OpenCores RV01 RISC-V Processor Core

Version 1.0

December 2017

Contents

Introduction 4
License 4
Conventions 5
Main Features 5
Source code language 5
Core architecture 6
General description 6
Block diagram 7
Pipeline Organization 7
Core interface 8
Memory 12
Instruction and data memories 12
Address space 13
General notes 14
Control and Status Registers 14
CSR access from control port 14
Non-optional, implementation-specific, CSR’s description 16
Debug Module CSR’s description 16
Halt Module CSR’s description 16
Core configuration options summary 17
General notes 17
Size options 17
Instruction, data and I/O memories 17
Branch History Table (BHT) 18
Functional Options 18
Parallel instruction execution 18
Delayed Instructions Execution 19
Branch and Jump-PC instructions prediction 19
Jump-register instructions prediction 19
FPU 20
Debug module 20
Halt module 20
PLIC 20
Flow of operations 21
Using Debug module 21
Using Halt module 22
Performance 22
Dhrystone benchmark 22
Sample timing diagrams 24
Core reset 24
CSR’s access 26
DMA access 28
Starting/Resuming execution 30

Appendix A: Xilinx© Vivado 17.3 synthesis test 32
Appendix B: Simulation & Implementation Hints 32
Appendix C: Bibliography 33

Introduction

This document describes OpenCores RV01 processor core available at http://opencores.org/project,RV01_core. The RV01 core implements RISC-V RV32I instruction set with "M" extension according to RISC-V ISA version V2.0 [1], and implements (with some exception) privileged architecture according to version V1.7 [2].

Hopefully this work can be of use to somebody.

License

In accordance with the existing version of the OpenCores RV01 processor core. This work is licensed under the GNU Lesser General Public License. The license can be obtained at http://www.gnu.org/licenses/lgpl.html . As such, the following applies to all source files added to the OpenCores RV01 processor core.

Conventions

The following conventions are used in this document:

- Core: the RV01 processor core described in this document.
- User logic: any circuitry connected to the core, and used to control it and/or exchanging data with it.
- RAS (Return Address Stack): the data structure that stores function call returns addresses, used by `jalr` instructions prediction logic.
- DMA (Direct Memory Access): DMA port is the main interface between the core and user logic, allowing user logic to read/write the core internal memories.
- Run state: the functional condition in which instructions execution is ongoing.
- Halt state: the functional condition in which instructions execution is suspended.
- Instruction #0: when parallel instruction execution is enabled, instruction #0 is the older of the two instructions in a given stage of the core CPU pipeline.
- Instruction #1: when parallel instruction execution is enabled, instruction #1 is the newer of the two instructions in a given stage of the core CPU pipeline.

Main Features

- Two-way in-order superscalar design (optionally configurable as scalar-only to reduce resource usage). The core can fetch, execute and complete up to two instructions per cycle.
- Implements RISC-V RV32I instruction set with “M” extension according to RISC-V ISA version 2.0 [1].
- Implements RISCV privileged architecture version 1.7 [2] (Mbare mode only), with some exception described later (most notably the use of physically separated instruction and data memories).
- Dual 7-stage asymmetrical pipeline: one pipeline can execute all instructions, while the other one can execute only a subset of instructions chosen between most frequently executed ones.
- Branch and Jump-PC (`jal`, but not `jalr`) instructions are predicted using 2-bit saturating counters. Prediction logic may be optionally removed to reduce resource usage.
- Jump-Register (`jalr)` instructions are predicted using a return address stack (RAS). Prediction logic may be optionally removed to reduce resource usage.
- Instructions and data are held in physically separated memory (of user-configurable size) in order to allow concurrent access to them from RV01 CPU. Instructions and data are accessible to user logic through a DMA-like port.
- Single-threaded only.
- No hardware support for misaligned load and store operations.
- No support for cache memories or virtual memory (MMU).
- Optional debug module, implemented according to spec. proposal version 0.9 [4].
- Optional PLIC module, implemented according to RISC-V ISA version 2.1 [3].
- “Full optional” version delivers > 1.7 DMips/Mhz.

