES	dcache_req_i_t[2:0]
dcache_wbuffer_empty_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
dcache_wbuffer_not_ni_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
pmpcfg_i	in	PMP configuration	CSR_REGFILE	riscv::pmpcfg_t[CVA6Cfg.NrPMPEntries-1:0]
pmpaddr_i	in	PMP address	CSR_REGFILE	logic[CVA6Cfg.NrPMPEntries-1:0] [CVA6Cfg.PLEN-3:0]

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVA = False,

- `amo_valid_commit_i` input is tied to 0
- `amo_req_o` output is tied to 0
- `amo_resp_i` input is tied to 0

As RVH = False,

- `tinst_i` input is tied to 0
- `enable_g_translation_i` input is tied to 0
- `en_ld_st_g_translation_i` input is tied to 0
- `v_i` input is tied to 0
- `ld_st_v_i` input is tied to 0
- `csr_hs_ld_st_inst_o` output is tied to 0
- `vs_sum_i` input is tied to 0
- `vmxr_i` input is tied to 0
- `vsatp_ppn_i` input is tied to 0
- `vs_asid_i` input is tied to 0
- `hgatp_ppn_i` input is tied to 0
- `vmid_i` input is tied to 0
- `vmid_to_be_flushed_i` input is tied to 0
- `gpaddr_to_be_flushed_i` input is tied to 0
- `flush_tlb_vvma_i` input is tied to 0
- `flush_tlb_gvma_i` input is tied to 0

As RVS = False,

- `enable_translation_i` input is tied to 0
- `en_ld_st_translation_i` input is tied to 0
- `sum_i` input is tied to 0
- `mxr_i` input is tied to 0
- `satp_ppn_i` input is tied to 0
- `asid_i` input is tied to 0
- `asid_to_be_flushed_i` input is tied to 0
- `vaddr_to_be_flushed_i` input is tied to 0

As PRIV = MachineOnly,

- `priv_lvl_i` input is tied to MachineMode
- `ld_st_priv_lvl_i` input is tied to MAchineMode

As MMUPresent = 0,

- `flush_tlb_i` input is tied to 0

As PerfCounterEn = 0,

- `itlb_miss_o` output is tied to 0
- `dtlb_miss_o` output is tied to 0

As IsRVFI = 0,

- `rvfi_lsu_ctrl_o` output is tied to 0
- `rvfi_mem_paddr_o` output is tied to 0

4.4.3.5.1. store_unit

The store_unit module manages the data store operations.

As stores can be speculative, the store instructions need to be committed by ISSUE_STAGE module before possibily altering the processor state. Store buffer keeps track of store requests. Outstanding store instructions (which are speculative) are differentiated from committed stores. When ISSUE_STAGE module commits a store instruction, outstanding stores become committed.

When commit buffer is not empty, the buffer automatically tries to write the oldest store to the data cache.

Furthermore, the store_unit module provides information to the load_unit to know if an outstanding store matches addresses with a load.

*Table 23. **store_unit module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	Flush	CONTROLLER	logic
stall_st_pending_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
no_st_pending_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
store_buffer_empty_o	out	Store buffer is empty	TO_BE_COMPLETED	logic
valid_i	in	Store instruction is valid	ISSUE_STAGE	logic
lsu_ctrl_i	in	Data input	ISSUE_STAGE	lsu_ctrl_t
pop_st_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
commit_i	in	Instruction commit	TO_BE_COMPLETED	logic
commit_ready_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
valid_o	out	Store result is valid	ISSUE_STAGE	logic
trans_id_o	out	Transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
result_o	out	Store result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
ex_o	out	Store exception output	TO_BE_COMPLETED	exception_t
translation_req_o	out	Address translation request	TO_BE_COMPLETED	logic
vaddr_o	out	Virtual address	TO_BE_COMPLETED	logic[CVA6Cfg.VLEN-1:0]
paddr_i	in	Physical address	TO_BE_COMPLETED	logic[CVA6Cfg.PLEN-1:0]
ex_i	in	Exception raised before store	TO_BE_COMPLETED	exception_t
page_offset_i	in	Address to be checked	load_unit	logic[11:0]
page_offset_matches_o	out	Address check result	load_unit	logic
req_port_i	in	Data cache request	CACHES	dcache_req_o_t
req_port_o	out	Data cache response	CACHES	dcache_req_i_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVA = False,

- `amo_valid_commit_i` input is tied to 0
- `amo_req_o` output is tied to 0
- `amo_resp_i` input is tied to 0

As IsRVFI = 0,

- `rvfi_mem_paddr_o` output is tied to 0

As RVH = False,

- `tinst_o` output is tied to 0
- `hs_ld_st_inst_o` output is tied to 0
- `hlvx_inst_o` output is tied to 0

For any HW configuration,

- `dtlb_hit_i` input is tied to 1

4.4.3.5.2. load_unit

The load unit module manages the data load operations.

