YRV

Technical Manual

Disclaimer

Author reserves the right to make changes at any time, without notice, to improve design or performance and provide the best IP possible. Author makes no warrant for the use of this IP and assumes no responsibility for any errors which may appear in this document nor does he make any commitment to update the information contained herein.

Author's IP is not authorized for use in life support devices or systems under any circumstances. Nothing contained herein shall be construed as a recommendation to use IP in violation of existing patents, copyrights or other rights of third parties. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Author.

Every effort has been made to ensure the accuracy of the information contain herein. If you find errors or inconsistencies please bring them to our attention. In all cases, however, the Verilog HDL source code for the YRV core defines “proper operation”.

© 2023, Monte Dalrymple. All rights reserved.

Acknowledgements

The RISC-V Instruction Set Architecture was originally developed at UC Berkeley. More information about the RISC-V ecosystem can be found at the RISC-V International website: www.riscv.org.

The “RISC-V” trade name is a registered trade mark of RISC-V International.

Table of Contents

1. Introduction 1
2. Features 3
3. External Interface 5
    - Clock and Reset 5
    - Configuration Control 5
    - Memory Interface 6
    - Interrupts 7
    - External CSR Interface 8
    - Miscellaneous 9
    - Debug Support 9
4. Timing 11
    - Instruction Fetch 13
    - Instruction Fetch Discontinuity 14
    - Load from Memory 15
    - Store to Memory 16
    - Atomic Memory Operation 17
    - CSR Access 18
    - Wait for Interrupt 19
    - Break 20
    - Reset 21
5. Instruction Set (32-bit opcode) 24
6. Pseudo-instructions 110
7. Instruction Set (16-bit opcode) 112
8. Control/Status Registers 151
    - Machine Status 153
    - Machine ISA 154
    - Machine Interrupt Enable 155
    - Machine Trap-Vector Base Address 156
    - Machine Counter-Inhibit 157
    - Machine Scratch 158
    - Machine Exception PC 159
    - Machine Trap Cause 160
    - Machine Trap Value 162
    - Machine Interrupt Pending 163
    - Clock-cycle Counter 165
    - Timer 166
    - Instructions-retired Counter 167
    - Vendor ID 168
    - Architecture ID 169

Implementation ID 170
Hardware Thread ID 171
Debug Control/Status 172
Debug Exception PC 174
Debug Scratch 0 and 1 175
7. Logic Synthesis Options 176
Read-Only CSR Values 176
CSR Reset Values 177
Default Addresses 178
Debug Operation 179
Hardware Features 180
Implementation-specific 181
8. YRV Microcontroller 182

Revison History

Date	Changes	Page(s)
06/05/2021	Initial release	
03/03/2023	Initial github release	

Introduction

The YRV core implements the base 32-bit RISC-V Instruction Set Architecture (ISA) along with several standard extensions. Options are available at the time of logic synthesis to tailor the core to meet specific requirements, including options to include or exclude certain architectural features.

The YRV CPU operates with a one-clock-cycle machine cycle using a six stage pipeline, and uses a subset of the AMBA AHB-Lite bus interface, with either a 32-bit or 16-bit data bus.

This document should always be used as the final word on the operation of the YRV core, but it is useful to refer to the RISC-V documentation if the description given here is too cryptic.

Features

---

* Verilog HDL implementation of the 32-bit RISC-V Instruction Set Architecture

* **RV32I** Base Integer Instruction Set

* **Zca** subset of **RV32C** Standard Extension for Compressed Instructions

* **Zcb** Code Size Reduction Instructions
  (as a logic synthesis option; excludes *c.mul*)

* **RV32A** Atomic Memory Operation Instructions
  (only *amoadd.w*, *aamoand.w*, *amoor.w*, *amoswap.w* and *amoxor.w*)

* **Zam** Standard Extension for Misaligned Atomics

* **Zbb** Basic Bit-manipulation Instructions
  (excluding *clz*, *cpop*, *ctz*, *max*, *maxu*, *min*, *minu* and *orc.b*)

* **Zbkb** Bit-manip Instructions for Cryptography
  (excluding *zip* and *unzip*)

* **Zbs** Single-bit Instructions

* **Zicntr** Standard Extension for Base Counters and Timers
  (as a logic synthesis option: 64-bit, 32-bit or none for each counter and timer)

* **Zicsr** Control and Status Register (CSR) Instructions
  (including an external hardware interface for custom CSR registers)

* **Zifencei** Instruction-Fetch Fence Instruction

* **Zkt** Data Independent Execution Latency

* **Sdext** ISA extension for debug features

* One-clock machine cycle, six-stage pipeline

* 16-bit or 32-bit data bus width
  (as a logic synthesis option)

* AMBA AHB-Lite protocol memory interface
  (no burst transfers, no error response)

* Internal Register File uses FPGA or ASIC memory macros
  (Lattice ICE40 version uses four memory macros;
  Xilinx Virtex-7 version uses two memory macros)

* One external interrupt request, sixteen *local* (non-standard) interrupt requests

External Interface

The YRV design is suitable for use in either an FPGA or an ASIC. The interface signals for the YRV core are detailed below.

Clock and Reset

This core uses a single clock and reset. No clock gating is employed in the design.

`clk` (input, active-High) The rising edge of the Master Clock samples all inputs except for the asynchronous `resetb`. All outputs normally change in response to the rising edge of the Master Clock.

`resetb` (input, active-Low) The Master Reset signal is used to initialize all state flip-flops and registers, consistent with RISC-V specification. This is an asynchronous signal, but the trailing edge should be synchronized with the rising edge of the Master Clock. There is a 128-clock delay after the release of `resetb` before instruction fetches begin.

Configuration Control

The core includes two configuration options that are not always suitable for definition at the time of logic synthesis, because multiple cores with different capabilities might undergo logic synthesis simultaneously.

`bus_32` (input, active-High) The 32-bit Bus Select signal selects between a 32-bit data bus (High) and a 16-bit data bus (Low). The bus width selection is sampled during reset, just before the processor begins fetching instructions.

`hw_id` (inputs, 10-bit bus) The Hardware Identification bus carries a 10-bit identifier that is normally hardwired with a static value. This value is available to software via the **mhartid** CSR register. This identifier should be 0x000 for a single core, and one core must be identified with 0x000 in the case of multiple cores in a system.

Memory Interface

The memory interface uses a subset of the AHB-Lite signals and protocol. This core does not generate burst transfers, but transfer sizes are optimized to use the minimum number of bus cycles. The memory interface does not accommodate an error response from the slave, but does handle wait-stated transfers.

`mem_addr` (outputs, 32-bit bus) The Memory Address bus carries the address for all memory transactions. Instruction fetches will always be half-word aligned, with `mem_addr[0]` Low.

`mem_ble` (outputs, 4-bit bus) The Memory Byte Lane Enable bus signals the valid bytes on the write data bus and the sampled bytes on the read data bus. Only the bit combinations shown in the table below will ever be generated.

`mem_ble`	Data width	Valid bits (32-bit bus)	Valid bits (16-bit bus)
0001	byte	[7:0]	[7:0]
0010	byte	[15:8]	[15:8]
0100	byte	[23:16]	n/a
1000	byte	[31:24]	n/a
0011	halfword	[15:0]	[15:0]
0110	halfword	[23:8]	n/a
1100	halfword	[31:16]	n/a
0111	3 bytes	[23:0]	n/a
1110	3 bytes	[31:8]	n/a
1111	word	[31:0]	n/a

`mem_lock` (output, active-High) The Memory Lock signal indicates that the bus transfer is part of an atomic read-modify-write operation, and is valid coincident with the `mem_addr` bus. This signal will only be generated for RV32A Atomic Memory Operation instructions.

`mem_rdata` (inputs, 32-bit bus) The Memory Read Data bus is sampled during the data phase of memory reads. When the 16-bit bus option is selected only the lower half of the `mem_rdata` bus is used and the upper half of the bus should be tied all Low.

`mem_ready` (input, active-High) The Memory Ready signal indicates that the target has either accepted write data or is providing read data during the data phase of a bus transfer.

`mem_trans` (outputs, 2-bit bus) The Memory Transaction Type bus indicates the type of operation on the memory bus and is valid coincident with the `mem_addr` bus.

`mem_trans`	Meaning	Use for
00	Idle	Start-up, WFI, Internal Op
01	Fetch	Instruction fetch
11	Data	Load, Store, AMO

`mem_wdata` (outputs, 32-bit bus) The Memory Write Data bus carries the output data for write transactions and is valid during the data phase of a transfer. To conserve power, this bus only changes state as required for a memory write. When the 16-bit bus option is selected only the lower half of the `mem_wdata` bus is used, and the upper half of the `mem_wdata` bus should be left unconnected.

`mem_write` (output, active-High) The Memory Write signal indicates that the bus transfer is for a write, and is valid coincident with the `mem_addr` bus.

Interrupts

The YRV core supports the four interrupt requests specified for RISC-V: external, non-maskable, software and timer. In addition, sixteen custom *local* interrupts are implemented. In the case of simultaneous interrupt requests, the non-maskable interrupt has the highest priority, followed by the local interrupts and then the external interrupt. The sixteen local interrupts are prioritized with the lower number having higher priority.

`ei_req` (input, active-High) The External Interrupt Request is level-sensitive. An External interrupt is only accepted while interrupts are enabled, or while the processor is in the Wait-For-Interrupt state.

`li_req` (16 inputs, active-High) The Local Interrupt Requests are level-sensitive, and are only accepted while interrupts are enabled, or while the processor is in the Wait-For-Interrupt state. These interrupts are not standardized, and the corresponding interrupt status occupies areas in CSR registers reserved for platform or custom use.

`nmi_req` (input, active-High) The Non-maskable Interrupt (NMI) Request signal is rising-edge-triggered. A Non-maskable interrupt is accepted at any time during normal operation, but is automatically masked during debugging. NMI is normally not recoverable, although there is currently a RISC-V extension proposal that may change this.

`sw_req` (input, active-High) The Software Interrupt Request is level-sensitive, and is only accepted while interrupts are enabled, or while the processor is in the Wait-For-Interrupt state. This interrupt is normally used for inter-processor communication, but is otherwise general-purpose.

`timer_match` (input, active-High) The Timer Interrupt Request is level-sensitive, and is only accepted while interrupts are enabled, or while the processor is in the Wait-For-Interrupt state. This interrupt is normally used to signal a timer match condition (the standard RISC-V timer is external to the core).

External CSR Interface

The external CSR interface is similar the the APB signals and protocol. Because invalid CSR addresses must generate an exception, the external CSR interface includes an address-check bus and an address-valid signal that must be asserted for every valid external CSR address. Internal CSR addresses generate their own internal address-valid signal.

All CSR accesses are visible on this bus, but if the access is for an internal CSR address the internal register will supply the CPU with read data and latch write data.

`csr_achk` (outputs, 12-bit bus) The CSR Address Check bus carries the address for an upcoming CSR transaction. Combinational logic must asset the `csr_ok_ext` signal whenever this bus contains a valid external CSR address.

`csr_addr` (outputs, 12-bit bus) The CSR Address bus carries the address for CSR transactions. This bus is driven with all zeros whenever no CSR transaction is in progress.

`csr_ok_ext` (input, active-High) The CSR External Address Valid signal indicates that the CSR address on the `csr_achk` bus is a valid external CSR address. This signal must be generated using combinational logic so that it is valid during the same clock cycle as the `csr_achk` address.

`csr_read` (output, active-High) The CSR Read signal indicates that the `csr_rdata` bus will be sampled on the next rising edge of the clk signal.

`csr_rdata` (inputs, 32-bit bus) The CSR Read Data bus is sampled while the `csr_read` signal is High. All 32 bits must be driven with valid data for an external CSR access. This bus is ignored in the case of a valid internal CSR address.

`csr_wdata` (outputs, 32-bit bus) The CSR Write Data bus carries the output data for write transactions and is valid while the `csr_write` signal is active. This bus is only driven with a value during CSR write transactions, and is driven with all zeros at all other times.

`csr_write` (output, active-High) The CSR Write signal indicates that the data on the `csr_wdata` bus is valid.

Miscellaneous

`timer_en` (output, active-High) The Timer Enable output is normally active, because the Timer is assumed to run continuously, but can be disabled by a bit while in Debug mode by a bit in the **dcsr** CSR.

`timer_rdata` (inputs, 64-bit bus) The Timer Read Data bus is available in the **time** and **timeh** CSRs. No synchronization is performed by the core, so this must be guaranteed externally.

`wfi_state` (output, active-High) The Wait-For-Interrupt State signal is set High by the WFI instruction, and remains High until an interrupt is accepted.

Debug Support

The YRV core does not include a Debug Module, but it does include the basic features necessary to support an attached RISC-V standard Debug Module. The standard Debug Module will include the state machine required by an external debugger.

`brk_req` (input, active-High) The Breakpoint Request signal causes the core to enter Debug mode. This signal is typically generated by an address or data trigger. For a breakpoint triggered by an instruction fetch, this signal should be asserted during the data phase of the instruction fetch. Debug mode will then be entered prior to the execution of this instruction. For a breakpoint triggered by a load, store or CSR access, this signal should be asserted during the data phase of the load or store. Debug mode will then be entered prior to the execution of the instruction following the load, store or CSR instruction.

`dbg_req` (input, active-High) The Debug Request signal causes the core to immediately enter Debug mode. This signal is typically generated by the debugger itself. The timing of this request is identical to that of a non-maskable interrupt.

`debug_mode` (output, active-High) The Debug Mode signal is High while the core is in Debug mode, and remains High until cleared by the **DRET** instruction or a reset of the core.

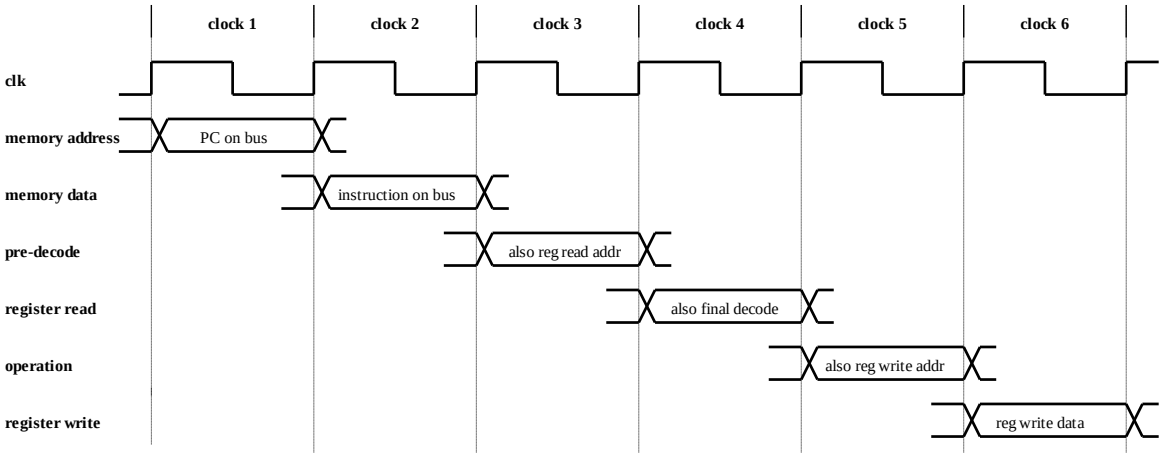
`dresetb` (input, active-Low) The Debug Reset signal is used to initialize certain debug features that are unaffected by the Global Reset. This is an asynchronous signal, but the trailing edge should be synchronized with the rising edge of the Master Clock, and coincident with the `resetb` signal.

`ebrk_inst` (output, active-High) The EBREAK Instruction signal is High for one clock cycle when an EBREAK instruction is executed. This one-clock-cycle pulse if coincident with the final clock cycl;e of the EBREAK instruction itself. This information is required for the RISC-V standard debugger when running code from the program buffer.

`halt_reg` (input, active-High) The Halt Request signal causes the core to immediately enter Debug mode. This signal is typically generated by the debugger itself when entry into Debug mode immediately after reset is required.

Timing

This core uses a uniform one-clock-cycle machine cycle, and this is the minimum apparent execution time for most instructions. Instructions actually require at least six clock cycles to complete, flowing through the pipeline stages as shown below.



During the first clock cycle the address of the instruction to be fetched is driven on the `mem_addr` bus.

During the second clock cycle the memory returns the instruction on the `mem_rdata` bus. If the memory system requires more time to retrieve this data the `mem_ready` signal can be used to halt the pipeline until the data is ready.

During the third clock cycle the instruction undergoes a pre-decode operation. This is where a compressed opcode is expanded into a regular word-wide opcode and common fields, such register addresses and immediate data, are extracted. The register source addresses are presented to the internal register file during this cycle and are latched by the register file at the end of this cycle. This register file timing is consistent with the timing of RAM macros common in both ASIC and FPGA technologies.

The fourth clock cycle is where the source register data is accessed from the register file. This is also where the final instruction decoding is done. Illegal instructions are resolved during this clock cycle.