Source code language

All source code files are written in synthesizable VHDL language.

Core architecture

General description

The core consists of:

- CPU module.
- Instruction and data memories.
- Interface logic managing the data transfers between the core and user logic.
- PLIC module (optional).

Data transfer between the core and user logic is implemented through the DMA port. The core must be in halt state (instruction execution is suspended) while DMA operations are in progress.

Instruction and data memories are physically separated and implemented using embedded (on-chip) dual port RAM blocks with a read/write port and a read-only port. For instruction memory, both ports are 64-bit wide, read-only port is used only for instruction fetching, while read-write one provides access to instruction memory as data memory (i.e. through regular load/instructions). For data memory, both ports are 32-bit wide and are used for load/store operations. Data memory can't be used for instruction fetching.

Two additional interfaces are available to exchange data between the core and user logic:

- Control port provides user logic access to CPU CSR's when the core is in Halt/Debug state.
- `MFROMHOST_i` input (along with the associated write-enable signal) and `MTOHOST_o` output (along with the associated output-enable signal) provides user logic access to `MTOHOST` and `MFROMHOST` CSR's even when the core in Run state.

The core can optionally include a PLIC module implemented according to RISC-V ISA version V2.1.

A simplified block diagram of the core (PLIC and Debug/Halt modules are not shown) is shown in Figure 1.