Before issuing a load, the load unit needs to check the store buffer for potential aliasing. It stalls until it can satisfy the current request. This means:

- Two loads to the same address are allowed.
- Two stores to the same address are allowed.
- A store after a load to the same address is allowed.
- A load after a store to the same address can only be processed if the store has already been sent to the cache i.e there is no fowarding.

After the check of the store buffer, a read request is sent to the D$ with the index field of the address (1). The load unit stalls until the D$ acknowledges this request (2). In the next cycle, the tag field of the address is sent to the D$ (3). If the load request address is non-idempotent, it stalls until the write buffer of the D$ is empty of non-idempotent requests and the store buffer is empty. It also stalls until the incoming load instruction is the next instruction to be committed. When the D$ allows the read of the data, the data is sent to the load unit and the load instruction can be committed (4).

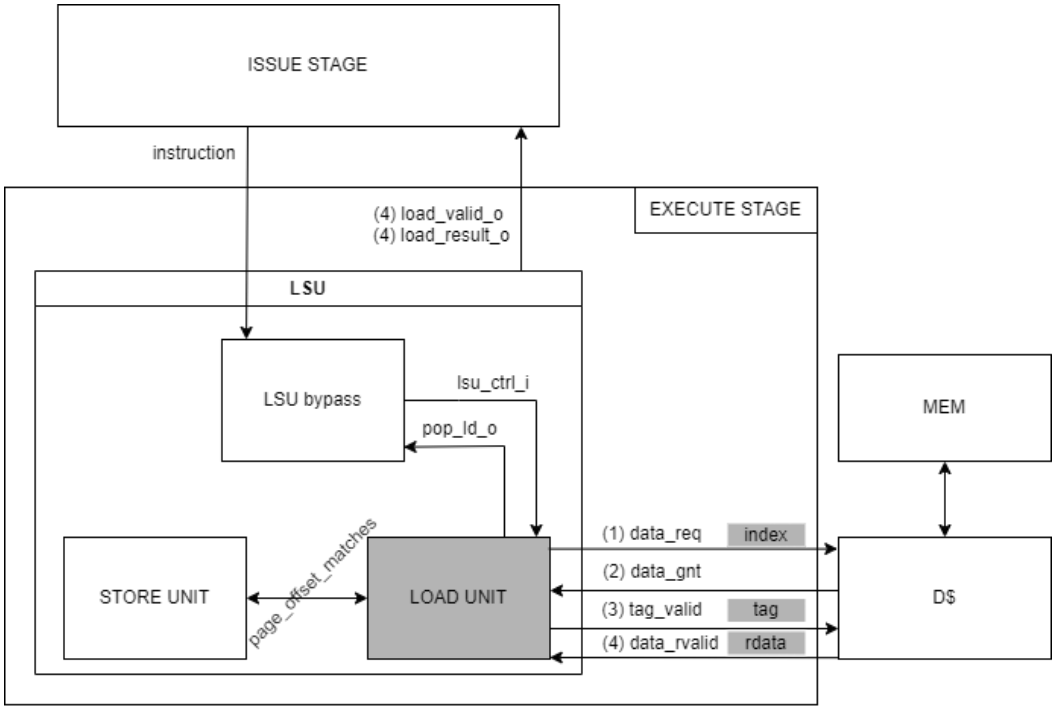


*Table 24. **load_unit module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic
`rst_ni`	in	Asynchronous reset active low	SUBSYSTEM	logic
`flush_i`	in	Flush signal	CONTROLLER	logic
`valid_i`	in	Load request is valid	LSU_BYPASS	logic
`lsu_ctrl_i`	in	Load request input	LSU_BYPASS	lsu_ctrl_t