During the fifth clock cycle the ALU is used to operate on the operands specified

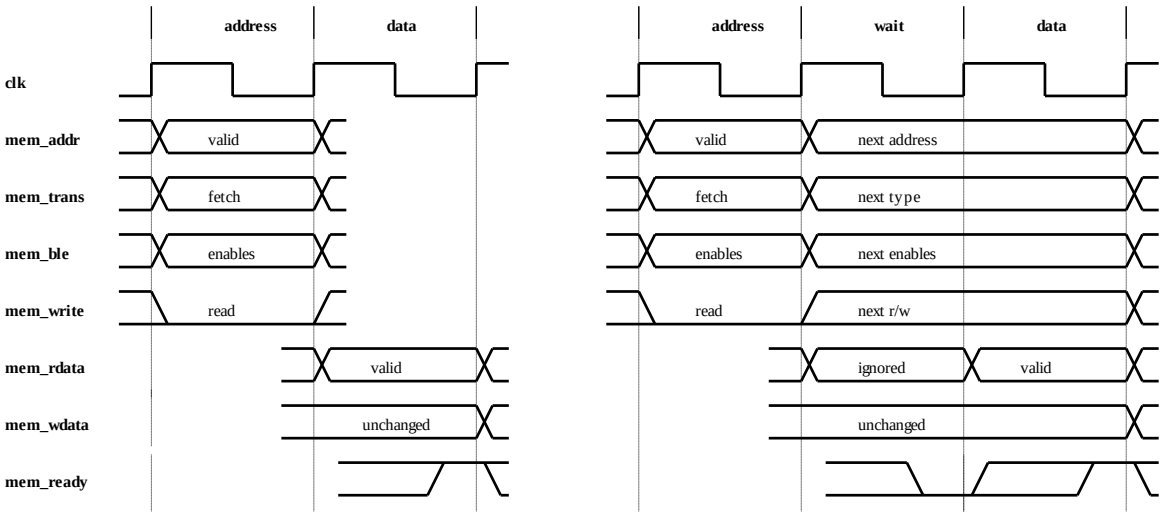
in the opcode. This is also where branch instructions and traps are recognized. Load, store, and CSR operation instructions are recognized during this cycle, and are executed at the end of this cycle, before continuing on the the sixth and final cycle.

The sixth and final clock cycle is where the results of the instruction are written back to the register file.

Instruction Fetch

The memory bus is pipelined, with the address driven during one clock cycle and the data returned during the subsequent clock cycle. Wait states pause the entire CPU pipeline, not just the external bus.

During an instruction fetch the `mem_size` bus will always signal word on a 32-bit bus, and half-word on a 16-bit bus.

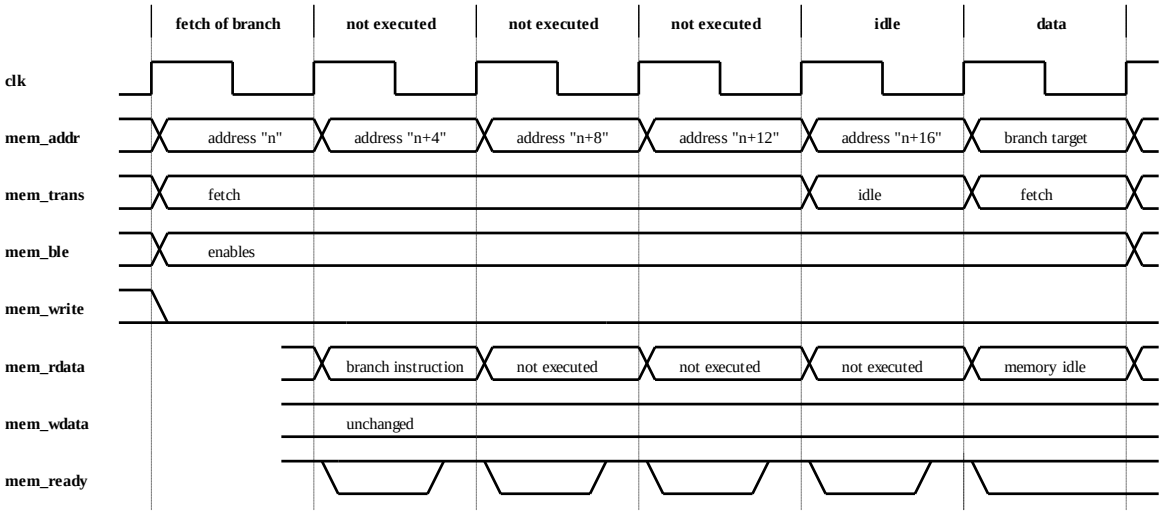


Instruction Fetch Discontinuity

The instruction execution pipeline will become apparent during non-sequential instruction fetches. The timing shown below is common to all instruction fetch discontinuities, whether they occur because of instruction execution or a trap. This diagram assumes that there are no loads, stores, or CSR accesses immediately prior to the branch instruction or trap condition.

The apparent execution time is only five clock cycles because the next fetch starts at the end of the ALU operation pipeline stage.

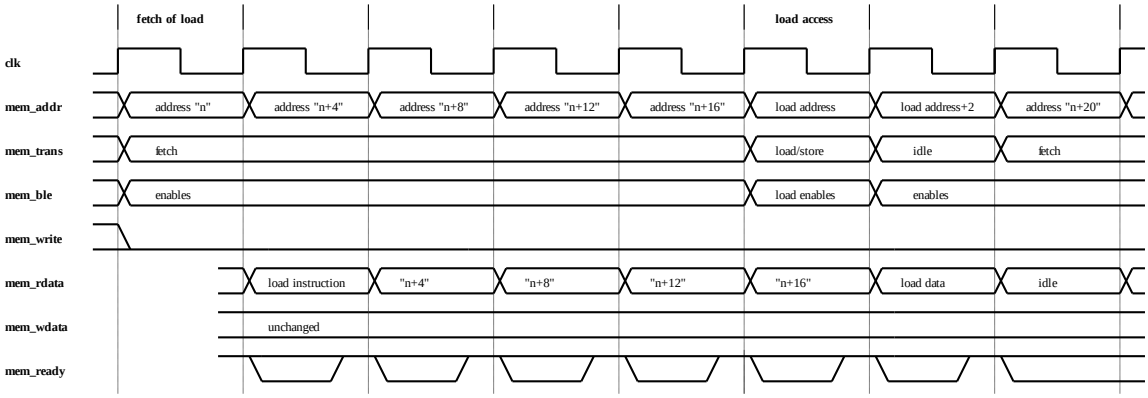
The data phase of the Idle cycle immediately prior to the fetch cycle at the branch target address does not sample the `mem_ready` signal, and will always be one clock cycle in length.



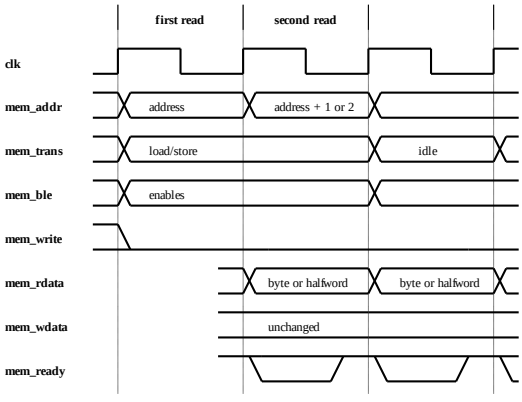
Load from memory

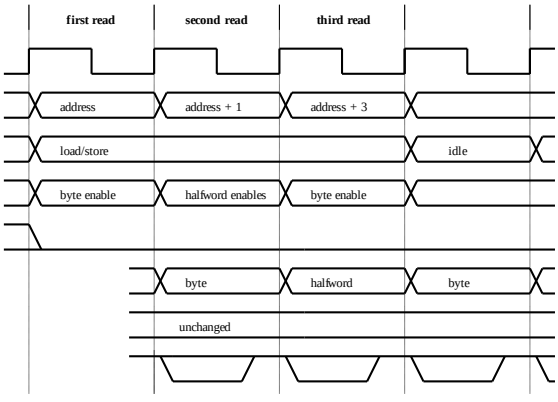
Memory reads for a Load instruction are inserted between the fifth and sixth clock cycles of the normal pipeline operation. The core supports both aligned and unaligned load addresses, on both a 16-bit bus and a 32-bit bus.

The figure below shows the timing of an aligned access relative to the fetch of the load instruction. Loads are always performed in program execution order. The idle cycle on the bus is required to allow the correct register data to be available for all instructions subsequent to the load instruction.



The figures below show the timing of multiple accesses. Load memory cycles are always performed least-significant byte/halfword first. Two reads are required for an unaligned halfword access, or a halfword-aligned word access. Three reads are required for any unaligned word access, independent of bus width.

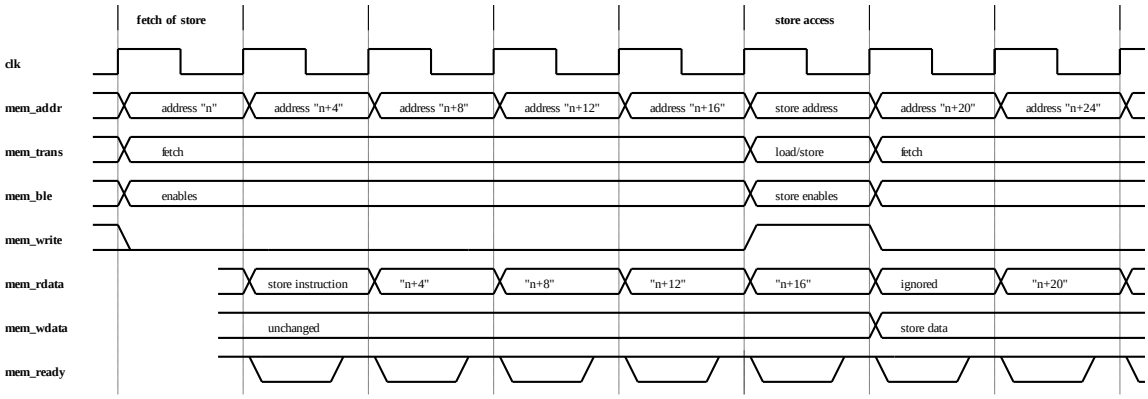




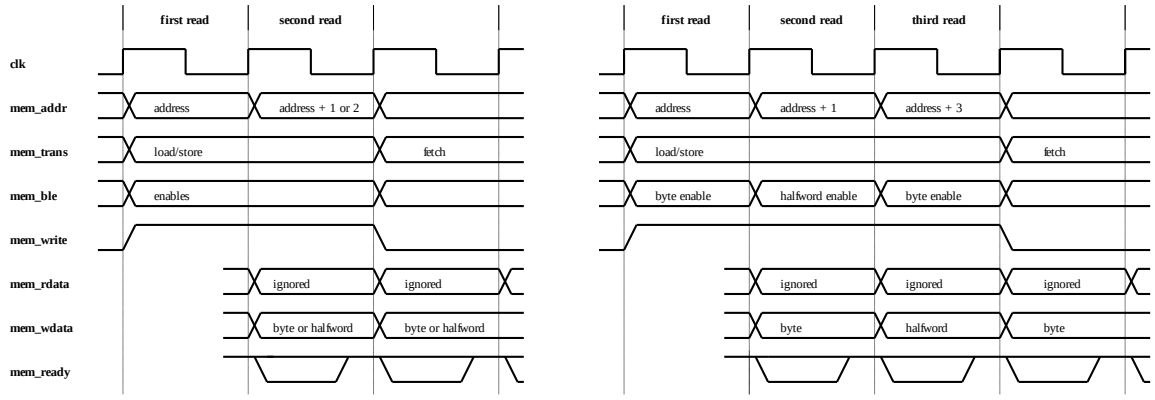
Store to memory

Memory writes for a Store instruction are inserted between the fifth and sixth clock cycles of the normal pipeline operation. The core supports both aligned and unaligned store addresses, on both a 16-bit bus and a 32-bit bus.

The figure below shows the timing of an aligned access relative to the fetch of the store instruction. Stores are always performed in program execution order.



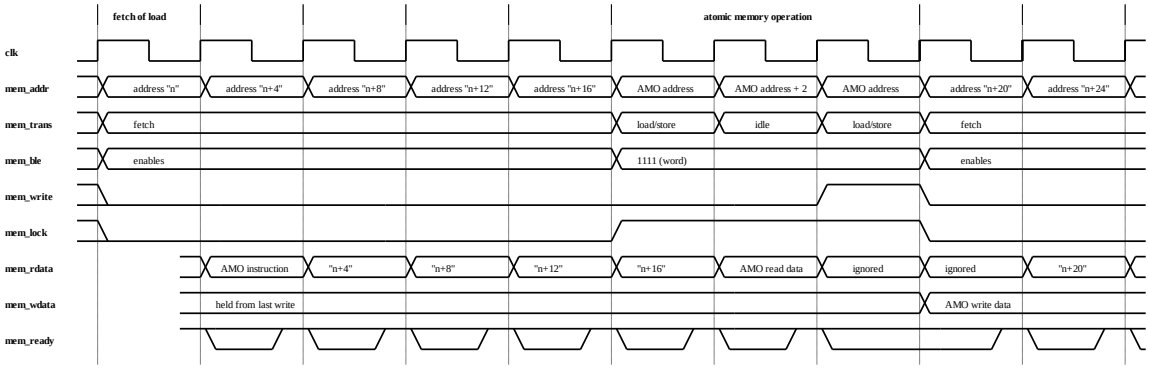
The figure below shows the timing of multiple accesses. Store memory cycles are always performed least-significant byte/halfword first. Two writes are required for an unaligned halfword access, or a halfword-aligned word access. Three writes are required for any unaligned word access, independent of bus width.



Atomic Memory Operation

Atomic memory operations are inserted between the fifth and sixth clock cycles of the normal pipeline operation. Atomic memory operations do not need to be word-aligned in this core.

The figure below shows the timing of an aligned atomic memory operation relative to the fetch of the AMO instruction. Atomic memory operations are always performed in program execution order. The idle cycle between the read and the write is required to perform the operation on the memory data.

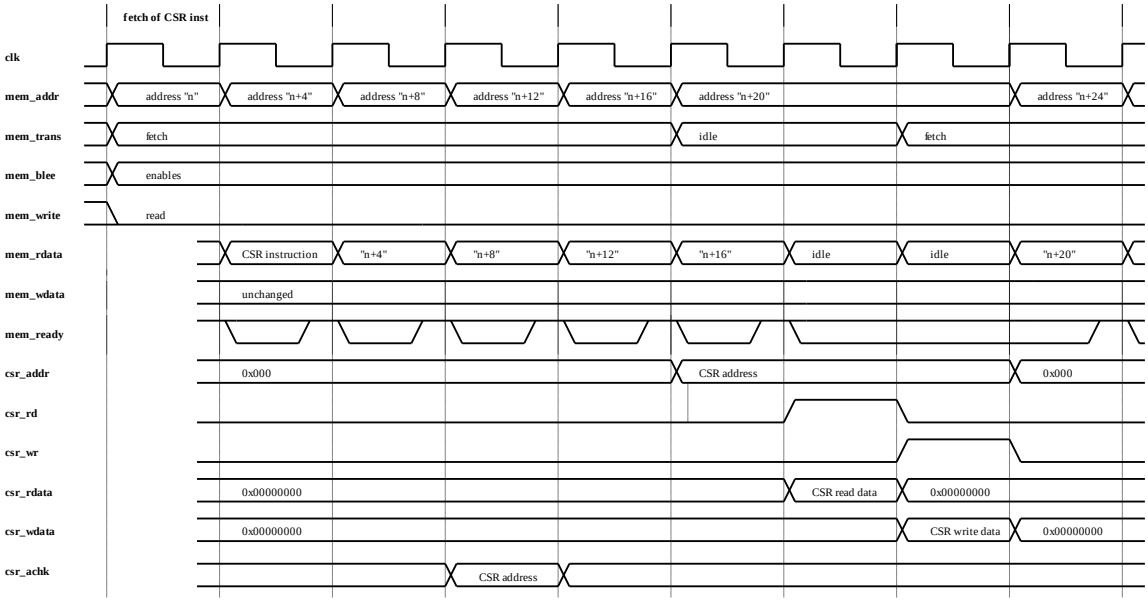


The timing of the atomic memory operation on a 16-bit bus is identical, except that two memory reads and two memory writes are required. This is also the case for an unaligned operation on a 32-bit bus. An unaligned operation on a 16-bit bus will result in three memory reads and three memory writes.

CSR access

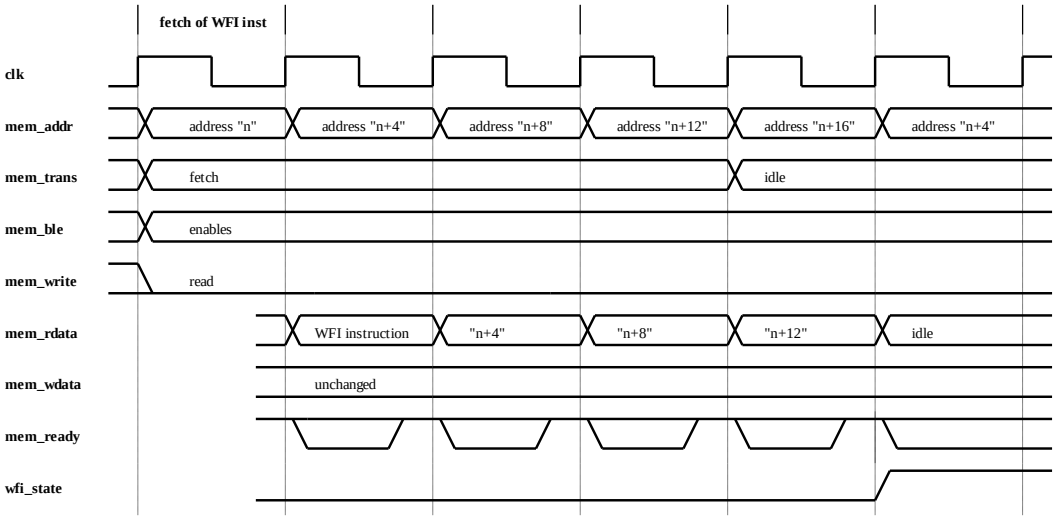
CSR accesses are inserted between the fifth and sixth clock cycles of the normal pipeline operation, but there is no explicit indication on the external bus that a CSR access is in progress.