Block diagram

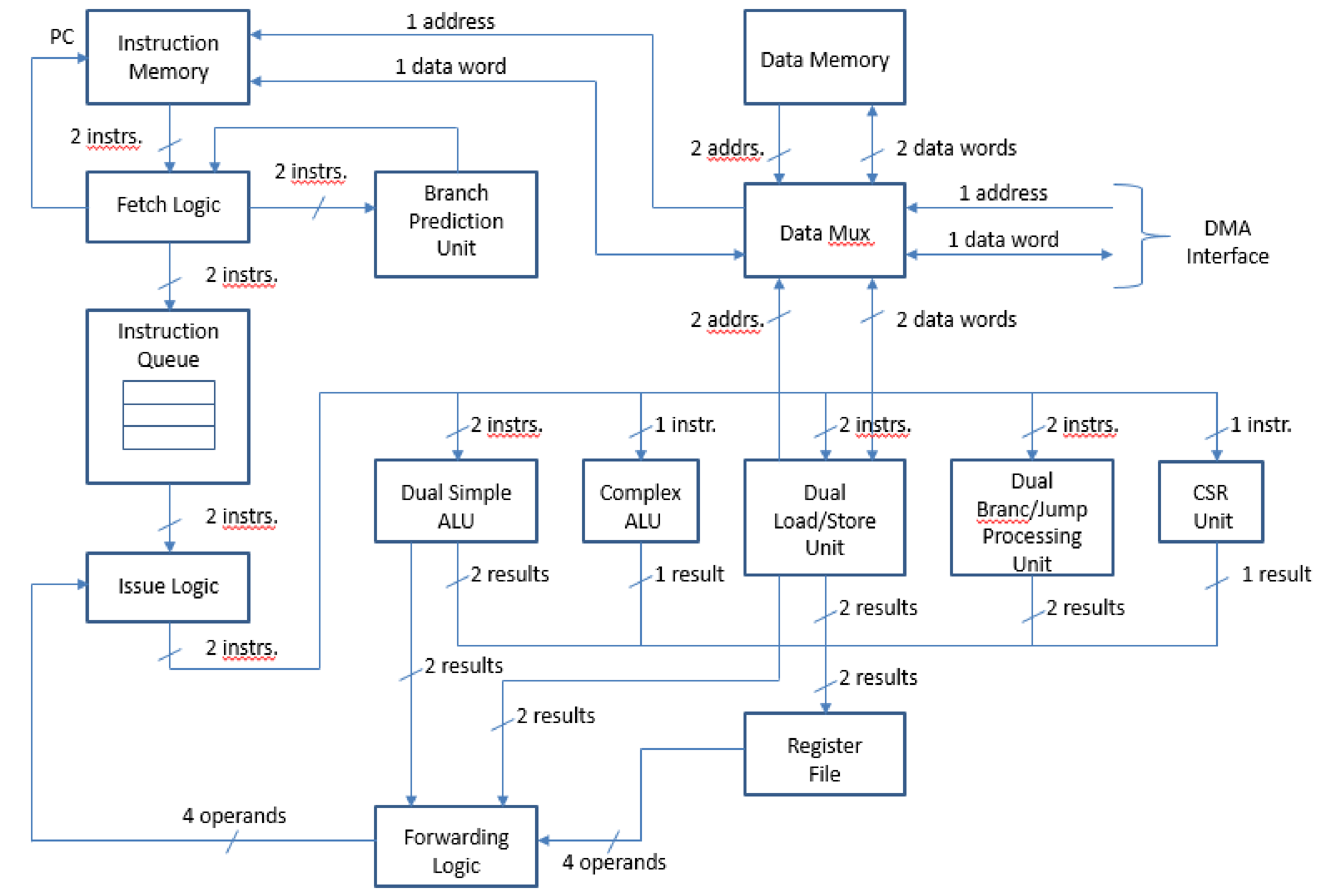

Figure 1: RV01 core block diagram (simplified).

Pipeline Organization

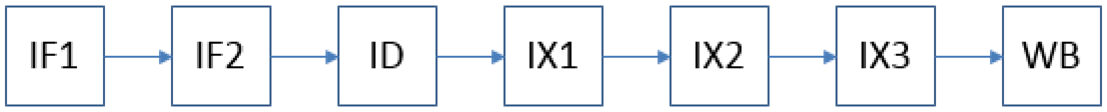

Figure 2: RV01 CPU pipeline organization.

The CPU module pipeline (see Figure 2) consists of seven stages:

- IF1: instruction fetching address generation.
- IF2: instruction fetching, branch and jump instructions prediction, instruction pre-decoding.
- ID: instruction decoding and issuing, operands fetching.
- IX1: Simple/Complex ALU instructions execution, load/store address generation, branch and jump processing and/or prediction verification (first part).
- IX2: Complex ALU instructions execution, load data fetching (LW instruction results made available for forwarding). Branch and jump processing and/or prediction verification (second part).
- IX3: half-word and byte load instructions result alignment, exceptions raising.
- WB: Instruction results write-back to architectural register file.

Fetch pipeline consists of stages IF1-2, while execution pipeline consists of stages IX1-3. Stage ID includes a three-instruction queue that provides some degree of decoupling between fetch and execute pipelines.
Fetch pipeline stalls in a given cycle if the number of empty entries in the ID stage instruction queue, minus the number of instructions released (to be executed) from the queue in that cycle is lower than two.
The two oldest instructions in ID stage instruction queue are the ones checked for issuing, such instructions remain in the queue until the conditions to execute them are satisfied (the required operands are available from execution pipeline and any resource conflict is resolved). Because of in-order issue constraint, the newer instruction in the pair (named instruction #1) can be issues only if the older one (named instruction #0) is issued too.
The execute pipelines never stalls, in the sense that, once an instruction is released from the ID stage instruction queue, it never stops in the execution pipeline, always reaching WB stage in three cycles. When no instruction can be issued from ID stage queue in a given cycle, a pair of null instructions is fed to the execute pipeline.
Instruction results can be forwarded from stages IX1-3 to ID stage, but only high frequency instruction results (generated by simple ALU) can be forwarded, in order to reduce multiplexing logic complexity.
All instructions complete execution in a maximum of three cycles, except for the division/reminder ones, which takes a variable amount of cycles (depending from the operands relative magnitude). Such instructions are handled using a mechanism called *re-fetching*:

1. When a division/reminder instruction reaches stage IX1 for the first time, the divider is started and the instruction is marked for re-fetching.
2. When an instruction marked for re-fetching reaches stage IX3, it is re-fetched (the net effect is equivalent to a jump to the instruction address). The instruction is not allowed to complete (divider result is usually not available at this point).
3. When the re-fetched instruction reaches stage ID, it is held in the instruction queue until the divider completes the operation in progress and then it is released (this time being not marked for re-fetching).
4. When the re-fetched instruction reaches stage IX2, the result is “picked-up” from divider and allowed to reach stage WB, completing its execution by writing the result to the target GPR.

The re-fetch mechanism is also used in every circumstance where an instruction that has already been released from the instruction queue can’t complete regularly. This circumstance occurs, for instance, when a load instruction targets a memory location affected by a store which is held in the store buffer: the load instruction is re-fetched, giving time to store instruction to complete.
The re-fetch mechanism is, however, not used when two load instructions accessing memory are released in the same cycle: the resulting conflict (only one read access on instruction memory per cycle is permitted) can be detected only when the effective address is calculated in stage IX1, but is handled by a conventional one-cycle pipeline slip.
The re-fetch mechanism, basically, removes the need to handle multi-cycle stalls in the execution pipeline, simplifying its design.

Core interface

The core interface (with reference to `RV01_TOP` module, from file `RV01_top.vhd`) consists of the following signals:

- `CLK_i`: clock input.
- `RST_i`: synchronous reset input (reset completes in one cycle).
- `CHK_ENB_i`: debug input.

- `EI_REQ_i[EI_SRC_CNT-1:0]`: external interrupt request input.
- `MFROMHOST_WE_i`: `MFROMHOST` CSR write-enable input.
- `MFROMHOST_i`: `MFROMHOST` CSR data input.
- `DP_WE_i`: DMA port write-enable input.
- `DP_ADR_i[32-1:0]`: DMA port address input.
- `DP_DAT_i[32-1:0]`: DMA port data input.
- `CP_RE_i`: halt/debug port read-enable input.
- `CP_WE_i`: halt/debug port write-enable input.
- `CP_ADR_i[17-1:0]`: halt/debug port address input.
- `CP_DAT_i[32-1:0]`: halt/debug port data input.
- `MTOHOST_OE_o`: `MTOMHOST` CSR output-enable (read-ready) input.
- `MTOHOST_o`: `MTOHOST` CSR data output.
- `DP_DO_o[32-1:0]`: DMA port data output
- `CP_DO_o[32-1:0]`: halt/debug port data output

Table 1: Core interface signals description.

Signal	Description
`CLK_i`	Core single clock input, all core operations (including DMA ones) are clocked by this signal.
`RST_i`	Synchronous resets input. It must be asserted by user logic for >=1 cycle(s)
`CHK_ENB_i`	Verification purpose only, must always be left to inactive level (‘0’).
`EI_REQ_i[EI_SRC_CNT-1:0]`	External interrupt request input, if the PLIC core is present and properly configured, asserting these inputs triggers an external interrupt exception. The number of request lines (`EI_SRC_CNT`) is user-configurable.
`MFROMHOST_WE_i`	`MFROMHOST` CSR write-enable input, a data word loaded by user logic on `MFROMHOST_i` is considered valid, <u>on the same cycle</u> where `MFROMHOST_WE_i` signal is asserted.
`MFROMHOST_i[32-1:0]`	`MFROMHOST` CSR data input.
`DP_WE_i`	DMA port write-enable input. User logic asserts `DP_WE_i` for one cycle when a data word needs to be written to the core internal memory.
`DP_ADR_i[32-1:0]`	DMA port address input, An address loaded by user logic on `DP_ADR_i` is considered valid, <u>on the same cycle</u> where `DP_WE_i` signal is asserted
`DP_DAT_i[32-1:0]`	DMA port data input, a data word loaded by user logic on `DP_DAT_i` is considered valid, <u>on the same cycle</u> where `DP_WE_i` signal is asserted.
`CP_RE_i`	Debug/Halt port read-enable signal. User logic asserts `CP_RE_i` for one cycle when a data word needs to be read from the core CSR’s
`CP_WE_i`	Debug/Halt port write-enable signal. User logic asserts `CP_WE_i` for one cycle when a data word needs to be written to the core