Signal	IO	Description	connexion	Type
pop_ld_o	out	Pop the load request from the LSU bypass FIFO	LSU_BYPASS	logic
valid_o	out	Load unit result is valid	ISSUE_STAGE	logic
trans_id_o	out	Load transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
result_o	out	Load result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
ex_o	out	Load exception	ISSUE_STAGE	exception_t
translation_req_o	out	Request address translation	MMU	logic
vaddr_o	out	Virtual address	MMU	logic[CVA6Cfg.VLEN-1:0]
paddr_i	in	Physical address	MMU	logic[CVA6Cfg.PLEN-1:0]
ex_i	in	Excepted which appears before load	MMU	exception_t
page_offset_o	out	Page offset for address checking	STORE_UNIT	logic[11:0]
page_offset_matches_i	in	Indicates if the page offset matches a store unit entry	STORE_UNIT	logic
store_buffer_empty_i	in	Store buffer is empty	STORE_UNIT	logic
commit_tran_id_i	in	Transaction ID of the committing instruction	COMMIT_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
req_port_i	in	Data cache request out	CACHES	dcache_req_o_t
req_port_o	out	Data cache request in	CACHES	dcache_req_i_t
dcache_wbuffer_not_ni_i	in	Presence of non-idempotent operations in the D$ write buffer	CACHES	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVH = False,

- `tinst_o` output is tied to 0
- `hs_ld_st_inst_o` output is tied to 0
- `hlvx_inst_o` output is tied to 0

For any HW configuration,

- `dtlb_hit_i` input is tied to 1

As MMUPresent = 0,

- `dtlb_ppn_i` input is tied to 0

4.4.3.5.3. lsu_bypass

The LSU bypass is a FIFO which keeps instructions from the issue stage when the store unit or the load unit are not available immediately.

*Table 25. **lsu_bypass module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
flush_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
lsu_req_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	lsu_ctrl_t
lsu_req_valid_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
pop_ld_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
pop_st_i	in	TO_BE_COMPLETED	TO_BE_COMPLETED	logic
lsu_ctrl_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	lsu_ctrl_t
ready_o	out	TO_BE_COMPLETED	TO_BE_COMPLETED	logic

4.4.3.6. CVXIF_fu

TO BE COMPLETED

*Table 26. **cvxif_fu module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
x_valid_i	in	CVXIF instruction is valid	ISSUE_STAGE	logic
x_trans_id_i	in	Transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
x_illegal_i	in	Instruction is illegal, determined during CVXIF issue transaction	ISSUE_STAGE	logic
x_off_instr_i	in	Offloaded instruction	ISSUE_STAGE	logic[31:0]
x_ready_o	out	CVXIF is ready	ISSUE_STAGE	logic
x_trans_id_o	out	CVXIF result transaction ID	ISSUE_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
x_exception_o	out	CVXIF exception	ISSUE_STAGE	exception_t
x_result_o	out	CVXIF FU result	ISSUE_STAGE	logic[CVA6Cfg.XLEN-1:0]
x_valid_o	out	CVXIF result valid	ISSUE_STAGE	logic
x_we_o	out	CVXIF write enable	ISSUE_STAGE	logic
x_rd_o	out	CVXIF destination register	ISSUE_STAGE	logic[4:0]
result_valid_i	in	none	none	logic
result_i	in	none	none	x_result_t
result_ready_o	out	none	none	logic

4.5. COMMIT_STAGE Module

4.5.1. Description

The COMMIT_STAGE module implements the commit stage, which is the last stage in the processor's pipeline. For the instructions for which the execution is completed, it updates the architectural state: writing CSR registers, committing stores and writing back data to the register file. The commit stage controls the stalling and the flushing of the processor.

The commit stage also manages the exceptions. An exception can occur during the first four pipeline stages (PCgen cannot generate an exception) or happen in commit stage, coming from the CSR_REGFILE or from an interrupt. Exceptions are precise: they are considered during the commit only and associated with the related instruction.