The CSR bus is internal to the design, connecting the CPU module to the CSR module, so if additional control or status registers are required the timing of this bus will be needed. The figure below shows the timing of the CSR bus. This bus is synchronous, is not pipelined, and does not accommodate Wait states. Either the read strobe or the write strobe may be missing, depending on the CSR instruction.

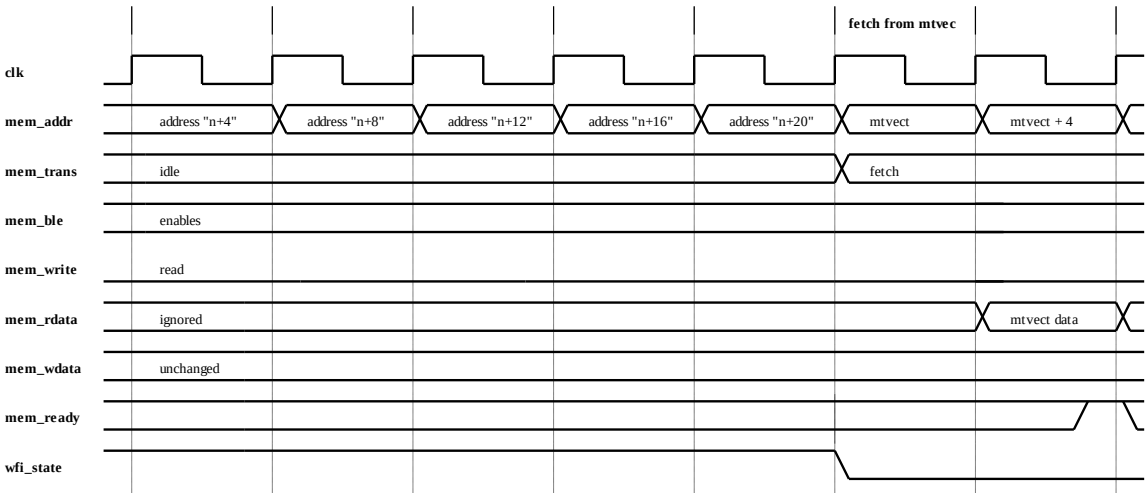


Wait For Interrupt

The WFI instruction stops the CPU pipeline and activates the `wfi_state` output. The timing for the WFI instruction is identical to a branch to the next program location, except that the pipeline then halts.

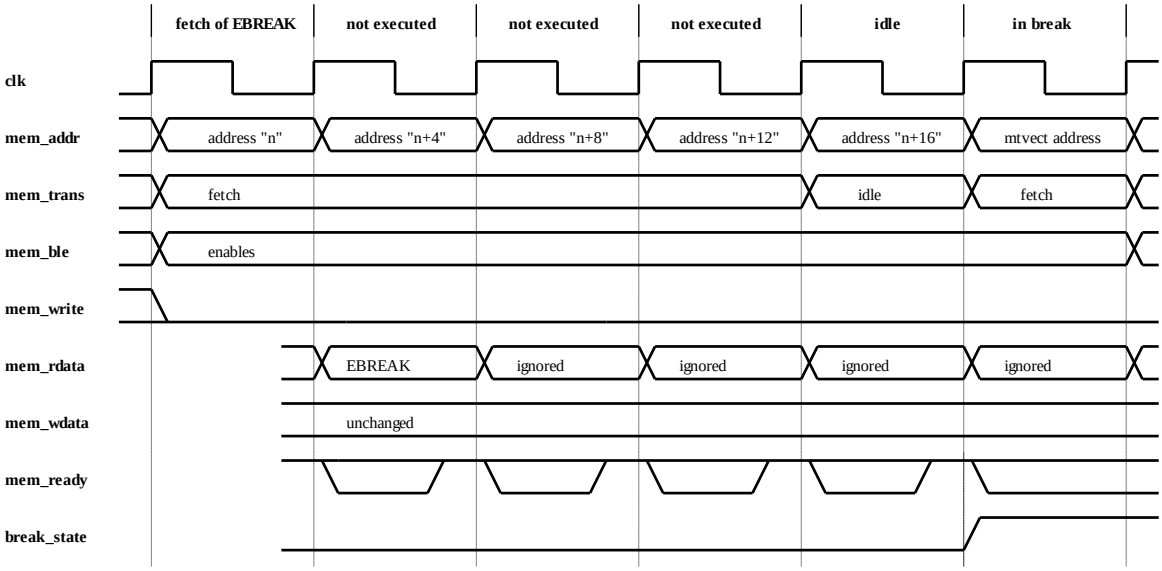


The wait-for-interrupt state is exited when an interrupt is accepted. If an interrupt is pending at the time that the WFI instruction is executed, the pipeline requires a minimum of two clock cycles to start back up, and the wfi_state signal will be active for a total of seven clock cycles. The sequence when exiting the wait-for-interrupt state is shown below.

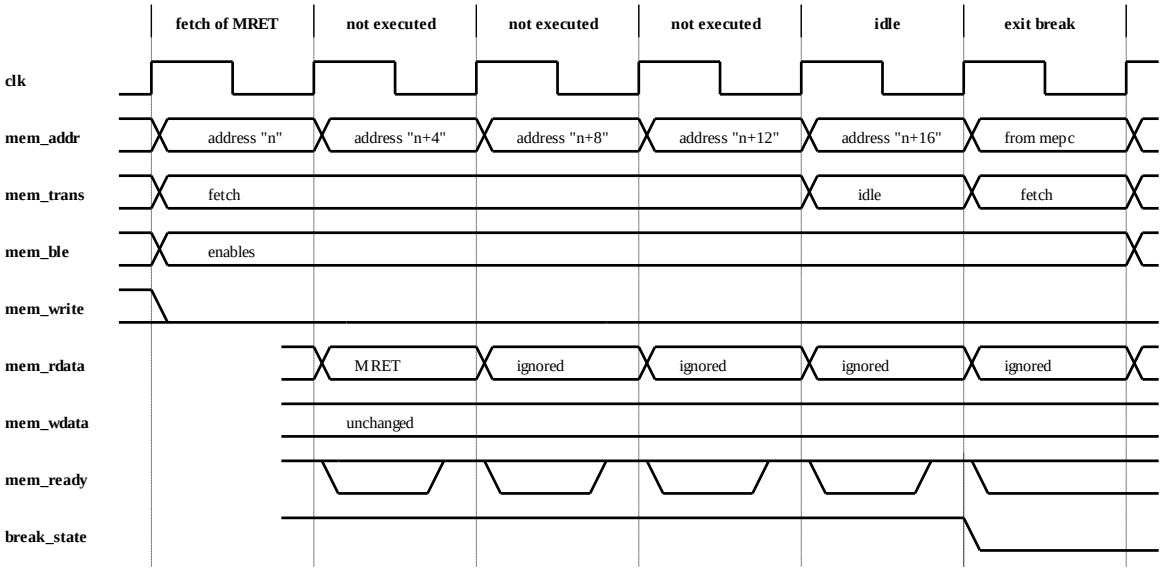


Break

The EBREAK instruction causes a branch to the break entry in the vector table, and the timing is identical to any other program branch.



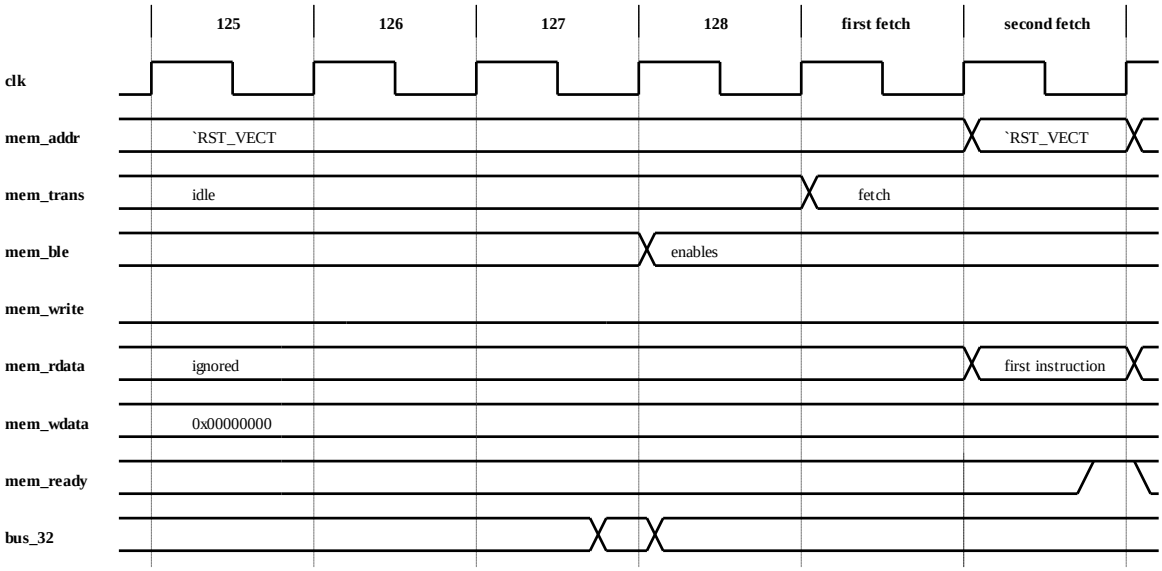
Exiting the break state requires an MRET instruction. Again, the timing is identical to any other program branch.



Reset

The Reset state is entered immediately when the `resetb` signal goes Low, independent of the current state, and this state continues until 128 clock cycles after the `resetb` signal is de-asserted. The `bus_32` signal is sampled during this delay, with the last sample at the end of clock number 127. This latched value remains in effect until `resetb` is asserted again. At the end of clock 128 the processor begins fetching the first instruction from the address specified by the `` `define RST_VECT `` statement in the *yrv_opt.v* design file.

The minimum width of the `resetb` signal is set by the flip-flops used to implement the core. The setup time for the `resetb` signal to the rising edge of the `clk` signal is likewise determined by the flip-flops used to implement the core.



Instruction Set (32-bit opcode)

This section presents the assembly language syntax, operation, binary encoding, and execution time for all implemented 32-bit opcode instructions. The YRV core implements the following RISC-V standards with 32-bit opcodes:

* **RV32I** Base Integer Instruction Set

* **Zca** subset of **RV32C** Standard Extension for Compressed Instructions

* **Zcb** Code Size Reduction Instructions
  (as a logic synthesis option; excludes *c.mul*)

* **RV32A** Atomic Memory Operation Instructions
  (only *amoadd.w*, *aamoand.w*, *amoor.w*, *amoswap.w* and *amoxor.w*)

* **Zam** Standard Extension for Misaligned Atomics

* **Zbb** Basic Bit-manipulation Instructions
  (excluding *clz*, *cpop*, *ctz*, *max*, *maxu*, *min*, *minu* and *orc.b*)

* **Zbkb** Bit-manip Instructions for Cryptography
  (excluding *zip* and *unzip*)

* **Zbs** Single-bit Instructions

* **Zicntr** Standard Extension for Base Counters and Timers
  (as a logic synthesis option; 64-bit, 32-bit or none for each counter and timer)

* **Zicsr** Control and Status Register (CSR) Instructions
  (plus an external hardware interface for custom CSR registers)

* **Zifencei** Instruction-Fetch Fence Instruction

A number of instructions from the full "B" Standard Extension for Bit-manipulation instructions (currently under development) are also included.

The following notation is used in the assembly language syntax:

csr	the CSR address field in the opcode
imm	immediate data contained in the opcode (will be sign-extended as necessary, may be scrambled)
offset	address offset contained in the opcode (will be sign-extended as necessary, may be scrambled)
rd, ***rs1***, ***rs2***	the register select field in the opcode
sbsel	unsigned nit select contained in the opcode
shamt	unsigned shift amount contained in the opcode
uimm	unsigned immediate data contained in the opcode

The operation of each instruction is specified in a format similar to Verilog HDL for minimum ambiguity, but no descriptive text or examples are included. The following notation is used:

imm, offset	immediate data or offset value from the opcode
rd, rs1, rs2	the contents of the indicated register
uimm, sbsel, shamt	immediate data or offset value from the opcode
urs1, urs2	the contents of the indicated register, treated as an unsigned number
@<rs1>	the memory location addressed by the contents of the register
@<csr>	the CSR location addressed by the csr field in the opcode

Unless specifically noted, data less than 32 bits wide is always sign-extended to 32 bits before use.

The instruction encoding is shown with the various fields separated by an underscore (_) to make the fields more apparent. Fields in the instruction encoding are listed using shortcuts for common fields. These shortcuts should be self-explanatory in most cases, but are detailed here for completeness.

`ar`	acquire/release control (ignored by instruction decode)
`cccccccccccc`	12-bit CSR address
`ddd`	***rd'***, 3-bit destination register select (RV32C only)
`ddddd`	***rd***, 5-bit destination register select
`dnzdd`	***rd***, 5-bit non-zero destination register select
`dn2dd`	***rd***, 5-bit destination register select, 0x2 not allowed
`iorw`	4-bit FENCE identifier
`mm...`	immediate data, various widths (may be scrambled)
`sss`	***rs1'***, 3-bit source-1 register select (RV32C only)
`snzss`	***rs1***, 5-bit non-zero source-1 register select
`sssss`	***rs1***, 5-bit source-1 register select
`ttt`	***rs2'***, 3-bit source-2 register select (RV32C only)
`tnztt`	***rs2***, 5-bit non-zero source-2 register select
`ttttt`	***rs2***, 5-bit source-2 register select
`xx...`	don't-care bits, ignored by instruction decode

The most common fields in an opcode specify a CPU register, employing one of the following encodings:

RV32I ***rd, rs1, rs2*** **(5-bit field)**	**RV32C** ***rd', rs1', rs2'*** **(3-bit field)**	**Register Number**	**Register ABI Name**	**Register ABI Description**
00000		x0	zero	Hard-wired zero
00001		x1	ra	Return address
00010		x2	sp	Stack pointer
00011		x3	gp	Global pointer
00100		x4	tp	Thread pointer
00101		x5	t0	Temporary register
00110		x6	t1	Temporary register
00111		x7	t2	Temporary register
01000	000	x8	s0/fp	Saved reg/frame pointer
01001	001	x9	s1	Saved register
01010	010	x10	a0	Function arg/return value
01011	011	x11	a1	Function arg/return value
01100	100	x12	a2	Function argument
01101	101	x13	a3	Function argument
01110	110	x14	a4	Function argument
01111	111	x15	a5	Function argument
10000		x16	a6	Function argument
10001		x17	a7	Function argument
10010		x18	s2	Saved register
10011		x19	s3	Saved register
10100		x20	s4	Saved register
10101		x21	s5	Saved register
10110		x22	s6	Saved register
10111		x23	s7	Saved register
11000		x24	s8	Saved register
11001		x25	s9	Saved register
11010		x26	s10	Saved register
11011		x27	s11	Saved register
11100		x28	t3	Temporary register
11101		x29	t4	Temporary register
11110		x30	t5	Temporary register
11111		x31	t6	Temporary register

Execution time is listed as some number of clocks, specified by three comma-separated numbers. The first number corresponds to the number of clock cycles on the bus with Instruction Fetch status. The second number corresponds to the number of clock cycles on the bus with Load/Store status, and the third number corresponds to the number of Idle cycles on the bus.

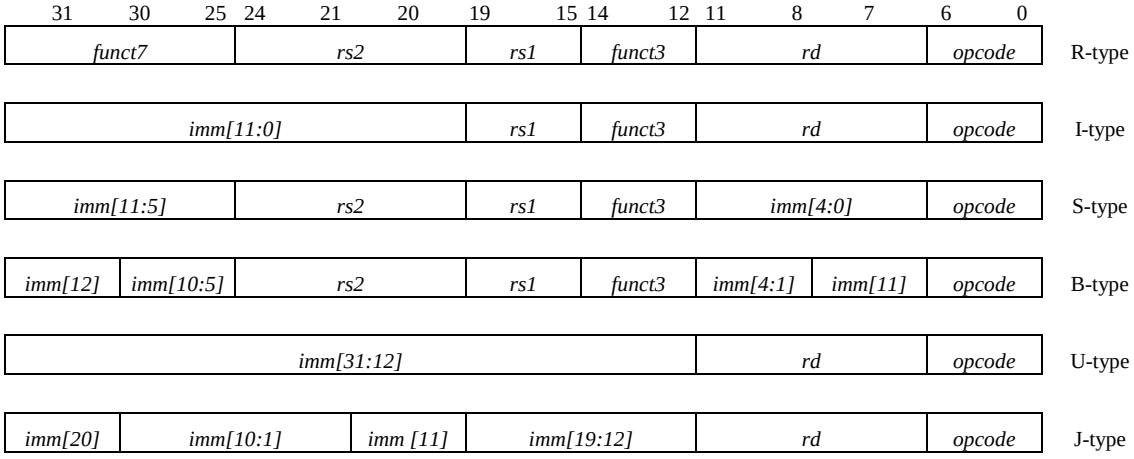
Instruction Fetch cycles can be extended by Wait states, and the number of Instruction Fetch cycles must be multiplied by two in the case of a 16-bit bus.

Load/Store cycles can also be extended by Wait states. Because Load and Store instructions allow for unaligned data, there will usually be a number of possibilities for the number of Load/Store cycles, and this will be shown in the execution time numbers.

Idle cycles cannot be extended by Wait states, so this is a fixed number, independent of Wait states. Idle cycles are used when accessing CSR registers, when waiting for Load data to be registered, and when a branch condition has been recognized soon enough to halt instruction fetches.

With a 32-bit bus if two compressed instructions are packed into a single word, or if a compressed instruction is unaligned, an additional cycle will be required to execute the unaligned compressed instruction. It is not possible to recognize these cases in time to issue an Idle cycle on the bus, so a normal Instruction Fetch cycle will result, which means that Wait states may also be added.

The RISC-V opcode formats are shown below. Immediate data in a 32-bit opcode is scrambled so that the MSB of immediate data is always in bit 31 of the opcode to facilitate sign extension.