	CSR's
CP_ADR_i[17-1:0]	Halt/Debug port address input. An address loaded by user logic on CP_ADR_i is considered valid, on the same cycle where CP_WE_i signal is asserted
CP_DAT_i[32-1:0]	Halt/Debug port data-in input. A data word loaded by user logic on CP_DAT _i is considered valid, on the same cycle where CP_WE_i signal is asserted
MTOHOST_OE_o	MTOHOST CSR output-enable output, it's asserted by the core for one cycle, to flag that a valid data word has been loaded by the core on MTOHOST_o
MTOHOST_o	MTOHOST CSR data output.
DP_DO_o[32-1:0]	DMA port data output. A data word loaded by the core on DP_DAT_o is valid one cycle after the corresponding address has been loaded on DP_ADR_i.
CP_DO_o[32-1:0]	Halt/Debug port data output. A data word loaded by the core on CP_DAT_o is valid one cycle after the corresponding address has been loaded on CP_ADR_i

All control signals are active-high: asserting a signal means driving it high, and de-asserting a signal means driving it low.

Core top-level module is named RV01_TOP and is located in source file RV01_top.vhd.

The core provides the following configuration parameters (via VHDL generics):

- ST_FILE, WB_FILE: used for verification purposes only. Please ignore them.
- IMEM_SIZE: instruction memory size (in 32-bit words).
- DMEM_SIZE: data memory size (in 32-bit words).
- IOMEM_SIZE: I/O memory size (in 32-bit words, must be a power-of-2 value).
- IMEM_SIZE_PO2: instruction memory size "power-of'2" flag, if this parameter is set to '1' when IMEM_SIZE is a power-of-2, a simpler (and hopefully faster) circuitry is used to check if instruction memory address is within valid bounds. Setting this parameter to '1' when IMEM_SIZE is NOT a power-of-2 has un-predictable results.
- DMEM_SIZE_PO2: data memory size "power-of'2" flag, if this parameter is set to '1' when DMEM_SIZE is a power-of-2, a simpler (and hopefully faster) circuitry is used to check if data memory address is within valid bounds. Setting this parameter to '1' when DMEM_SIZE is NOT a power-of-2 has un-predictable results.
- IMEM_LOWM: instruction memory "low-memory" flag, if this parameter is set to '1', instruction memory address space is located in lower portion of the global address space. Leave this parameter always set to default value of '1'.
- BHT_SIZE: Branch History Table size (number of entries), this parameter specifies the size of the BHT (only when branch prediction is implemented in the core).
- EI_SRC_CNT: external interrupt source count, this parameter specified the number of external interrupt request lines (only when PLIC module is present).

- `EI_TRIG_TYPE`: external interrupt trigger type, this parameter selects the trigger type (`LEVEL` or `EDGE`) of the external interrupt sources (only when PLIC module is present).
- `EI_REQ_MAXCNT`: external interrupt request maximum count, this parameter specifies the maximum number of pending interrupts when source trigger type is set to `EDGE` (only when PLIC module is present).
- `CFG_FLAGS[16-1:0]`: configuration flags (see Table 2). These flags enable specific core functionalities (like, for instance, parallel execution), allowing user to balance performance and hardware resource usage.
- `SIMULATION_ONLY`: used for verification purposes only. Leave this parameter always set to default value of '0'.

Table 2: Configuration flags.