The module is connected to:

- TO BE COMPLETED

*Table 27. **commit_stage module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
halt_i	in	Request to halt the core	CONTROLLER	logic
flush_dcache_i	in	request to flush dcache, also flush the pipeline	CACHE	logic
exception_o	out	TO_BE_COMPLETED	EX_STAGE	exception_t
commit_instr_i	in	The instruction we want to commit	ISSUE_STAGE	scoreboard_entry_t[CVA6Cfg.NrCommitPorts-1:0]
commit_drop_i	in	The instruction is cancelled	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
commit_ack_o	out	Acknowledge that we are indeed committing	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
commit_macro_ack_o	out	Acknowledge that we are indeed committing	CSR_REGFILE	logic[CVA6Cfg.NrCommitPorts-1:0]
waddr_o	out	Register file write address	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0][4:0]
wdata_o	out	Register file write data	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0] [CVA6Cfg.XLEN-1:0]
we_gpr_o	out	Register file write enable	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
we_fpr_o	out	Floating point register enable	ISSUE_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
pc_o	out	TO_BE_COMPLETED	FRONTEND_CSR_REGFILE	logic[CVA6Cfg.VLEN-1:0]
csr_op_o	out	Decoded CSR operation	CSR_REGFILE	fu_op
csr_wdata_o	out	Data to write to CSR	CSR_REGFILE	logic[CVA6Cfg.XLEN-1:0]
csr_rdata_i	in	Data to read from CSR	CSR_REGFILE	logic[CVA6Cfg.XLEN-1:0]
csr_exception_i	in	Exception or interrupt occurred in CSR stage (the same as commit)	CSR_REGFILE	exception_t
commit_lsu_o	out	Commit the pending store	EX_STAGE	logic
commit_lsu_ready_i	in	Commit buffer of LSU is ready	EX_STAGE	logic

Signal	IO	Description	connexion	Type
commit_tran_id_o	out	Transaction id of first commit port	ID_STAGE	logic[CVA6Cfg.TRANS_ID_BITS-1:0]
no_st_pending_i	in	no store is pending	EX_STAGE	logic
commit_csr_o	out	Commit the pending CSR instruction	EX_STAGE	logic
flush_commit_o	out	Request a pipeline flush	CONTROLLER	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVF = 0,

- dirty_fp_state_o output is tied to 0
- csr_write_fflags_o output is tied to 0

As DebugEn = False,

- single_step_i input is tied to 0

As RVA = False,

- amo_resp_i input is tied to 0
- amo_valid_commit_o output is tied to 0

As FenceEn = 0,

- fence_i_o output is tied to 0
- fence_o output is tied to 0

As RVS = False,

- sfence_vma_o output is tied to 0

As RVH = False,

- hfence_vvma_o output is tied to 0
- hfence_gvma_o output is tied to 0

4.5.2. Functionality

TO BE COMPLETED

4.6. CONTROLLER Module

4.6.1. Description

The CONTROLLER module implements ... TO BE COMPLETED

The module is connected to:

- TO BE COMPLETED

*Table 28. **controller module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
set_pc_commit_o	out	Set PC om PC Gen	FRONTEND	logic
flush_if_o	out	Flush the IF stage	FRONTEND	logic
flush_unissued_instr_o	out	Flush un-issued instructions of the scoreboard	FRONTEND	logic

Signal	IO	Description	connexion	Type
flush_id_o	out	Flush ID stage	ID_STAGE	logic
flush_ex_o	out	Flush EX stage	EX_STAGE	logic
flush_bp_o	out	Flush branch predictors	FRONTEND	logic
flush_icache_o	out	Flush ICache	CACHE	logic
flush_dcache_o	out	Flush DCache	CACHE	logic
flush_dcache_ack_i	in	Acknowledge the whole DCache Flush	CACHE	logic
halt_csr_i	in	Halt request from CSR (WFI instruction)	CSR_REGFILE	logic
halt_o	out	Halt signal to commit stage	COMMIT_STAGE	logic
eret_i	in	Return from exception	CSR_REGFILE	logic
ex_valid_i	in	We got an exception, flush the pipeline	FRONTEND	logic
resolved_branch_i	in	We got a resolved branch, check if we need to flush the front-end	EX_STAGE	bp_resolve_t
flush_csr_i	in	We got an instruction which altered the CSR, flush the pipeline	CSR_REGFILE	logic
flush_commit_i	in	Flush request from commit stage	COMMIT_STAGE	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVH = False,

- `v_i` input is tied to 0
- `flush_tlb_vvma_o` output is tied to 0
- `flush_tlb_gvma_o` output is tied to 0
- `hfence_vvma_i` input is tied to 0
- `hfence_gvma_i` input is tied to 0