ADD

Add Register

ADD *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 + rs2

Encoding: `0000000_ttttt_sssss_000_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

ADDI

Add Immediate

ADDI *rd, rs1, imm*

rd: x0-x31
rs1: x0-x31
imm: imm[11:0]

Operation: rd <= rs1 + { {20{imm[11]}}, imm[11:0]}

Encoding: `mmmmmmmmmmmm_sssss_000_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

1. The **MV *rd*, *rs*** pseudo-instruction is just **ADDI *rd*, *rs*, 0**.
2. The **NOP** pseudo-instruction is just **ADDI x0, x0, 0**.

AMOADD.W

Atomic Memory Add

AMOADD.W *aqrl, rd, rs1, rs2* ***aqrl***: ignored
rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

```
fork
   rd <= @<rs1>
   @<rs1> <= rs2 + @<rs1>
join
```

Encoding: `00000_ar_ttttt_sssss_010_ddddd_01011_11`

R-type "A" Extension

Clocks: 1, 2, 1 (32-bit bus, *rs1* address word-aligned)
1, 4, 1 (32-bit bus, *rs1* address not word-aligned)
1, 4, 1 (16-bit bus, *rs1* address halfword-aligned)
1, 6, 1 (16-bit bus, *rs1* address unaligned)

Notes:

AMOAND.W

Atomic Memory AND

AMOAND.W *aqrl, rd, rs1, rs2* ***aqrl***: ignored
rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

```
fork
   rd <= @<rs1>
   @<rs1> <= rs2 & @<rs1>
join
```

Encoding: `01100_ar_ttttt_sssss_010_ddddd_01011_11`

R-type "A" Extension

Clocks: 1, 2, 1 (32-bit bus, *rs1* address word-aligned)
1, 4, 1 (32-bit bus, *rs1* address not word-aligned)
1, 4, 1 (16-bit bus, *rs1* address halfword-aligned)
1, 6, 1 (16-bit bus, *rs1* address unaligned)

Notes:

AMOOR.W

Atomic Memory OR

AMOOR.W *aqrl, rd, rs1, rs2* ***aqrl***: ignored
rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

```
fork
   rd <= @<rs1>
   @<rs1> <= rs2 | @<rs1>
join
```

Encoding: `01000_ar_ttttt_sssss_010_ddddd_01011_11`

R-type "A" Extension

Clocks: 1, 2, 1 (32-bit bus, *rs1* address word-aligned)
1, 4, 1 (32-bit bus, *rs1* address not word-aligned)
1, 4, 1 (16-bit bus, *rs1* address halfword-aligned)
1, 6, 1 (16-bit bus, *rs1* address unaligned)

Notes:

AMOSWAP.W

Atomic Memory Swap

AMOSWAP.W *aqrl, rd, rs1, rs2* ***aqrl***: ignored
rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

```
fork
   rd <= @<rs1>
   @<rs1> <= rs2
join
```

Encoding: `00001_ar_ttttt_sssss_010_ddddd_01011_11`

R-type "A" Extension

Clocks: 1, 2, 1 (32-bit bus, *rs1* address word-aligned)
1, 4, 1 (32-bit bus, *rs1* address not word-aligned)
1, 4, 1 (16-bit bus, *rs1* address halfword-aligned)
1, 6, 1 (16-bit bus, *rs1* address unaligned)

Notes:

AMOXOR.W

Atomic Memory Exclusive-OR

AMOXOR.W *aqrl, rd, rs1, rs2* ***aqrl***: ignored
rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

```
fork
   rd <= @<rs1>
   @<rs1> <= rs2 ^ @<rs1>
join
```

Encoding: `00100_ar_ttttt_sssss_010_ddddd_01011_11`

R-type "A" Extension

Clocks: 1, 2, 1 (32-bit bus, *rs1* address word-aligned)
1, 4, 1 (32-bit bus, *rs1* address not word-aligned)
1, 4, 1 (16-bit bus, *rs1* address halfword-aligned)
1, 6, 1 (16-bit bus, *rs1* address unaligned)

Notes:

AND

Logical AND Register

AND *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 & rs2

Encoding: `0000000_ttttt_sssss_111_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

ANDI

Logical AND Immediate

ANDI *rd*, *rs1*, *imm*

rd: x0-x31
rs1: x0-x31
imm: imm[11:0]

Operation: rd <= rs1 & { {20{imm[11]}}, imm[11:0]}

Encoding: mmmmmmmmmmmm_sssss_111_ddddd_00100_11

I-type RV32I

Clocks: 1, 0, 0

Notes:

1. The **ZEXT.B *rd*, *rs*** pseudo-instruction is just **ANDI *rd*, *rs*, 0xff**.

ANDN

Logical AND-With-Negate Register

ANDN *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 & ~rs2

Encoding: `0100000_ttttt_sssss_111_ddddd_01100_11`

R-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

AUIPC

Add Upper Immediate to PC

AUIPC *rd*, *imm* ***rd***: x0-x31
imm: offset[31:12]

Operation: rd <= PC + {offset[31:12], 12'h0}

Encoding: mmmmmmmmmmmmmmmmmmmm_ddddd_00101_11

U-type RV32I

Clocks: 1, 0, 0

Notes:

1. The PC used in the computation is the PC of **this** instruction.

BCLR

Bit Clear Register

BCLR *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 & ~(1 << rs2[4:0])

Encoding: `0100100_ttttt_sssss_001_ddddd_01100_11`

R-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BCLRI

Bit Clear Immediate

BCLRI *rd, rs1, sbsel* ***rd***: x0-x31
rs1: x0-x31
sbsel: sbsel[4:0]

Operation: rd <= rs1 & ~(1 << sbsel)

Encoding: `0100100_mmmmm_sssss_001_ddddd_00100_11`

I-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BEQ

Branch if Equal

BEQ *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (rs1 == rs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_000_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The **BEQZ *rs*, *offset*** pseudo-instruction is encoded as **BEQ *rs*, x0, *offset***.

BEXT

Bit Extract Register

BEXT *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= {31'h0, |(rs1 & (1 << rs2[4:0]))}

Encoding: `0100100_ttttt_sssss_101_ddddd_01100_11`

R-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BEXTI

Bit Extract Immediate

BEXTI *rd, rs1, sbsel* ***rd***: x0-x31
rs1: x0-x31
sbsel: sbsel[4:0]

Operation: rd <= {31'h0, |(rs1 & (1 << sbsel))}

Encoding: `0100100_mmmmm_sssss_101_ddddd_00100_11`

I-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BGE

Branch if Greater-than or Equal

BGE *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (rs1 >= rs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_101_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The **BGEZ *rs*, *offset*** pseudo-instruction is just **BGE *rs*, x0, *offset***.

3. The **BLE *rs*, *rt*, *offset*** pseudo-instruction is just **BGE *rt*, *rs*, *offset***.

4. The **BLEZ *rs*, *offset*** pseudo-instruction is just **BGE x0, *rs*, *offset***.

BGEU

Branch if Greater-than or Equal (Unsigned)

BGEU *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (urs1 >= urs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_111_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The contents of registers ***rs1*** and ***rs2*** are treated as unsigned numbers.

3. The **BLEU *rs*, *rt*, *offset*** pseudo-instruction is just **BGEU *rt*, *rs*, *offset***.

BINV

Single-Bit Invert Register

BINV *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 ^ (1 << rs2[4:0])

Encoding: `0110100_ttttt_sssss_001_ddddd_01100_11`

R-type "Zbs" extension

Clocks: 1, 0, 0

Notes:

BINVI

Single-Bit Invert Immediate

BINVI *rd, rs1, sbsel* ***rd***: x0-x31
rs1: x0-x31
sbsel: sbsel[4:0]

Operation: rd <= rs1 ^ (1 << sbsel)

Encoding: `0110100_mmmmm_sssss_001_ddddd_00100_11`

I-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BLT

Branch if Less-Than

BLT *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (rs1 < rs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_100_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The **BLTZ *rs*, *offset*** pseudo-instruction is just **BLT *rs*, x0, *offset***.

3. The **BGT *rs*, *rt*, *offset*** pseudo-instruction is just **BLT *rt*, *rs*, *offset***.

4. The **BGTZ *rs*, *offset*** pseudo-instruction is just **BLT x0, *rs*, *offset***.

BLTU

Branch if Less-than (Unsigned)

BLTU *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (urs1 < urs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_110_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The **BGTU *rs*, *rt*, *offset*** pseudo-instruction is just **BLTU *rt*, *rs*, *offset***.

BNE

Branch if Not Equal

BNE *rs1*, *rs2*, *offset*

rs1: x0-x31
rs2: x0-x31
offset: offset[12:1]

Operation: if (rs1 != rs2) PC <= PC + { {19{offset[12]}}, offset[12:1], 1'b0}

Encoding: `mmmmmmm_ttttt_sssss_001_mmmmm_11000_11`

B-type RV32I

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, word-aligned destination)
5, 0, 1 (taken, 32-bit bus, halfword-aligned destination)
4, 0, 1 (taken, 16-bit bus)

Notes:

1. The PC used in the computation is the PC of **this** instruction.

2. The **BNEZ *rs*, *offset*** pseudo-instruction is just **BNE *rs*, x0, *offset***.

BSET

Bit Set Register

BSET *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 | (1 << rs2[4:0])

Encoding: `0010100_ttttt_sssss_001_ddddd_01100_11`

R-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

BSETI

Bit Set Immediate

BSETI *rd, rs1, sbsel* ***rd***: x0-x31
rs1: x0-x31
sbsel: sbsel[4:0]

Operation: rd <= rs1 | (1 << sbsel)

Encoding: `0010100_mmmmm_sssss_001_ddddd_00100_11`

I-type "Zbs" Extension

Clocks: 1, 0, 0

Notes:

CSRRC

Atomic Read and Clear Bits in CSR

CSRRC *rd*, *csr*, *rs1* ***rd***: x0-x31
rs1: x0-x31
csr: csr[11:0]

Operation:

```
fork
   rd <= @<csr>
   if (rs1 != x0) @<csr> <= @<csr> & ~rs1
join
```

Encoding: `ccccccccccc_sssss_011_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRC *rd*, *csr*, x0** no CSR write occurs, so no Illegal Instruction exception will be generated when accessing a read-only CSR address.

4. The **CSRC *csr*, *rs*** pseudo-instruction is just **CSRRC x0, *csr*, *rs***.

CSRRCI

Atomic Read and Clear Bits Immediate in CSR

CSRRC *rd*, *csr*, *uimm* ***rd***: x0-x31
uimm: uimm[4:0]
csr: csr[11:0]

Operation:

```
fork
   rd <= @<csr>
   if (uimm[4:0] != 0) @<csr> <= @<csr> & ~{27'h0, uimm[4:0]}
join
```

Encoding: `ccccccccccc_mmmmm_111_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRCI *rd*, *csr*, 0** no CSR write occurs, so no Illegal Instruction exception will be generated when accessing a read-only CSR address.

4. The **CSRCI *csr*, *uimm*** pseudo-instruction is just **CSRRCI x0**, ***csr***, ***uimm***.

CSRRS

Atomic Read and Set Bits in CSR

CSRRS *rd*, *csr*, *rs1* ***rd***: x0-x31
rs1: x0-x31
csr: csr[11:0]

Operation:
```
fork
   rd <= @<csr>
   if (rs1 != x0) @<csr> <= @<csr> | rs1
join
```

Encoding: `ccccccccccc_sssss_010_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRS *rd*, *csr*, x0** no CSR write occurs, so no Illegal Instruction exception will be generated when accessing a read-only CSR address.

4. The **CSRR *rd*, *csr*** pseudo-instruction is just **CSRRS *rd*, *csr*, x0**.

5. The **CSRS *csr*, *rs*** pseudo-instruction is just **CSRRS x0, *csr*, *rs***.

6. **RDCYCLE(H) *rd*** is just **CSRRS *rd*, RDCYCLE(H), x0**.

7. **RDINSTRET(H) *rd*** is just **CSRRS *rd*, RDINSTRET(H), x0**.

8. **RDTIME(H) *rd*** is encoded as **CSRRS *rd*, RDTIME(H), x0**.

CSRRSI

Atomic Read and Set Bits Immediate in CSR

CSRRSI *rd*, *csr*, *uimm*

rd: x0-x31
uimm: uimm[4:0]
csr: csr[11:0]

Operation:

```
fork
   rd <= @<csr>
   if (uimm[4:0] != 0) @<csr> <= @<csr> | {27'h0, uimm[4:0]}
join
```

Encoding: `ccccccccccc_mmmmm_110_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRSI *rd*, *csr*, 0** no CSR write occurs, so no Illegal Instruction exception will be generated when accessing a read-only CSR address.

4. The **CSRSI *csr*, *uimm*** pseudo-instruction is just **CSRRSI x0, *csr*, *uimm***.

CSRRW

Atomic Read and Write in CSR

CSRRW *rd*, *csr*, *rs1* ***rd***: x0-x31
rs1: x0-x31
csr: csr[11:0]

Operation:
```
fork
   if (rd != x0) rd <= @<csr>
   @<csr> <= rs1
join
```

Encoding: `cccccccccccc_sssss_001_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRW x0**, ***csr***, ***rs1*** no CSR read occurs.

4. The **CSRW *csr*, *rs1*** pseudo-instruction is just **CSRRW x0**, ***csr***, ***rs1***.

CSRRWI

Atomic Read and Write Immediate in CSR

CSRRWI *rd*, *csr*, *uimm* ***rd***: x0-x31
uimm: uimm[4:0]
csr: csr[11:0]

Operation:
```
fork
   if (uimm[4:0] != 0) rd <= @<csr>
   @<csr> <= {27'h0, uimm[4:0]}
join
```

Encoding: `ccccccccccc_mmmmm_101_ddddd_11100_11`

I-type "Zicsr" Extension

Clocks: 1, 0, 2

Notes:

1. An invalid CSR address will generate an Illegal Instruction exception.

2. An attempt to write a read-only CSR address will generate an Illegal Instruction exception.

3. In the case of **CSRRWI x0**, ***csr***, ***uimm*** no CSR read occurs.

4. The **CSRWI *csr*, *uimm*** pseudo-instruction is just **CSRRWI x0**, ***csr***, ***uimm***.

DRET

Debug Return

DRET

Operation:

```
if (debug_mode) begin
  PC <= dpc
  debug_mode <= 0
  end
else Illegal Instruction
```

Encoding: `0111101_10010_00000_000_00000_11100_11`

R-type "Sdext" Extension

Clocks: 4, 0, 1

Notes:

1. The **DRET** instruction is only valid while the in Debug mode (i.e. the debug_mode signal is active.) At all other times this opcode will generate an Illegal Instruction exception.

EBREAK

Environment Breakpoint

EBREAK

Operation:

```
if (EBREAK_causes_debug) begin
   dpc <= PC
   dcsr[8:6] <= 3'b001
   PC <= `DBG_VECT
   debug_mode <= 1
   end
else if (~debug_mode) begin
   mepc <= PC
   mcause <= 32'h00000003
   PC <= mtvec
   end
```

Encoding: `0000000_00001_00000_000_00000_11100_11`

I-type RV32I

Clocks: 4, 0, 1

Notes:

1. The PC value written to the **dpc** or **mepc** register is the address of the **EBREAK** instruction.

2. Execution of the **EBREAK** instruction does not increment the Instructions-retired counter.

3. In Debug mode this instruction is normally a **NOP**.

ECALL

Environment Call

ECALL

Operation:

```
if (~debug_mode) begin
  mepc <= PC
  mcause <= 32'h0000000b
  PC <= mtvec
  end
```

Encoding: `0000000_00000_00000_000_00000_11100_11`

I-type RV32I

Clocks: 4, 0, 1

Notes:

1. The PC value written to the **mepc** register is the address of the **ECALL** instruction.

2. Execution of the **ECALL** instruction does not increment the Instructions-retired counter.

3. In Debug mode this instruction is a **NOP**.

FENCE

Memory and I/O Fence

FENCE *pred*, *succ* ***pred***: input/output/read/write
succ: input/output/read/write

Operation: Flush pipeline

Encoding: `mmmm_iorw_iorw_sssss_000_ddddd_00011_11`

I-type RV32I

Clocks: 4, 0, 1

Notes:

1. The ***pred***, ***succ***, ***imm[11:8]***, ***rs1*** and ***rd*** fields in the opcode are ignored in this implementation. This means that any future uses of this opcode will also execute as a pipeline flush.

2. This instruction executes like a jump to the next program address, which guarantees that all instructions in the pipeline will be discarded and fetching restarts with the next instruction.

FENCE.I

Instruction Fence

FENCE.I

Operation: Flush pipeline

Encoding: `mmmmmmm_mmmmm_sssss_001_ddddd_00011_11`

I-type "Zifenci" Extension

Clocks: 4, 0, 1

Notes:

1. The ***imm***, ***rs1*** and ***rd*** fields in the opcode are ignored in this implementation.

2. This instruction executes like a jump to the next program address, which guarantees that all instructions in the pipeline will be discarded and fetching restarts with the next instruction.

JAL

Jump and Link

JAL *rd*, *offset*

rd: x0-x31
offset: offset[20:1]

Operation: rd <= PC++
PC <= PC + { {11{offset[20]}}, offset[20:1], 1'b0}

Encoding: mmmmmmmmmmmmmmmmmmmm_ddddd_11011_11

J-type RV32I

Clocks: 4, 0, 1 (32-bit bus, word-aligned destination)
5, 0, 1 (32-bit bus, halfword-aligned destination)
4, 0, 1 (16-bit bus)

Notes:

1. The PC value saved in ***rd*** is normally PC+4. However, if the instruction was generated from a compressed opcode the saved PC value will be PC+2.