Bit Index	Mnemonic	Purpose
0	PARALLEL_EXECUTION_ENABLED	1→ 2-way superscalar CPU (0 → scalar CPU)
1	DELAYED EXECUTION ENABLED	1→delayed execution enabled
2	BRANCH PREDICTION ENABLED	1→branch (and `jal`) instruction prediction enabled.
3	JALR PREDICTION ENABLED	1→ `jalr` instruction prediction enabled
4	FPU_PRESENT	1→Floating Point Unit present.
5	DM_PRESENT	1→ Debug Module present (0 → Halt Module present)
6	PLIC_PRESENT	1→PLIC module present.
7:15	N.A.	Reserved for future use, leave to '0'.

Memory

Instruction and data memories

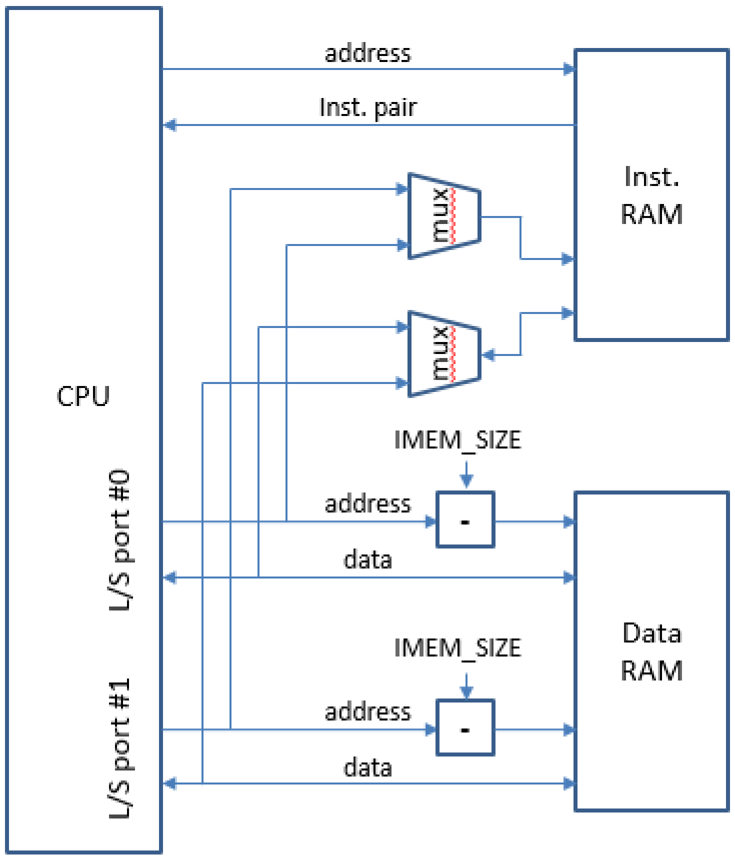

Figure 3: core memory architecture

Instruction and data memories are implemented with independent dual-port RAM's (see Figure 3).

Instruction RAM has a 64-bit read-only port used for instruction fetching and a 64-bit read/write port used to access it through load/store instructions (this port is shared between the two CPU load/store ports so that only one load/store access per cycle is allowed).

Data RAM has a 32-bit read/write port and a 32-bit read-only port, both used to access it through load/store instructions. Being one port read-only, only one write access per cycle is allowed.

A simple address translation is performed on load/store addresses mapped to data RAM, allowing load/store instructions to access the instruction and data RAM's as a single memory block. As a consequence, load/store instructions can be used to read and write any memory location, but instruction fetching is restricted to instruction RAM (i.e. instructions can't be fetched from data RAM).

DMA port addresses are subject to the same address translation used for data ones.

Current core memory design requires instruction RAM to be mapped on the lower portion of data address space (configuration parameter `IMEM_LOWM` can only be set to '1'). Future core releases may support mapping of instruction RAM on higher portion of data address space.

Memory design supports up to two load or store operations per cycle, with the following restrictions:

- Only one store per cycle is permitted (one of the data RAM port is read-only).

- Only one operation per cycle targeting instruction memory is permitted.

Instruction fetching and load/store operations can run in parallel as the instruction RAM read-only port is reserved for instruction fetching.