As MMUPresent = 0,

- `flush_tlb_o` output is tied to 0

As EnableAccelerator = 0,

- `halt_acc_i` input is tied to 0
- `flush_acc_i` input is tied to 0

As DebugEn = False,

- `set_debug_pc_i` input is tied to 0

As FenceEn = 0,

- `fence_i_i` input is tied to 0
- `fence_i` input is tied to 0

As RVS = False,

- `sfence_vma_i` input is tied to 0

4.6.2. Functionality

TO BE COMPLETED

4.7. CSR_REGFILE Module

4.7.1. Description

The CSR_REGFILE module implements ... TO BE COMPLETED

The module is connected to:

- TO BE COMPLETED

*Table 29. **csr_regfile module** IO ports*

Signal	IO	Description	connexion	Type
clk_i	in	Subsystem Clock	SUBSYSTEM	logic
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
time_irq_i	in	Timer threw a interrupt	SUBSYSTEM	logic
flush_o	out	send a flush request out when a CSR with a side effect changes	CONTROLLER	logic
halt_csr_o	out	halt requested	CONTROLLER	logic
commit_instr_i	in	Instruction to be committed	ID_STAGE	scoreboard_entry_t[CVA6Cfg.NrCommitPorts-1:0]
commit_ack_i	in	Commit acknowledged a instruction → increase instret CSR	COMMIT_STAGE	logic[CVA6Cfg.NrCommitPorts-1:0]
boot_addr_i	in	Address from which to start booting, mtvec is set to the same address	SUBSYSTEM	logic[CVA6Cfg.VLEN-1:0]
hart_id_i	in	Hart id in a multicore environment (reflected in a CSR)	SUBSYSTEM	logic[CVA6Cfg.XLEN-1:0]
ex_i	in	We've got an exception from the commit stage, take it	COMMIT_STAGE	exception_t
csr_op_i	in	Operation to perform on the CSR file	COMMIT_STAGE	fu_op
csr_addr_i	in	Address of the register to read/write	EX_STAGE	logic[11:0]
csr_wdata_i	in	Write data in	COMMIT_STAGE	logic[CVA6Cfg.XLEN-1:0]
csr_rdata_o	out	Read data out	COMMIT_STAGE	logic[CVA6Cfg.XLEN-1:0]
pc_i	in	PC of instruction accessing the CSR	COMMIT_STAGE	logic[CVA6Cfg.VLEN-1:0]
csr_exception_o	out	attempts to access a CSR without appropriate privilege	COMMIT_STAGE	exception_t

Signal	IO	Description	connexion	Type
epc_o	out	Output the exception PC to PC Gen, the correct CSR (mepc, sepc) is set accordingly	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
eret_o	out	Return from exception, set the PC of epc_o	FRONTEND	logic
trap_vector_base_o	out	Output base of exception vector, correct CSR is output (mtvec, stvec)	FRONTEND	logic[CVA6Cfg.VLEN-1:0]
irq_ctrl_o	out	interrupt management to id stage	ID_STAGE	irq_ctrl_t
irq_i	in	external interrupt in	SUBSYSTEM	logic[1:0]
ipi_i	in	inter processor interrupt → connected to machine mode sw	SUBSYSTEM	logic
icache_en_o	out	L1 ICache Enable	CACHE	logic
dcache_en_o	out	L1 DCache Enable	CACHE	logic
rvfi_csr_o	out	none	none	rvfi_probes_csr_t

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As RVF = 0,

- `dirty_fp_state_i` input is tied to 0
- `csr_write_fflags_i` input is tied to 0
- `fs_o` output is tied to 0
- `fflags_o` output is tied to 0
- `frm_o` output is tied to 0
- `fprec_o` output is tied to 0

As EnableAccelerator = 0,

- `dirty_v_state_i` input is tied to 0
- `acc_fflags_ex_i` input is tied to 0
- `acc_fflags_ex_valid_i` input is tied to 0
- `acc_cons_en_o` output is tied to 0
- `pmpcfg_o` output is tied to 0
- `pmpaddr_o` output is tied to 0

As PRIV = MachineOnly,

- `priv_lvl_o` output is tied to MachineMode
- `ld_st_priv_lvl_o` output is tied to MAchineMode
- `tvm_o` output is tied to 0
- `tw_o` output is tied to 0
- `tsr_o` output is tied to 0