2. The **J *offset*** pseudo-instruction is just **JAL x0**, ***offset***.

3. The **JAL *offset*** pseudo-instruction is just **JAL x1**, ***offset***.

JALR

Jump and Link Register

JALR *rd*, *rs1*, *offset*

rd: x0-x31
rs1: x0-x31
offset: offset[11:0]

Operation:

```
fork
   rd <= PC++
   PC <= (rs1 + { {20{offset[11]}}, offset[11:0]}) & 32'hfffffffe
join
```

Encoding: `mmmmmmmmmmmm_sssss_000_ddddd_11001_11`

I-type RV32I

Clocks: 4, 0, 1 (32-bit bus, word-aligned destination)
5, 0, 1 (32-bit bus, halfword-aligned destination)
4, 0, 1 (16-bit bus)

Notes:

1. The least-significant bit of result of the addition of the contents of ***rs1*** with the offset is discarded. This guarantees that the PC will always be at least halfword-aligned.

2. The PC value saved in ***rd*** is normally PC+4. However, if the instruction was generated from a compressed opcode the saved PC value will be PC+2.

3. The **JALR *rs*** pseudo-instruction is just **JALR x1**, ***rs***, **0**.

4. The **JR *rs*** pseudo-instruction is just **JALR x0**, ***rs***, **0**.

5. The **RET** pseudo-instruction is just **JALR x0**, **x1**, **0**.

LB

Load Byte

LB *rd*, *offset*(*rs1*)

rd: x0-x31
rs1: x0-x31
offset: offset[11:0]

Operation:

```
mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
rd[7:0] <= @<mem_addr>
rd[31:8] <= {24{rd[7]}}
```

Encoding:

```
mmmmmmmmmmmm_sssss_000_ddddd_00000_11
```

I-type RV32I

Clocks: 1, 1, 1

Notes:

LBU

Load Byte Unsigned

LBU *rd*, *offset*(*rs1*)

rd: x0-x31
rs1: x0-x31
offset: offset[11:0]

Operation:

```
mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
rd[7:0] <= @<mem_addr>
rd[31:8] <= 24'h0
```

Encoding: `mmmmmmmmmmmm_sssss_100_ddddd_00000_11`

I-type RV32I

Clocks: 1, 1, 1

Notes:

LH

Load Halfword

LH *rd*, *offset*(*rs1*)

rd: x0-x31
rs1: x0-x31
offset: offset[11:0]

Operation:

mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
rd[7:0] <= @<mem_addr>
rd[15:8] <= @<mem_addr + 1>
rd[31:16] <= {16{rd[15]}}

Encoding: `mmmmmmmmmmmm_sssss_001_ddddd_00000_11`

I-type RV32I

Clocks:

1, 1, 1 (source address halfword-aligned)
1, 2, 1 (source address unaligned)

Notes:

LHU

Load Halfword Unsigned

LHU *rd*, *offset*(*rs1*)

rd: x0-x31
rs1: x0-x31
offset: offset[11:0]

Operation:

```
mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
rd[7:0] <= @<mem_addr>
rd[15:8] <= @<mem_addr + 1>
rd[31:16] <= 16'h0000
```

Encoding: `mmmmmmmmmmmm_sssss_101_ddddd_00000_11`

I-type RV32I

Clocks: 1, 1, 1 (source address halfword-aligned)
1, 2, 1 (source address unaligned)

Notes:

LUI

Load Upper Immediate

LUI *rd*, *imm*

rd: x0-x31
imm: imm[31:12]

Operation: rd <= {imm[31:12], 12'h0}

Encoding: mmmmmmmmmmmmmmmmmmmm_ddddd_01101_11

U-type RV32I

Clocks: 1, 0, 0

Notes:

LW
Load Word

LW *rd*, *imm*(*rs1*)

rd: x0-x31
rs1: x0-x31
imm: offset[11:0]

Operation:

mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
rd[7:0] <= @<mem_addr>
rd[15:8] <= @<mem_addr + 1>
rd[23:16] <= @<mem_addr + 2>
rd[31:24] <= @<mem_addr + 3>

Encoding:

`mmmmmmmmmmmm_sssss_010_ddddd_00000_11`

I-type RV32I

Clocks:

1, 1, 1 (32-bit bus, source address word-aligned)
1, 2, 1 (32-bit bus, source address not word-aligned)
1, 2, 1 (16-bit bus, source address aligned)
1, 3, 1 (16-bit bus, source address unaligned)

Notes:

MRET

Machine-level Return

MRET

Operation: if (~debug_mode) PC <= mepc

Encoding: `0011000_00010_00000_000_00000_11100_11`

R-type Privileged Instruction Set

Clocks: 4, 0, 1

Notes:

1. The **MRET** instruction is used to return from a trap. In the case of aa exception, the **mepc** register, used for the return, contains the address of the instruction that caused the exception, so the contents of the **mepc** may need to be adjusted before the return. In the case of an interrupt exception the **mepc** contains the address of the next instruction to execute, so no adjustment of the **mepc** is required.

2. In Debug mode this instruction is a NOP.

OR

Logical OR Register

OR *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 | rs2

Encoding: `0000000_ttttt_sssss_110_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

ORI

Logical OR Immediate

ORI *rd*, *rs1*, *imm*

rd: x0-x31
rs1: x0-x31
imm: imm[11:0]

Operation: rd <= rs1 | { {20{imm[11]}}, imm[11:0]}

Encoding: mmmmmmmmmmmm_sssss_110_ddddd_00100_11

I-type RV32I

Clocks: 1, 0, 0

Notes:

ORN

Logical OR-With-Negate Register

ORN *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 | ~rs2

Encoding: `0100000_ttttt_sssss_110_ddddd_01100_11`

R-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

PACK

Pack Low Halves of Registers

PACK *rd*, *rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= {rs2[15:0], rs1[15:0]}

Encoding: `0000100_ttttt_sssss_100_ddddd_01100_11`

R-type "Zbkb" Extension

Clocks: 1, 0, 0

Notes:

1. The **ZEXT.H *rd*, *rs*** pseudo-instruction is just **PACK *rd*, *rs*, x0**.

PACKH

Pack Low Bytes of Registers

PACKH *rd*, *rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= {16'h0, rs2[7:0], rs1[7:0]}

Encoding: `0000100_ttttt_sssss_111_ddddd_01100_11`

R-type "Zbkb" Extension

Clocks: 1, 0, 0

Notes:

PACKU

Pack High Halves of Registers

PACKU *rd*, *rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= {rs2[31:16], rs1[31:16]}

Encoding: `0100100_ttttt_sssss_100_ddddd_01100_11`

R-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction was once part of the Bit Manipulation proposal, but is not included in the current version of that extension.

REV

Reverse Bits

REV *rd*, *rs* ***rd***: x0-x31
rs: x0-x31

Operation: rd <= rs[0:31]

Encoding: `0110100_11111_sssss_101_ddddd_00100_11`

I-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction is a particular case of the **GREVI** (Generalized Reverse Immediate) instruction that was once slated for inclusion in the Bit Manipulation Extension.

REV.B

Reverse Bits in Bytes

REVB *rd*, *rs* ***rd***: x0-x31
rs: x0-x31

Operation: rd <= {rs[24:31], rs[16:23], rs[8:15], rs[0:7]}

Encoding: `0110100_00111_sssss_101_ddddd_00100_11`

I-type "Zbkb" Extension

Clocks: 1, 0, 0

Notes:

1. The "Zbkb" Extension uses the mnemonic **BREV8** for this instruction.

REV.H

Reverse Bits, by Halfword

REV.H *rd*, *rs* ***rd***: x0-x31
rs: x0-x31

Operation: rd <= {rs[16:31], rs[0:15]}

Encoding: `0110100_01111_sssss_101_ddddd_00100_11`

I-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction is a particular case of the **GREVI** (Generalized Reverse Immediate) instruction that was once slated for inclusion in the Bit Manipulation Extension.

REV8

Reverse Bytes

REV8 *rd*, *rs* ***rd***: x0-x31
rs: x0-x31

Operation: rd <= {rs[7:0], rs[15:8], rs[23:16], rs[31:24]}

Encoding: `0110100_11000_sssss_101_ddddd_00100_11`

I-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

ROL

Rotate Left Register

ROL *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= (rs1 << rs2[4:0]) | (rs1 >> (32 - rs2[4:0]))

Encoding: `0110000_ttttt_sssss_001_ddddd_01100_11`

R-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

ROR

Rotate Right Register

ROR *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= (rs1 >> rs2[4:0]) | (rs1 << (32 - rs2[4:0]))

Encoding: `0110000_ttttt_sssss_101_ddddd_01100_11`

R-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

RORI

Rotate Right Immediate

RORI *rd, rs1, shamt* ***rd***: x0-x31
rs1: x0-x31
shamt: shamt[4:0]

Operation: rd <= (rs1 >> shamt) | (rs1 << (32-shamt))

Encoding: `0110000_mmmmm_sssss_101_ddddd_00100_11`

I-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

SB

Store Byte

SB *rs2*, *imm*(*rs1*)

rs1: x0-x31
rs2: x0-x31
imm: offset[11:0]

Operation: mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
@<mem_addr> <= rs2[7:0]

Encoding: `mmmmmmm_ttttt_sssss_000_mmmmm_01000_11`

S-type RV32I

Clocks: 1, 1, 0

Notes:

SEXT.B

Sign Extend Byte

SEXT.B *rd*, *rs*

rd: x0-x31
rs: x0-x31

Operation: rd <= { {24{rs[7]}}, rs[7:0]}

Encoding: `0110000_00100_sssss_001_ddddd_00100_11`

I-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

SEXT.H

Sign Extend Halfword

SEXT.H *rd*, *rs*

rd: x0-x31
rs: x0-x31

Operation: rd <= { {16{rs[15]}}, rs[15:0]}

Encoding: `0110000_00101_sssss_001_ddddd_00100_11`

I-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

SH

Store Halfword

SH *rs2*, *imm*(*rs1*)

rs1: x0-x31
rs2: x0-x31
imm: offset[11:0]

Operation:

```
mem_addr <= rs1 + { {20{offset[11]}}, offset[11:0]}
@<mem_addr> <= rs2[7:0]
@<mem_addr + 1> <= rs2[15:8]
```

Encoding: `mmmmmmm_ttttt_sssss_001_mmmmm_01000_11`

S-type RV32I

Clocks: 1, 1, 0 (destination address aligned)
1, 2, 0 (destination address unaligned)

Notes:

SLL

Shift Left Logical Register

SLL *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

shamt <= rs2[4:0]
rd <= rs1 << shamt

Encoding: `0000000_ttttt_sssss_001_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

SLLI

Shift Left Logical Immediate

SLLI *rd*, *rs1*, *shamt*

rd: x0-x31
rs1: x0-x31
shamt: shamt[4:0]

Operation: rd <= rs1 << shamt

Encoding: `0000000_mmmmm_sssss_001_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

SLO

Shift Left Ones Register

SLO *rd*, *rs1*, *rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation:

shamt <= rs2[4:0]
rd <= (rs1 << shamt) | {shamt{1}}

Encoding: `0010000_ttttt_sssss_001_ddddd_01100_11`

R-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction was once part of the Bit Manipulation proposal, but is not included in the current version of that extension.

SLOI

Shift Left Ones Immediate

SLOI *rd*, *rs1*, *shamt* ***rd***: x0-x31
rs1: x0-x31
shamt: shamt[4:0]

Operation: rd <= (rs1 << shamt) | {shamt{1}}

Encoding: `0010000_mmmmm_sssss_001_ddddd_00100_11`

I-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction was once part of the Bit Manipulation proposal, but is not included in the current version of that extension.

SLT

Set if Less Than Register

SLT *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: if (rs1 < rs2) rd <= 32'h1
else rd <= 32'h0

Encoding: `0000000_ttttt_sssss_010_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

1. The **SLTZ *rd, rs*** pseudo-instruction is encoded as **SLT *rd, rs*, x0**.
2. The **SGTZ *rd, rs*** pseudo-instruction is encoded as **SLT *rd,* x0, *rs***.

SLTI

Set if Less Than Immediate

SLTI *rd, rs1, imm*

rd: x0-x31
rs1: x0-x31
imm: imm[11:0]

Operation: if (rs1 < {{20{imm[11]}}, imm[11:0]}) rd <= 32'h1
else rd <= 32'h0

Encoding: `mmmmmmmmmmmm_sssss_010_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

SLTIU

Set if Less Than Immediate (Unsigned)

SLTIU *rd, rs1, imm*

rd: x0-x31
rs1: x0-x31
imm: im[11:0]

Operation: if (urs1 < u{{20{imm[11]}}, imm[11:0]}) rd <= 32'h1
else rd <= 32'h0

Encoding: mmmmmmmmmmmm_sssss_011_ddddd_00100_11

I-type RV32I

Clocks: 1, 0, 0

Notes:

1. The contents of register *rs1* and the sign-extended immediate value are treated as unsigned numbers as far as the compare is concerned..

2. The **SEQZ *rd, rs*** pseudo-instruction is encoded as **SLTIU *rd*, x0, *rs***.

SLTU

Set if Less Than Register (Unsigned)

SLTU *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: if (urs1 < urs2) rd <= 32'h1
else rd <= 32'h0

Encoding: `0000000_ttttt_sssss_011_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

1. The contents of registers *rs1* and *rs2* are treated as unsigned numbers.
2. The **SNEZ *rd, rs*** pseudo-instruction is encoded as **SLTU *rd*, x0, *rs***.

SRA

Shift Right Arithmetic Register

SRA *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: shamt <= rs2[4:0]
rd <= (rs1 >> shamt) | ({shamt{rs1[31]}} << (32 - shamt))

Encoding: `0100000_ttttt_sssss_101_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

SRAI

Shift Right Arithmetic Immediate

SRAI *rd, rs1, shamt*

rd: x0-x31
rs1: x0-x31
shamt: shamt[4:0]

Operation: rd <= (rs1 >> shamt) | ({shamt{rs1[31]}} << (32 - shamt))

Encoding: `0100000_mmmmm_sssss_101_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

SRL

Shift Right Logical Register

SRL *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: shamt <= rs2[4:0]
rd <= rs1 >> shamt

Encoding: `0000000_ttttt_sssss_101_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

SRLI

Shift Right Logical Immediate

SRLI *rd, rs1, shamt* ***rd***: x0-x31
rs1: x0-x31
shamt: shamt[4:0]

Operation: rd <= rs1 >> shamt

Encoding: `0000000_mmmmm_sssss_101_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

SRO

Shift Right Ones Register

SRO *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: shamt <= rs2[4:0]
rd <= (rs1 >> shamt) | ({shamt{1}} << (32 - shamt))

Encoding: `0010000_ttttt_sssss_101_ddddd_01100_11`

R-type not currently standardized

Clocks: 1, 0, 0

Notes:

1. This instruction was once part of the Bit Manipulation proposal, but is not included in the current version of this extension.

SUB

Subtract Register

SUB *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs2 - rs1

Encoding: `0100000_ttttt_sssss_000_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

1. The **NEG *rd, rs*** pseudo-instruction is encoded as **SUB *rd, rs,* x0**.

SW

Store Word

SW *rs2*, *offset*(*rs1*)

rs1: x0-x31
rs2: x0-x31
offset: offset[11:0]

Operation:

```
maddr <= rs1 + {{20{offset[11]}}, offset[11:0]}
@<maddr> <= rs2[7:0]
@<maddr + 1> <= rs2[15:8]
@<maddr + 2> <= rs2[23:16]
@<maddr + 3> <= rs2[31:24]
```

Encoding: `mmmmmmm_ttttt_sssss_010_mmmmm_01000_11`

S-type RV32I

Clocks:

1, 1, 0 (32-bit bus, destination address word-aligned)
1, 2, 0 (32-bit bus, destination address not word-aligned)
1, 2, 0 (16-bit bus, destination address aligned)
1, 3, 0 (16-bit bus, destination address unaligned)

Notes:

WFI

Wait For Interrupt

WFI

Operation: `wfi_state` <= 1

Encoding: `0001000_00101_00000_000_00000_11100_11`

R-type Privileged Instruction Set

Clocks: 4, 0, 5 (minimum)

Notes:

1. This instruction executes like a jump to the next location, and the **wfi_state** output is activated at the same time that the next address is driven on the address bus. This new address remains static, and the bus remains Idle, until an interrupt is accepted. When an interrupt is accepted the pipeline (PC increment) is restarted, but with the Idle status still signalled on the bus. The **wfi_state** output is deasserted at the same time that the **mtvec** address is driven on the bus and fetching of intructions restarts.

2. The global interrupt enable is forced active while the **wfi_state** signal is active, but individual interrupt enables will still inhibit individual interrupts from causing an exit from the wait-for-interrupt state.

3. The instruction following the **WFI** instruction is not executed before the interrupt is accepted, and the address written to the **mepc** register is the address of the instruction following the **WFI** instruction.

4. An external debug request, signalled via the `brk_req` or `dbg_req` input, will cause the core to exit the wait-for-interrupt state just like an interrupt.

5. In Debug mode the only interrupt accepted is via the `brk_req` or `dbg_req` input.

XNOR

Logical Exclusive-NOR Register

XNOR *rd, rs1, rs2*

rd: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 ^ ~rs2

Encoding: `0100000_ttttt_sssss_100_ddddd_01100_11`

R-type "Zbb" Extension

Clocks: 1, 0, 0

Notes:

XOR

Logical Exclusive-OR Register

XOR *rd, rs1, rs2* ***rd***: x0-x31
rs1: x0-x31
rs2: x0-x31

Operation: rd <= rs1 ^ rs2

Encoding: `0000000_ttttt_sssss_100_ddddd_01100_11`

R-type RV32I

Clocks: 1, 0, 0

Notes:

XORI

Logical Exclusive-OR Immediate

XORI *rd, rs1, imm* — ***rd***: x0-x31; ***rs1***: x0-x31; ***imm***: imm[11:0]

Operation: rd <= rs1 ^ {{20{imm[11]}}, imm[11:0]}

Encoding: `mmmmmmmmmmmm_sssss_100_ddddd_00100_11`

I-type RV32I

Clocks: 1, 0, 0

Notes:

1. The **NOT *rd, rs*** pseudo-instruction is encoded as **XORI *rd, rs,* -1**.

ILLEGAL

Defined Illegal Instruction

ILLEGAL

Operation: Illegal Instruction

Encoding: `1111111_11111_11111_111_11111_11111_11`

R-type

Clocks: 4, 0, 1

Notes:

1. All ones in an instruction word always generates an Illegal Instruction exception. Exceptions do not increment the Instructions-retired counter.