As RVH = False,

- `v_o` output is tied to 0
- `vfs_o` output is tied to 0

- `en_g_translation_o` output is tied to 0
- `en_ld_st_g_translation_o` output is tied to 0
- `ld_st_v_o` output is tied to 0
- `csr_hs_ld_st_inst_i` input is tied to 0
- `vs_sum_o` output is tied to 0
- `vmxr_o` output is tied to 0
- `vsatp_ppn_o` output is tied to 0
- `vs_asid_o` output is tied to 0
- `hgatp_ppn_o` output is tied to 0
- `vmid_o` output is tied to 0
- `vtw_o` output is tied to 0
- `hu_o` output is tied to 0

As RVV = False,

- `vs_o` output is tied to 0

As RVS = False,

- `en_translation_o` output is tied to 0
- `en_ld_st_translation_o` output is tied to 0
- `sum_o` output is tied to 0
- `mxr_o` output is tied to 0
- `satp_ppn_o` output is tied to 0
- `asid_o` output is tied to 0

As DebugEn = False,

- `debug_req_i` input is tied to 0
- `set_debug_pc_o` output is tied to 0
- `debug_mode_o` output is tied to 0
- `single_step_o` output is tied to 0

As PerfCounterEn = 0,

- `perf_addr_o` output is tied to 0
- `perf_data_o` output is tied to 0
- `perf_data_i` input is tied to 0
- `perf_we_o` output is tied to 0
- `mcountinhibit_o` output is tied to 0

4.7.2. Functionality

TO BE COMPLETED

4.8. CACHES Module

4.8.1. Description

The CACHES module implements an instruction cache, a data cache and an AXI adapter.

The module is connected to:

- TO_BE_COMPLETED

*Table 30. **cva6_hpdcache_subsystem module** IO ports*

Signal	IO	Description	connexion	Type
`clk_i`	in	Subsystem Clock	SUBSYSTEM	logic

Signal	IO	Description	connexion	Type
rst_ni	in	Asynchronous reset active low	SUBSYSTEM	logic
noc_req_o	out	noc request, can be AXI or OpenPiton	SUBSYSTEM	noc_req_t
noc_resp_i	in	noc response, can be AXI or OpenPiton	SUBSYSTEM	noc_resp_t
icache_en_i	in	Instruction cache enable	CSR_REGFILE	logic
icache_flush_i	in	Flush the instruction cache	CONTROLLER	logic
icache_areq_i	in	Input address translation request	EX_STAGE	icache_areq_t
icache_areq_o	out	Output address translation request	EX_STAGE	icache_arsp_t
icache_dreq_i	in	Input data translation request	FRONTEND	icache_dreq_t
icache_dreq_o	out	Output data translation request	FRONTEND	icache_drsp_t
dcache_enable_i	in	Data cache enable	CSR_REGFILE	logic
dcache_flush_i	in	Data cache flush	CONTROLLER	logic
dcache_flush_ack_o	out	Flush acknowledge	CONTROLLER	logic
dcache_amo_req_i	in	AMO request	EX_STAGE	ariane_pkg::amo_req_t
dcache_amo_resp_o	out	AMO response	EX_STAGE	ariane_pkg::amo_resp_t
dcache_req_ports_i	in	Data cache input request ports	EX_STAGE	dcache_req_i_t[NumPorts-1:0]
dcache_req_ports_o	out	Data cache output request ports	EX_STAGE	dcache_req_o_t[NumPorts-1:0]
wbuffer_empty_o	out	Write buffer status to know if empty	EX_STAGE	logic
wbuffer_not_ni_o	out	Write buffer status to know if not non idempotent	EX_STAGE	logic

Due to cv32a65x configuration, some ports are tied to a static value. These ports do not appear in the above table, they are listed below