2. The compressed opcode decoder outputs this opcode when an Illegal Instruction exception is called for.

Pseudo-instructions

A number RV32I instructions have useful special cases when one of the operands is `x0` or the immmediate data is zero or negative-one. These special case are listed below:

Pseudo-instruction	Name	Base instruction
BEQZ *rs, offset*	Branch if = zero	**BEQ *rs*, x0, *offset***
BGEZ *rs, offset*	Branch if ≥ Zero	**BGE *rs*, x0, *offset***
BGT *rs, rt, offset*	Branch if >	**BLT *rt, rs, offset***
BGTU *rs, rt, offset*	Branch if >, unsigned	**BLTU *rt, rs, offset***
BGTZ *rs, offset*	Branch if > Zero	**BLT x0, *rs, offset***
BLE *rs, rt, offset*	Branch if ≤	**BGE *rt, rs, offset***
BLEU *rs, rt, offset*	Branch if ≤, unsigned	**BGEU *rt, rs, offset***
BLEZ *rs, offset*	Branch if ≤ Zero	**BGE x0, *rs, offset***
BLTZ *rs, offset*	Branch if < Zero	**BLT *rs*, x0, *offset***
BNEZ *rs, offset*	Branch if ≠ Zero	**BNE *rs*, x0, *offset***
CSRC *csr, rs*	Clear Bits in CSR	**CSRRC x0, *csr, rs***
CSRCI *csr, imm*	Clear Bits in CSR, immediate	**CSRRCI x0, *csr, imm***
CSRR *rd, csr*	Read CSR	**CSRRS *rd, csr*, x0**
CSRS *csr, rs*	Set Bits in CSR	**CSRRS x0, *csr, rs***
CSRSI *csr, imm*	Set Bits in CSR, mmediate	**CSRRSI x0, *csr, imm***
CSRW *csr, rs*	Write CSR	**CSRRW x0, *csr, rs***
CSRWI *csr, imm*	Write CSR, immediate	**CSRRWI x0, *csr, imm***
J *offset*	Jump	**JAL x0, *offset***
JAL *offset*	Jump and Link	**JAL x1, *offset***
JR *rs*	Jump Register	**JALR x0, *rs*, 0**
JALR *rs*	Jump and Link Register	**JALR x1, *rs*, 0**
MV *rd, rs*	Copy Register	**ADDI *rd, rs*, 0**
NEG *rd, rs*	Two's Complement	**SUB *rd*, x0, *rs***
NOP	No Operation	**ADDI x0, x0, 0**
NOT *rd*, *rs*	One's Complement	**XORI *rd, rs*, -1**
RDCYCLE *rd*	Read Cycle Counter	**CSRRS *rd*, 0xc00, x0**

RDCYCLEH *rd*	Read Cycle Counter MS Word	**CSRRS *rd*, 0xc80, x0**
RDINSTRET *rd*	Read Instructions-retired	**CSRRS *rd*, 0xc02, x0**
RDINSTRETH *rd*	Read Instructions-retired MS Word	**CSRRS *rd*, 0xc82, x0**
RDTIME *rd*	Read Timer	**CSRRS *rd*, 0xc01, x0**
RDTIMEH *rd*	Read Timer MS Word	**CSRRS *rd*, 0xc81, x0**
RET	Return from Subroutine	**JALR x0, x1, 0**
SEQZ *rd, rs*	Set if = Zero	**SLTIU *rd, rs*, 1**
SGTZ *rd, rs*	Set if > Zero	**SLT *rd*, x0, *rs***
SLTZ *rd, rs*	Set if < Zero	**SLT *rd, rs*, x0**
SNEZ *rd, rs*	Set if ≠ Zero	**SLTU *rd*, x0, *rs***
ZEXT.B *rd, rs*	Zero Extend Byte	**ANDI *rd, rs*, 0xff**
ZEXT.H *rd*, *rs*	Zero Extend Halfword	**PACK *rd*, *rs*, x0**.

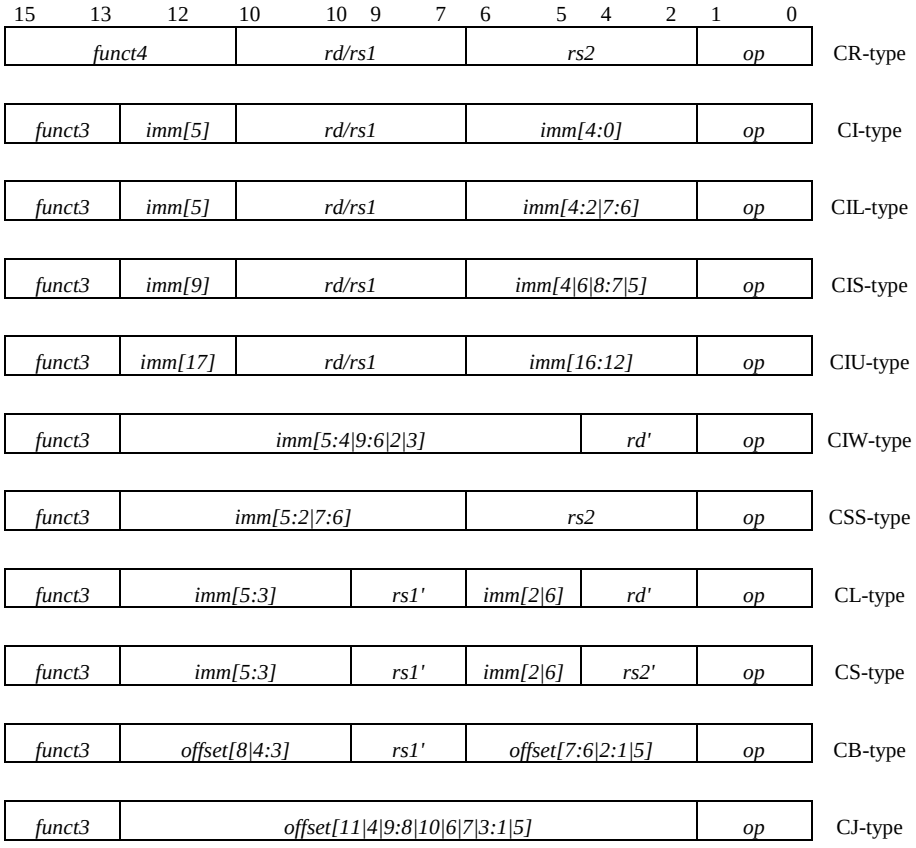
Instruction Set (16-bit opcode)

This section presents the assembly language syntax, operation, binary encoding, and execution time for all implemented 16-bit opcode instructions. The YRV core implements the following RISC-V standards with 16-bit opcodes:

* **Zca** subset of **RV32C** Standard Extension for Compressed Instructions

* **Zcb** Code Size Reduction Instructions
  (as a logic synthesis option; excludes *c.mul*)

The RV32C opcode formats are shown below. Immediate data in a 16-bit opcode is scrambled to allow for a more efficient implementation. Unless otherwise noted, immediate data is sign-extended to the appropriate width used by a regular RV32I instruction.



C.ADD

Add Register

C.ADD *rd, rs2* ***rd***: x0-x31
rs2: x1-x31

Expands to: ADD *rd, rd, rs2*

Encoding: `1001_ddddd_tnztt_10`

CR-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.ADDI

Add Immediate

C.ADDI *rd, nzimm* ***rd***: x1-x31
nzimm: nzimm[5:0]

Expands to: ADDI *rd*, *rd*, {{6{*nzimm*[5]}}, *nzimm*[5:0]}

Encoding: `000_m_dnzdd_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. The case of ***rd*** = **x0** and ***nzimm*** = **0** is defined as **C.NOP**.

2. In revision 0.0 of this design if ***rd*** ≠ **x0** and ***nzimm*** = **0** an Illegal Instruction exception is generated. The ratified "C" Extension specifies that this case should be a *HINT*. This behavior will be corrected in revision 1.0.

C.ADDI16SP

Add Immediate (scaled by 16) to Stack Pointer

C.ADDI16SP *nzimm* ***nzimm***: nzimm[9:4]

Expands to: ADDI x2, x2, {{2{nzimm[9]}}, nzimm[9:4], 4'h0}

Encoding: `011_m_00010_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. ***nzimm*** = **0** generates an Illegal Instruction exception.

2. Normally x2, used as the stack pointer, will be word-aligned, so the result will also be word-aligned.

C.ADDI4SPN

Add Immediate (scaled by 4) to Stack Pointer

C.ADDI4SPN *rd', nzuimm* ***rd'***: x8-x15
nzuimm: nzuimm[9:2]

Expands to: ADDI *rd'*, `x2`, {2'h0, nzuimm[9:2], 2'h0}

Encoding: `000_mmmmmmmm_ddd_00`

CIW-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. ***nzuimm*** = **0** generates an Illegal Instruction exception.

2. Normally x2, used as the stack pointer, will be word-aligned, so the result will also be word-aligned.

C.AND

AND Register

C.AND *rd', rs2'*

rd': x8-x15
rs2': x8-x15

Expands to: AND *rd', rd', rs2'*

Encoding: `1000_11_ddd_11_ttt_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.ANDI

AND Immediate

C.ANDI *rd', imm* ***rd'***: x8-x15
imm: imm[5:0]

Expands to: ANDI *rd', rd',* {{6{imm[5]}}, imm[5:0]}

Encoding: `100_m_10_ddd_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.BEQZ

Branch if Equal to Zero

C.BEQZ *rs1', offset* ***rs1'***: x8-x15
offset: offset[8:1]

Expands to: BEQ *rs1'*, `x0`, {{3{offset[8]}}, offset[8:1], 1'h0}

Encoding: `110_mmm_sss_mmmmm_01`

CB-type "C" Extension

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, destination address word-aligned)
5, 0, 1 (taken, 32-bit bus, destination address halfword-aligned)
4, 0, 1 (taken, 16-bit bus)

Notes:

C.BNEZ

Branch if Equal to Zero

C.BNEZ *rs1', offset* ***rs1'***: x8-x15
offset: offset[8:1]

Expands to: BNE *rs1'*, `x0`, {{3{offset[8]}}, offset[8:1], 1'h0}

Encoding: `111_mmm_sss_mmmmm_01`

CB-type "C" Extension

Clocks: 1, 0, 0 (not taken)
4, 0, 1 (taken, 32-bit bus, destination address word-aligned)
5, 0, 1 (taken, 32-bit bus, destination address halfword-aligned)
4, 0, 1 (taken, 16-bit bus)

Notes:

C.EBREAK

Environment Breakpoint

C.EBREAK

Expands to: EBREAK

Encoding: `1001_00000_00000_10`

CR-type "C" Extension

Clocks: 4, 0, 1

Notes:

C.J

Jump

C.J *offset* ***offset***: offset[11:1]

Expands to: JAL `x0`, {{9{offset[11]}}, offset[11:1]}

Encoding: `101_mmmmmmmmmmm_01`

CJ-type "C" Extension

Clocks: 4, 0, 1 (32-bit bus, destination address word-aligned)
5, 0, 1 (32-bit bus, destination address halfword-aligned)
4, 0, 1 (16-bit bus)

Notes:

C.JAL

Jump and Link

C.JAL *offset* ***offset***: offset[11:1]

Expands to: JAL `x1`, {{9{offset[11]}}, offset[11:1]}

Encoding: `001_mmmmmmmmmmm_01`

CJ-type "C" Extension

Clocks: 4, 0, 1 (32-bit bus, destination address word-aligned)
5, 0, 1 (32-bit bus, destination address halfword-aligned)
4, 0, 1 (16-bit bus)

Notes:

1. The PC written to x1 is PC+2, which is different from the normal **JAL**, which writes PC+4.

C.JALR

Jump and Link Register

C.JALR *rs1* ***rs1***: x1-x31

Expands to: JALR `x1`, *rs1*, 0

Encoding: `1001_snzss_00000_10`

CR-type "C" Extension

Clocks: 4, 0, 1 (32-bit bus, destination address word-aligned)
5, 0, 1 (32-bit bus, destination address halfword-aligned)
4, 0, 1 (16-bit bus)

Notes:

1. The PC written to x1 is PC+2, which is different from the normal **JALR**, which writes PC+4.

C.JR

Jump Register

C.JR *rs1* ***rs1***: x1-x31

Expands to: JALR `x0`, *rs1*, 0

Encoding: `1000_sssss_00000_10`

CR-type "C" Extension

Clocks: 4, 0, 1 (32-bit bus, destination address word-aligned)
5, 0, 1 (32-bit bus, destination address not word-aligned)
4, 0, 1 (16-bit bus)

Notes:

1. ***rs1*** = **x0** generates an Illegal Instruction exception.

C.LBU

Load Byte Unsigned

C.LBU *rd'*, *uoffset*(*rs1'*)

rd': x8-x15
rs1': x8-x15
uoffset: uoffset[1:0]

Expands to: LBU *rd'*, {10'h0, uoffset}(*rs1'*)

Encoding: `100_000_sss_mm_ddd_00`

modified CL-type "Zcb" Extension

Clocks: 1, 1, 1

Notes:

1. uoffset[1:0] = {opcode[5], opcode[4]}
2. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.LH

Load Halfword

C.LH *rd', uoffset(rs1')* ***rd'***: x8-x15
rs1': x8-x15
uoffset: uoffset[1]

Expands to: LH *rd'*, {10'h0, uoffset, 1'b0}(*rs1'*)

Encoding: `100_001_sss_1m_ddd_00`

modified CL-type "Zcb" Extension

Clocks: 1, 1, 1

Notes:

1. uoffset[1] = opcode[5]
2. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.LHU

Load Halfword Unsigned

C.LHU *rd', uoffset(rs1')* ***rd'***: x8-x15
rs1': x8-x15
uoffset: uoffset[1]

Expands to: LHU *rd'*, {10'h0, uoffset, 1'b0}(*rs1'*)

Encoding: `100_001_sss_0m_ddd_00`

modified CL-type "Zcb" Extension

Clocks: 1, 1, 1

Notes:

1. uoffset[1] = opcode[5]
2. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.LI

Load Immediate

C.LI *rd, imm* ***rd***: x0-x31
imm: imm[5:0]

Expands to: ADDI *rd*, `x0`, {{6{imm[5]}}, imm[5:0]}

Encoding: `010_m_ddddd_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.LUI

Load Upper Immediate

C.LUI *rd, nzimm* ***rd***: x0, x1, x3-x31
nzimm: imm[17:12]

Expands to: ADDI *rd*, `x0`, {{6{nzimm[17]}}, nzimm[17:12]}

Encoding: `011_m_dn2dd_mmmmm_01`

CIU-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. ***nzimm*** = **0** generates an Illegal Instruction exception.
2. The ***rd*** = x2 case is used for **C.ADDI16SP**.

C.LW

Load Word

C.LW *rd'*, *uoffset*(*rs1'*)

rd': x8-x15
rs1': x8-x15
uoffset: uoffset[6:2]

Expands to: LW *rd'*, {5'h0, uoffset[6:2], 2'h0}(*rs1'*)

Encoding: `010_mmm_sss_mm_ddd_00`

CL-type "C" Extension

Clocks: 1, 1, 1 (32-bit bus, source address word-aligned)
1, 2, 1 (32-bit bus, source address not word-aligned)
1, 2, 1 (16-bit bus, source address aligned)
1, 3, 1 (16-bit bus, source address unaligned)

Notes:

C.LWSP

Load Word relative to Stack Pointer

C.LWSP *rd, uoffset* ***rd***: x1-x31
uoffset: offset[7:2]

Expands to: LW *rd*, {4'h0, uoffset[7:2], 2'h0}(x2)

Encoding: `010_m_dnzdd_mmmmm_10`

CI-type "C" Extension

Clocks: 1, 1, 1 (32-bit bus, source address word-aligned)
1, 2, 1 (32-bit bus, source address not word-aligned)
1, 2, 1 (16-bit bus, source address aligned)
1, 3, 1 (16-bit bus, source address unaligned)

Notes:

1. ***rd*** = **0** generates an Illegal Instruction exception.

C.MV

Move Register

C.MV *rd, rs2* ***rd***: x0-x31
rs2: x1-x31

Expands to: ADD *rd*, x0, *rs2*

Encoding: `1000_ddddd_tnztt_10`

CR-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. The ***rs2*** = **x0** case is used for **C.JR**.

C.NOP

No Operation

C.NOP

Expands to: ADDI x0, x0, 0

Encoding: `0000_00000_00000_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

1. Strictly speaking, with the CI-type encoding there are six bits of immediate data in bit positions 12 and 6-2, and only ***nzimm*** = **0** is defined as **C.NOP**, while every other value for this immediate data is a *HINT*. This design performs no operation for every value of *nzimm*.