As PerfCounterEn = 0,

- `icache_miss_o` output is tied to 0
- `dcache_miss_o` output is tied to 0

For any HW configuration,

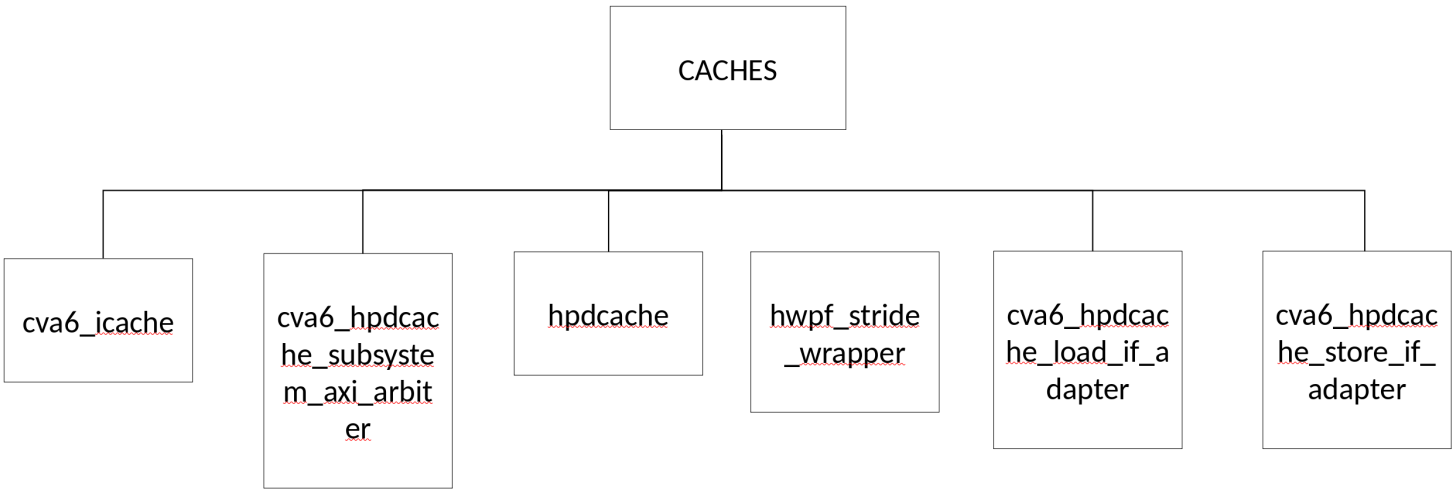
- `dcache_cmo_req_i` input is tied to 0
- `dcache_cmo_resp_o` output is tied to open
- `hwpf_base_set_i` input is tied to 0
- `hwpf_base_i` input is tied to 0
- `hwpf_base_o` output is tied to 0
- `hwpf_param_set_i` input is tied to 0
- `hwpf_param_i` input is tied to 0

- `hwpf_param_o` output is tied to 0
- `hwpf_throttle_set_i` input is tied to 0
- `hwpf_throttle_i` input is tied to 0
- `hwpf_throttle_o` output is tied to 0
- `hwpf_status_o` output is tied to 0

4.8.2. Functionality

TO BE COMPLETED

4.8.3. Submodules



5. Glossary

- **ALU**: Arithmetic/Logic Unit
- **APU**: Application Processing Unit
- **ASIC**: Application-Specific Integrated Circuit
- **AXI**: Advanced eXtensible Interface
- **BHT**: Branch History Table
- **BTB**: Branch Target Buffer
- **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)
- **CVA6**: Core-V Application class processor with a 6 stage pipeline
- **D$**: Data Cache
- **DPI**: Direct Programming Interface
- **EX** or **EXE**: Instruction Execute
- **FPGA**: Field Programmable Gate Array
- **FPU**: Floating Point Unit
- **Halfword**: 16-bit data item
- **Halfword aligned address**: An address is halfword aligned if it is divisible by 2
- **I$**: Instruction Cache
- **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)
- **MMU**: Memory Management Unit
- **NC**: Not Cacheable
- **OBI**: Open Bus Interface
- **OoO**: Out Of Order
- **PC**: Program Counter
- **PMP**: Physical memory protection (RISC-V Instruction Set Manual, Volume II: Privileged Architecture)
- **PTW**: Page Table Walker
- **PULP platform**: Parallel Ultra Low Power Platform (https://pulp-platform.org)
- **RAS**: Return Address Stack
- **RV32C**: RISC-V Compressed (C extension)
- **RV32F**: RISC-V Floating Point (F extension)
- **S-Mode**: Supervisor Mode (RISC-V Instruction Set Manual, Volume II: Privileged Architecture)
- **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)
- **TLB**: Translation Lookaside Buffer
- **U-Mode**: User Mode (RISC-V Instruction Set Manual, Volume II: Privileged Architecture)
- **VLEN**: Virtual address length

- **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
- **XLEN**: RISC-V processor data length

Last updated 2024-07-26 16:50:22 +0200