C.NOT

Bitwise Not

C.NOT *rd′* ***rd′***: x8-x15

Expands to: XORI *rd′*, *rd′*, -1

Encoding: `100_111_ddd_11_101_01`

modified CL-type "Zcb" Extension

Clocks: 1, 0, 0

Notes:

1. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.OR

Logical-OR Register

C.OR *rd', rs2'*

rd': x8-x15
rs2': x8-x15

Expands to: OR *rd', rd', rs2'*

Encoding: `1000_11_ddd_10_ttt_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.SB

Store Byte

C.SB *rs2'*, *uoffset*(*rs1'*)

rs1': x8-x15
rs2': x8-x15
uoffset: uoffset[1:0]

Expands to: SB *rs2'*, {10'h0, uoffset}(*rs1'*)

Encoding: `100_010_sss_mm_ttt_00`

modified CS-type "Zcb" Extension

Clocks: 1, 1, 0

Notes:

1. uoffset[1:0] = {opcode[5], opcode[6]}
2. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.SEXT.B

Sign Extend Byte

C.SEXT.B *rd'* ***rd'***: x8-x15

Expands to: SEXT.B *rd'*, *rd'*

Encoding: `100_111_ddd_11_001_01`

modified CS-type "Zcb" Extension

Clocks: 1, 0, 0

Notes:

1. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.SEXT.H

Sign Extend Halfword

C.SEXT.H *rd'* ***rd'***: x8-x15

Expands to: SEXT.H *rd'*, *rd'*

Encoding: `100_111_ddd_11_011_00`

modified CS-type "Zcb" Extension

Clocks: 1, 0, 0

Notes:

1. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.SH

Store Halfword

C.SH *rs2', uoffset(rs1')* ***rs1'***: x8-x15
rs2': x8-x15
uoffset: uoffset[1]

Expands to: SH *rs2'*, {10'h0, uoffset, 1'b0}(*rs1'*)

Encoding: `100_011_sss_0m_ttt_00`

modified CS-type "Zcb" Extension

Clocks: 1, 1, 0 (32-bit bus, destination address halfword aligned)
1, 2, 0 (32-bit bus, destination address not halfword aligned)
1, 1, 0 (16-bit bus, destination address aligned)
1, 2, 0 (16-bit bus, destination address unaligned)

Notes:

1. uoffset[1] = opcode[5]

2. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.SLLI

Shift Left Logical Immediate

SLLI *rd, shamt* ***rd***: x0-x31
shamt: shamt[4:0]

Expands to: SLLI *rd*, *rd*, shamt[4:0]

Encoding: `0000_ddddd_mmmmm_10`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.SRAI

Shift Right Arithmetic Immediate

SRAI *rd', shamt* — ***rd'***: x8-x15; ***shamt***: shamt[4:0]

Expands to: SRAI *rd, rd,* shamt[4:0]

Encoding: `1000_01_ddd_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.SRLI

Shift Right Logical Immediate

SRLI *rd', shamt* ***rd'***: x8-x15
shamt: shamt[4:0]

Expands to: SRLI *rd, rd,* shamt[4:0]

Encoding: `1000_00_ddd_mmmmm_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.SUB

Subtract register

SUB *rd', rs2'*

rd': x8-x15
rs2': x8-x15

Expands to: SUB *rd', rd', rs2'*

Encoding: `1000_11_ddd_00_ttt_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.SW

Store Word

C.SW *rs2', offset(rs1')* ***rs1'***: x8-x15
rs2': x8-x15
offset: offset[6:2]

Expands to: SW *rs2'*, {{5[offset[6]}}, offset[6:2], 2'h0}(*rs1'*)

Encoding: `110_mmm_sss_mm_ttt_00`

CS-type "C" Extension

Clocks: 1, 1, 0 (32-bit bus, destination address word aligned)
1, 2, 0 (32-bit bus, destination address not word aligned)
1, 2, 0 (16-bit bus, destination address aligned)
1, 3, 0 (16-bit bus, destination address unaligned)

Notes:

C.SWSP

Store Word relative to Stack Pointer

C.SWSP *rs2, uoffset* ***rs2***: x1-x31
uoffset: uoffset[7:2]

Expands to: SW *rs2*, {4'h0, uoffset[7:2], 2'h0}(`x2`)

Encoding: `110_mmmmmm_ttttt_10`

CSS-type "C" Extension

Clocks: 1, 1, 0 (32-bit bus, destination address word aligned)
1, 2, 0 (32-bit bus, destination address not word aligned)
1, 2, 0 (16-bit bus, destination address aligned)
1, 3, 0 (16-bit bus, destination address unaligned)

Notes:

1. Normally x2, used as the stack pointer, will be word-aligned, so the resulting **SW** destination will also be word-aligned.

C.XOR

Logical-Exclusive-OR Register

C.XOR *rd', rs2'*

rd': x8-x15
rs2': x8-x15

Expands to: XOR *rd', rd', rs2'*

Encoding: `1000_11_ddd_01_ttt_01`

CI-type "C" Extension

Clocks: 1, 0, 0

Notes:

C.ZEXT.B

Zero Extend Byte

C.ZEXT.B *rd'* ***rd'***: x8-x15

Expands to: ANDI *rd'*, *rd'*, 0xff

Encoding: `100_111_ddd_10_000_01`

modified CS-type "Zcb" Extension

Clocks: 1, 0, 0

Notes:

1. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.ZEXT.H

Zero Extend Halfword

C.ZEXT.H *rd′* ***rd′***: x8-x15

Expands to: PACK *rd′*, *rd′*, x0

Encoding: `100_111_sss_11_000_01`

modified CS-type "Zcb" Extension

Clocks: 1, 0, 0

Notes:

1. This instruction is enabled with the `` `define ZCB `` statement in the Verilog source.

C.ILLEGAL

Defined Illegal Instruction

C.ILLEGAL

Expands to: Illegal Instruction

Encoding: `0000_00000_00000_00`

CR-type

Clocks: 4, 0, 1

Notes:

1. All zeros in the lower half of an instruction word always generates an Illegal Instruction exception. Exceptions do not increment the Instructions-retired counter.

Control/Status Registers

The RISC-V Privileged Architecture specifies a large number of CSR registers, but only a few of them are currently mandated for all RISC-V implementations. This design implements only the required M-mode CSR registers.

Even though a CSR might be mandated, in a number of cases the specification allows a read value of 0x00000000 to indicate that the full functionality is not present. This design chooses this option where appropriate to reduce logic complexity.

The table below lists the standard Control and Status Registers (CSR) present in the YRV core, along with the mnemonic, read/write status, address and hardware reset state. Attempting to access any CSR address not listed here will result in an Illegal Instruction exception, as will an attempt to write to a read-only CSR address.

Register Name	Mnemonic	R/W	Address	Reset
Machine Status	**mstatus**	R/W	0x300	0x00001880
ISA and Extensions	**misa**	R/W	0x301	`MISA_DEF
Machine Interrupt Enable	**mie**	R/W	0x304	0x00000000
Machine Trap-handler Base Address	**mtvec**	R/W	0x305	`MTVEC_RST
Machine Counter-Inhibit	**mountinhibit**	R/W	0x320	0x00000000
Machine Scratch	**mscratch**	R/W	0x340	`MSCR_RST
Machine Exception Program Counter	**mepc**	R/W	0x341	`MEPC_RST
Machine Trap Cause	**mcause**	R/W	0x342	0x00000000
Machine Trap Value	**mtval**	R/W	0x343	0x00000000
Machine Interrupt Pending	**mip**	R/W	0x344	0x00000000
Lower 32 bits of Cycle Counter	**mcycle**	R/W	0xB00	0x00000001
Debug Control/Status	**dcsr**	R/W	0x7B0	0x00000000
Debug Program Counter	**dpc**	R/W	0x7B1	0x00000000
Debug Scratch 0	**dscratch0**	R/W	0x7B2	0x00000000
Debug Scratch 1	**dscratch1**	R/W	0x7B3	0x00000000
Lower 32 bits of Instructions-retired Counter	**minstret**	R/W	0xB02	0x00000001
Upper 32 bits of Cycle Counter	**mcycleh**	R/W	0xB80	0x00000000
Upper 32 bits of Instructions-retired Counter	**minstreth**	R/W	0xB82	0x00000000
Lower 32 bits if Timer	**time**	RO	0xC01	timer_rdata
Upper 32 bits of Timer	**timeh**	RO	0xC81	timer_rdata
Vendor ID	**mvendorid**	RO	0xF11	`VEND_ID
Architecture ID	**marchid**	RO	0xF12	`ARCH_ID
Implementation ID	**mimpid**	RO	0xF13	`IMP_ID
Hardware Thread ID	**mhartid**	RO	0xF14	hw_id

Machine Status

The Machine Status register is required by the RISC-V specification, but only the bits related to M-mode operation with no memory management are implemented.

Only bits 7 and 3 of this register are read-write, and all other bits always return the values indicated.

31:8	7	6:4	3	2:0
0x000018	**MPIE**	000	**MIE**	000

mstatus (CSR Address 0x300)

The Machine Previous Interrupt Enable bit (**MPIE**, bit 7) can be set and cleared by writing to this register, but its primary use is to automatically save the previous state of the **MIE** bit when an exception or interrupt is accepted. An MRET instruction transfers the state of **MPIE** bit back to the **MIE** bit, restoring the interrupt-enable state to what it was before the trap was accepted, while at the same time setting the **MPIE** bit to one.

The Machine Interrupt Enable bit (**MIE**, bit 3) is set and cleared by writing this register. This is the global interrupt enable control, but the state of this bit is ignored during the Wait-for-interrupt state. When an interrupt or exception is accepted the state of this bit is transferred to the **MPIE** bit, while at the same time this bit is cleared.

A table showing the operation of the **MPIE** and **MIE** bits is shown below. These bits are not affected by any operation not listed in the table below.

Condition	**MPIE**	**MIE**
Hard Reset	High	Low
Exception or Interrupt	**MIE**	Low
MRET instruction	High	**MPIE**
CSR write to **mstatus**	written	written

Machine ISA

The RISC-V specification makes this CSR read-write, and in this core writes to this register are allowed, but the write data is ignored. In this design the read value is constant, set by the `MISA_DEF global macro.



misa (CSR address 0x301)

A value of 0x00000000 is legal, indicating that the register is not implemented. For the this design a value of 0x40800105, indicating that the design is 32-bit (in bits 31 and 30), supports RV32I (in bit 8), the "C" extension (in bit 2), the "A" extension (in bit 0), as well as a non-standard extension (in bit 23) might be appropriate.

In reality, given the proliferation of RISC-V standards, the information provided by the **misa** register is of limited utility, which is why 0x00000000 is the default value for `MISA_DEF.

Machine Interrupt Enable

This register is required by the RISC-V specification, but only the lower halfword is standardized. In this core only the bits relative to M-mode operation in the lower halfword of the register are implemented. The upper halfword is available for custom use and this core implements sixteen *local* interrupts which use this halfword.

Only bits 31-16, 11, 7 and 3 of this register are read-write, and all other bits always return zeros. This register is reset to all zeros by a hard reset but is unaffected by any other type of reset.

31:16	15:12	11	10:8	7	6:4	3	2:0
MLIE[15:0]	0000	**MEIE**	000	**MTIE**	000	**MSIE**	000

mie (CSR address 0x304)

The Machine Local Interrupt Enable bits (**MLIE**[15:0], bits 31-16) are set and cleared by writing to this register. A *local* interrupt will be requested when at least one of these bits, along with the corresponding **MLIP** bit in the **mip** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state.

The Machine External Interrupt Enable bit (**MEIE**, bit 11) is set and cleared by writing to this register. An External interrupt will be requested when this bit, and the **MEIP** bit in the **mip** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state.

The Machine Timer Interrupt Enable bit (**MTIE**, bit 7) is set and cleared by writing to this register. A Timer interrupt will be requested when this bit, and the **MTIP** bit in the **mip** register, are both set; but only if interrupts are enabled or the core is in the Wait-for-interrupt state.

The Machine Software Interrupt Enable bit (**MSIE**, bit 3) is set and cleared by writing to this register. A Software interrupt will be requested when this bit, and the **MSIP** bit in the **mip** register, are both set; but only if interrupts are enabled or the core is in the Wait-for-interrupt state.

Machine Trap-Vector Base Address

The Machine Trap-Vector Base Address register holds the base address of the trap handler as well as control bits to enable vectored-interrupt operation.

All thirty-two bits of the register are read-write. Bits 31-2 of this register are reset to the `MTVEC_RST value in the *yrv_opt.v* design file by a hard reset but are unaffected by any other type of reset. Bits 1:0 of this register are always cleared by a hard reset but are unaffected by any other type of reset.

31:2	1:0
BASE	**VMODE**

mtvec (CSR address 0x305)

The Machine Trap-Vector Base Address (**BASE**, in bits 31:2) holds the address for the first instruction of the trap handler when operating in non-vectored mode. This address must be word-aligned, which is why only thirty bits are required.

In vectored mode only bits 31-8 of **BASE** are used and bits 7:2 are ignored. This means that the vectored mode base address will be aligned on a 256-byte boundary and the vector is formed by a concatenation rather than an addition. The RISC-V specification allows for this method to save logic gates.

The vector offset is formed by shifting the Exception Code from the **mcause** CSR left by two bits, which allows space for exactly one instruction per Exception Code. Typically this will be a BEQZ instruction to branch to a service routine.

The Vector mode bits (**VMODE**, in bits 1:0) control how the address for the trap handler are formed, as shown in the table below.

VMODE	Interrupt starting address	Exception starting address
00	{**BASE**, 2'b00}	{**BASE**, 2'b00}
01	{**BASE**[31:8], CODE, 2'b00}	{**BASE**, 2'b00}
10	{**BASE**, 2'b00}	{**BASE**, 2'b00}
11	{**BASE**, 2'b00}	{**BASE**, 2'b00}

Note that **VMODE** = 10 and **VMODE** = 11 are currently reserved, although the first encoding is used in the "Smclic" proposed standard.

Machine Counter-Inhibit

The Machine Counter-Inhibit register is not required by the RISC-V specification, but the functionality it provides is useful so it is included in this core.

Only bits 2 and 0 are implemented as read/write and all remaining bits are always zero. Bits 2 and 0 of this register are always cleared by a hard reset but are unaffected by any other type of reset.

31:3	2	1	0
0x00000000	**IR**	0	**CY**

mcountinhibit (CSR address 0x320)

The bits in this register inhibit the counting by various counters.

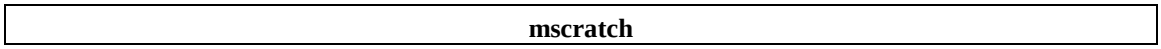
When the Instructions-retired Counter Inhibit (**IR**, bit 2) is clear the Instructions-retired counter counts as usual. When this bit is one the Instructions-retired counter does not increment.

When the Clock-cycle Counter Inhibit (**CY**, bit 0) is clear the Clock-cycle counter counts as usual. When this bit is one the Clock-cycle counter does not increment.

Machine Scratch

This read-write register is typically used by trap handlers to hold a pointer to context information for the trap handler itself, outside of the normal register space.

This register is reset to the `MSCR_RST value in the *yrv_opt.v* design file by a hard reset but is unaffected by any other type of reset.



mscratch (CSR address 0x340)

When a trap is taken a RISC-V processor branches to the address specified in the **mtvec** CSR, either directly or modified by the contents of the **mcause** CSR. However, without a dedicated stack pointer the trap service routine needs a way to store register contents. Hence the need for the **mscratch** CSR.

Machine Exception PC

Whenever a interrupt or trap is taken this read-write register is written with the PC of the instruction that caused the trap, or the address of the next instruction to execute in the case of an interrupt. The least-significant bit of this register is hardwired with zero.

This register is reset to the `MEPC_RST value in the *yrv_opt.v* design file by a hard reset but is unaffected by any other type of reset.

31:1	0
mepc	0

mepc (CSR address 0x341)

The contents of this register are used by the MRET instruction to return from an exception handler. In the case of an ECALL instruction the **mepc** will need to be incremented by 4 to point to the instruction following the ECALL. In the case of interrupts the **mepc** will already point to the next instruction to be executed.

The Non-Maskable Interrupt also writes the address of the next instruction to execute to this register, but because this may overwrite an exisiting valid **mepc** value the Non-Maskable interrupt should only be used for non-returnable (fatal) events.

Machine Trap Cause

This read-write register is used to determine the type of exception or interrupt (local, global, or non-maskable) that occurred, as these status bits are set whenever one of these exceptions is encountered.

Only bits 31 and 5-0 of this register are read-write, and all other bits always return zeros. The read-write bits may be written with any value, but only those values with a cause identified in the table below will ever be returned by the hardware as a result of an interrupt or exception.

31	30:5	5:0
INT	0x000000	**CODE**

mcause (CSR address 0x342)

The Interrupt bit (**INT**, bit 31) is set if the trap was caused by a Local, External, Timer, or Software interrupt. The RISC-V specification says that an NMI "should" set this bit.

The table below shows the Exception codes (**CODE**, in bits 5:0) that will be returned by this core. All of these **CODE** values are globally defined in the *defines_ec.v* file, so they can be modified if necessary.

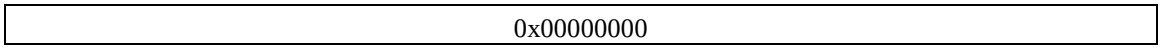
INT	CODE	Interrupt/Exception
0	0x00	Reset or NMI
0	0x02	Illegal instruction
0	0x03	EBREAK instruction
0	0x0B	ECALL instruction
1	0x03	Software interrupt
1	0x07	Timer interrupt
1	0x0B	External interrupt
1	0x20	Local Interrupt 0
1	0x21	Local interrupt 1
1	0x22	Local interrupt 2
	. .	. .
1	0x2D	Local interrupt 13
1	0x2E	Local interrupt 14
1	0x2F	Local interrupt 15

The NMI exception code has not been standardized.

The sixteen Local interrupts utilize exception codes that were originally reserved for custom use in the RISC-V standard, but these exception codes have since been standardized in the proposed RISC-V Advanced Interrupt Architecture, so it may be appropriate to change them in the future.

Machine Trap Value

This register is read-write when fully implemented, but this design chooses the option of hardwiring the register to all zeros, which is allowed by the RISC-V specification.



mtval (CSR address 0x343)

Machine Interrupt Pending

This register is required by the RISC-V specification, but only the lower halfword is standardized. In this core only the bits relative to M-mode operation in the lower halfword of the register are implemented. The upper halfword is available for custom use and this core implements sixteen *local* interrupts which use this halfword.

Only bits 31-16, 11, 7 and 3 of this register are implemented, but only bits 31-16 are read-write. All unused bits always return zeros. This register is reset to all zeros by a hard reset but is unaffected by any other type of reset.

31:16	15:12	11	10:8	7	6:4	3	2:0
MLIP[15:0]	0000	**MEIP**	000	**MTIP**	000	**MSIP**	000

mip (CSR address 0x344)

The Machine Local Interrupt Pending bits (MLIP[15:0], bits 31-16) normally reflect the state of the **li_req** inputs to the core, but the value returned during a read of these bits is actually the logical-OR of the state of the **li_req** inputs and the data latched during a write to this CSR address. This feature allows software to easily generate local interrupt requests if necessary.

A Local interrupt will be requested when at least one of these bits (the logical-OR version), along with the corresponding **MLIE** bit in the **mie** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state. Local interrupts are prioritized with the lower index number having higher priority.

The Machine External Interrupt Pending bit (**MEIP**, bit 11) reflects the state of the **ei_req** input to the core. An External interrupt will be requested when this bit, and the **MEIE** bit in the **mie** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state.

The Machine Timer Interrupt Pending bit (**MTIP**, bit 7) is set by the timer compare logic and is cleared by writing to the memory-mapped Timer Compare register. A Timer interrupt will be requested when this bit, and the **MTIP** bit in the **mip** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state.

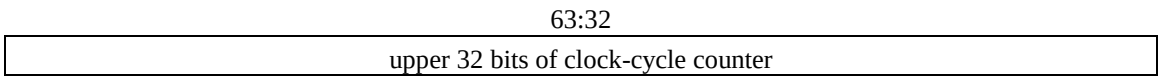
The Machine Software Interrupt Pending bit (**MSIP**, bit 3) is set and cleared by accessing a memory-mapped Software Interrupt control register (which is external to the core). A Software interrupt will be requested when this bit, and the **MSIE** bit in the **mie** register, are both set; but only if interrupts are globally enabled or the core is in the Wait-for-interrupt state.

Simultaneous interrupt and exception requests are prioritized, highest priority to lowest priority, as follows: non-maskable interrupts, local interrupts (lower number has higher priority), external interrupt, software interrupt, timer interrupt, and exceptions.

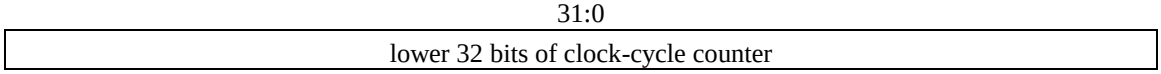
Clock-cycle Counter

The clock-cycle counter is normally a 64-bit counter that increments once per clock cycle. This counter is mandated by the RISC-V specification.

This counter is reset to 0x1 by a hard reset but is unaffected by any other type of reset.



cycleh (CSR address 0xC80, read-only)
mcycleh (CSR address 0xB80)



cycle (CSR address 0xC00, read-only)
mcycle (CSR address 0xB00)

The clock-cycle counter can be restricted to a 32-bit width at logic synthesis time using a `` `define CYCCNT_32 `` statement in the *yrv_opt.v* design file. When this option is selected a Clock-cycle Overflow pulse is generated whenever the 32-bit counter rolls over. If the pulse latched and connected to an interrupt input software can keep track of higher count values. In this case the **cycleh** and **mcycleh** CSR addresses are removed from the list of valid CSR addresses so that an access of the upper 32 bits of the counter will result in an Illegal Instruction exception.

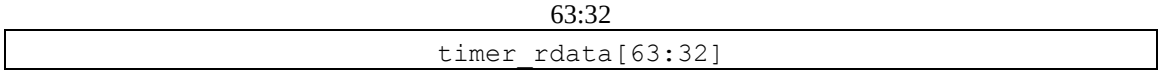
The clock-cycle counter can be completely eliminated at logic synthesis time using a `` `define CYCCNT_0 `` statement in the *yrv_opt.v* design file. In this case all four of **cycle**, **mcycle**, **cycleh** and **mcycleh** CSR addresses are removed from the list of valid CSR addresses.

The table below shows the effect of these macros.

`CYCCNT_0	`CYCCNT_32	width	**cycleh mcycleh**	**cycle mcycle**
not defined	not defined	64 bits	valid	valid
not defined	defined	32 bits	exception	valid
defined	not defined	none	exception	exception
defined	defined	none	exception	exception

Timer

The timer in RISC-V is normally a 64-bit counter that increments at some platform-specific frequency. This timer is accessed via a platform-specific memory address but can be read via these CSR addresses.



timeh (CSR address 0xC81, read-only)



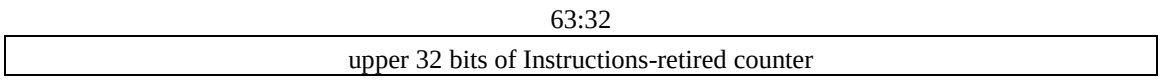
time (CSR address 0xC01, read-only)

The timer is not part of this core, and the timer value is only an input to the core.

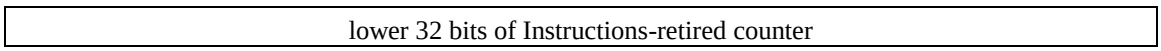
Instructions-retired Counter

The instructions-retired counter is a 64-bit counter that increments by one when each instruction completes. This counter is mandated by the RISC-V specification.

This counter is reset to 0x0 by a hard reset but is unaffected by any other type of reset.



instreth (CSR address 0xC82, read-only)
minstreth (CSR address 0xB82)



instret (CSR address 0xC02, read-only)
minstret (CSR address 0xB02)

The instructions-retired counter can be restricted to a 32-bit width at logic synthesis time using a `` `define INSTRET_32 `` statement in the *yrv_opt.v* design file. When this option is selected an Instructions-retired Overflow pulse is generated whenever the 32-bit counter rolls over. If the pulse latched and connected to an interrupt input software can keep track of higher count values. In this case the **instret** and **minstreth** CSR addresses are removed from the list of valid CSR addresses so that an access of the upper 32 bits of the counter will result in an Illegal Instruction exception.

The instructions-retired counter can be completely eliminated at logic synthesis time using a `` `define INSTRET_0 `` statement in the *yrv_opt.v* design file. In this all four of **instret**, **minstret**, **instreth** and **minstreth** CSR addresses are removed from the list of valid CSR addresses.

The table below shows the effect of these macros.

`INSTRET_0	`INSTRET_32	width	**instreth minstreth**	**instret minstret**
not defined	not defined	64 bits	valid	valid
not defined	defined	32 bits	exception	valid
defined	not defined	none	exception	exception
defined	defined	none	exception	exception

Vendor ID

This read-only register returns the value set by the `define VEND_ID macro in the *yrv_opt.v* design file. The default value of this macro is 0x00000000.

31:0
`VEND_ID

mvendorid (CSR address 0xF11)

Architecture ID

This read-only register returns the value set by the `define ARCH_ID macro in the *yrv_opt.v* design file. The default value of this macro is 0x00000000.

31:0
`ARCH_ID

marchid (CSR address 0xF12)

Implementation ID

This read-only register returns the value set by the `` `define IMP_ID `` macro in the *yrv_opt.v* design file. The default value of this macro is 0x00000000.

31:0
`` `IMP_ID ``

mimpid (CSR address 0xF13)

Hardware Thread ID

This read-only register always returns all zeros in bits 31 through 10, and the value present on the **hw_id** bus into the core in bits 9 through 0.

31:10	9:0
0x000000	hw_id

mhartid (CSR address 0xF14)

In a single-core implementation the **hw_id** bus should be hard-wired to 0x000, and in a multi-core implementation one core must have its **hw_id** bus wired to 0x000.

Debug Control/Status

The Debug Control/Status register provides global status and control for Debug mode. In this core only the mandatory bits related to M-mode operation are implemented.

Only bits 15, 10, 9 and 2 of this register are read-write, and all other bits always return the values indicated.

31:28	27:18	15	14:11	10	9	8:6	5:4	3	2	1:0
`XDEBUGVER	0x000	**ebrkd**	0000	**stopc**	**stopt**	**cause**	00	**nmip**	**step**	11

dcsr (CSR Address 0x7B0)

This register can only be accessed by software while in Debug mode. Any attempt to access the **dcsr** CSR outside of Debug mode will result in an Illegal Instruction trap.

The Debug Version field (bits 31-28) is a read-only value that signals what type of debug support is available in the core. The value is set by the `define XDEBUGVER macro in the *yrv_opt.v* design file. The default value of this macro is 0xF, which the RISC-V Debug standard defines as "debug support, but not conforming to any available version of the spec."

The EBREAK Causes Debug bit (**ebrkd**, bit 15) controls the behavior of the EBREAK instruction. When this bit is zero, the EBREAK instruction operates normally. When this bit is one, an EBREAK instruction will cause the core to enter Debug mode.

The Stop Counters In Debug Mode bit (**stopc**, bit 10) controls the operation of the Clock-cycle and Instructions-retired counters while in Debug mode. If this bit is zero, these two counters continue to operate normally in Debug mode. If this bit is one the two counters stop counting while in Debug mode. The RISC-V Debug standard requires that these counters also not increment on an EBREAK instruction that causes entry into Debug mode. The Instructions-retired counter never increments on an EBREAK instruction, but in this core the Clock-cycle counter will increment on any EBREAK instruction.

The Stop Timers In Debug Mode bit (**stopt**, bit 9) controls the operation of the Timer while in Debug mode. If this bit is zero. The Timer continues to operate normally while in Debug mode. If this bit is one, the `timer_en` output from the core goes Low while in Debug mode to halt the external Timer.

The Debug Entry Cause field (**cause**, bits 8-6) is updated every time that the core enters Debug mode. A table showing the operation of the **cause** bits is shown below. These bits are not affected by any operation not listed in the table below.

The table represents the prioritization of the different **cause** cases, with the Breakpoint request having the highest priority and the Single-step request having the lowest priority.

cause	Debug cause	YRV signal
010	Breakpoint request	brk_req (external signal)
001	EBREAK instruction	ebrk_4_dec
101	Halt request	halt_reg (external signal)
011	Debugger request	dbg_req (external signal)
100	Single-step	ss_reg

The three external signals, `brk_reg`, `halt_reg`, and `dbg_req`, all result in the core entering Debug mode, and the only difference between these three cases is the code returned in the **cause** field. The Breakpoint request (`brk_req`) will typically be generated by a trigger module. The Halt request (`halt_reg`) is reserved for the special case of an immediate entry into Debug mode out of reset. The Debugger request (`dbg_req`) is used when the debugger itself is requesting Debug mode entry.

The Non-maskable Interrupt Pending bit (**nmip**, bit 3) reports that a Non-maskable Interrupt has been signalled during Debug mode. The RISC-V Debug specification requires that no interrupts (not even Non-maskable interrupts) be accepted during Debug mode. Because NMI is often used to signal a hardware failure this bit provides a way to determine that an NMI occurred while in Debug mode and the core may no longer be operating properly. This bit is automatically cleared upon exit from Debug mode.

The Single-step bit (**step**, bit 2) can be set while in Debug mode to enable single-stepping after Debug mode is exited. In single-step mode exactly one instruction will be executed before reentering Debug mode. If this one instruction does not execute, but instead generates an exception, Debug mode will be entered before the first instruction of the trap handler is executed.

Debug Exception PC

Upon Debug mode entry this register is written with the address of the next instruction to execute after leaving Debug mode. The least-significant bit of this register is hardwired with zero.

31:1	0
depc	0

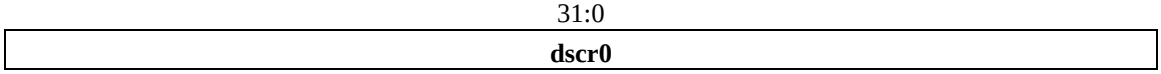
depc (CSR address 0x7B1)

This register can only be accessed by software while in Debug mode. Any attempt to access the **depc** CSR outside of Debug mode will result in an Illegal Instruction trap.

The contents of this register are used by the DRET instruction to return from Debug mode.

Debug Scratch 0 and 1

These read-write registers are typically used by Debug mode to hold a pointer to context information for the Debug mode itself, outside of the normal register space. This is analogous to the **mscratch** register used for traps.



dscr0 (CSR address 0x7B2)



dscr1 (CSR address 0x7B3)

These registers can only be accessed by software while in Debug mode. Any attempt to access the **dscr0** or **dscr1** CSR outside of Debug mode will result in an Illegal Instruction trap.

When entering Debug mode a RISC-V processor branches to an address specified by the `DBG_VECT` macro. However, Debug mode needs a way to store register contents without first disturbing them. Hence the need for the **dcsr0** and **dcsr1** SCRs.

Logic Synthesis Options

There are a number of options available for configuring the YRV core at the time of logic synthesis. All of these options are controlled by entries in the *yrv_opt.v* Verilog design file using global `define text macros.

Read-Only CSR Values

These macro definitions assign data values to read-only CSRs. Read the RISC-V specification before assigning different values to any of these macros.

`MARCHID_DEF` A 32-bit value that is returned in the **marchid** CSR. The default value of this macro is 0x00000000.

`MIMPID_DEF` A 32-bit value that is returned in the **mimpid** CSR. The default value of this macro is 0x00000000.

`MISA_DEF` A 32-bit value that is returned in the **misa** CSR. The default value of this macro is 0x00000000.

`MVENDORID_DEF` A 32-bit value that is returned in the **mvendorid** CSR. The default value of this macro is 0x00000000.

CSR Reset Values

These macro definitions assign a CSR register reset value. These three CSRs are critical for trap handling and must be initialized as soon as possible after reset. Allowing the reset value to be specified during logic synthesis may help with this critical register initialization.

`MEPC_RST` A 32-bit value that is the reset value for the **mepc** CSR. The default value of this macro is 0x00000000.

`MSCR_RST` A 32-bit value that is the reset value for the **mscratch** CSR. The default value of this macro is 0x00000000.

`MTVEC_RST` A 30-bit value that is the reset value for the **BASE** field in the **mtvec** CSR. The default value of this macro is 0x00000040.

Default Addresses

These macro definitions set two addresses that are listed as platform-specific in the RISC-V Specification.

`NMI_VECT` This 32-bit value is the address where a trap caused by an NMI will begin execution. This address is required because an NMI does not use the value in the **mtvec** register, probably because of the initialization issue mentioned previously. This address must be word-aligned. The default value of this macro is 0x00000100.

`RST_BASE` This 31-bit value, shifted left by one bit, is the location of the first instruction fetched by the core after reset. The default value of this macro is 0x00000100, which makes the address of the first instruction fetched 0x00000200.

Debug Operation

These macro definitions are required for Debug mode, which exists outside of the normal operation of a RISC-V core. There are two addresses that are listed as platform-specific in the RISC-V Debug Specification, as well as the version of the debug specification supported by the core.

`DBG_VECT` This 32-bit value is the address where a trap caused by a debug request will begin execution. This address must be word-aligned. The default value of this macro is 0x00000140.

`DEX_VECT` This 32-bit value is the address where a trap that occurs during debug mode will begin execution. This address must be word-aligned. The default value of this macro is 0x000001C0.

`XDEBUGVER` This 4-bit value identifies the version of the Debug specification implemented in the core, which is returned in bits 31:28 of the **dcsr** register. The default value is 0xF, signifying that there is debug support, but that it does not conform to any available version of the specification.

Hardware Features

These macro definitions control certain functions included in the core during logic synthesis.

`ZCB` — If this macro is defined the core will include the instructions included in the "Zcb" Standard Extension. These instructions were not included in the original release of the core, which is why they are enabled separately with a macro definition.

`CYCCNT_64`
`CYCCNT_32`
`CYCCNT_0` — One, and only one, of these options should be chosen to determine the width of the Clock-cycle counter. The RISC-V specification mandates a 64-bit width for this counter, but since not all applications might take advantage of this feature it is made optional. Refer to the **mcounth** and **mcount** CSR description for the details of these choices.

`INSTRET_64`
`INSTRET_32`
`INSTRET_0` — One, and only one, of these options should be chosen to determine the width of the Instructions-retired counter. The RISC-V specification mandates a 64-bit width for this counter, but since not all applications might take advantage of this feature it is made optional. Refer to the **minstreth** and **minstret** CSR description for the details of these choices.

Implementation-specific

These macro definitions set options for logic synthesis.

`ICE40_VERSION`
`SERIES7_VERSION`
`SIM_VERSION` — One, and only one, of these options should be chosen to select the technology target for logic synthesis of the core. If some other target is required add that selection to this list of options and `` `define `` that option.

`INSTANCE_REG` — Defining this macro substitutes a technology-specific logic block, where available, for the register file.

`INSTANCE_ADD` — Defining this macro substitutes a technology-specific logic block, where available, for the adder in the main ALU and the adder in the AMO ALU.

`INSTANCE_SUB` — Defining this macro substitutes a technology-specific logic block, where available, for the subtractor in the branch address calculation.

`INSTANCE_INC` — Defining this macro substitutes a technology-specific logic block, where available, for the incrementer for the PC as well as the incrementer used in the load/store state machine.

`INSTANCE_CNT` — Defining this macro substitutes a technology-specific logic block, where available, for the incrementer in the Clock-cycle and Instructions-retired counters.