Address space

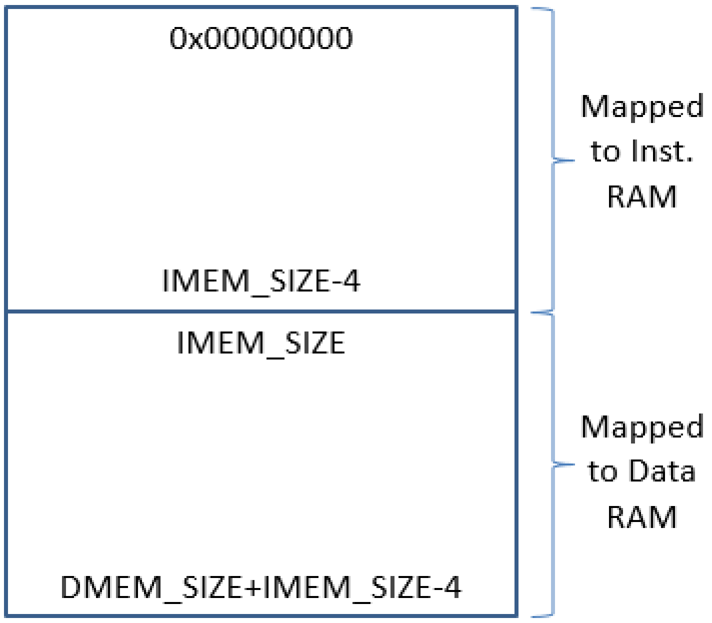

Figure 4: core address space.

The core address space (see Figure 4), as seen by the user, consists of a continuous range of DMEM_SIZE + IMEM_SIZE words starting from address 0x00000000.

In current core version, lower portion of valid address space is always mapped to instruction RAM, while higher portion is always mapped to data RAM.

I/O memory, when present, is always located above data memory (e.g. starting from address DMEM_SIZE+IMEM_SIZE, see Figure 5) and is treated as an extension of it. User must take care of avoiding concurrent load/store operations on I/O memory locations if the corresponding I/O module doesn't support them (parallel instruction execution can be temporarily disabled clearing `MRV01CC[PXE]` bit, see "Control and Status Registers" section under "Privileged architecture implementation" chapter for details).

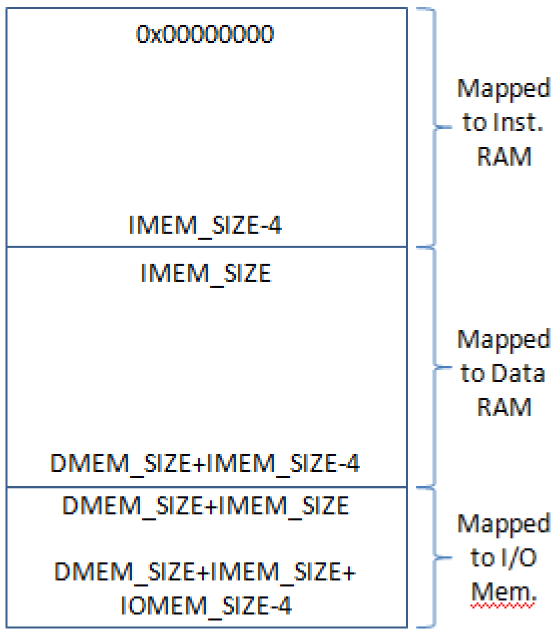

Figure 5: core address space with I/O memory.

Attempt to fetch an instruction from an address mapped to data RAM results in an instruction access fault exception.

Attempt to access any location outside of valid address space (including I/O memory) results in an instruction, or data, access fault exception (from this point of view, I/O memory is treated as data one).

Attempt to access any location belonging to I/O memory, but not mapped to an I/O module, can results in reading of garbage data or trashing of write data. No exception is, however, raised.

Privileged architecture implementation

General notes

The core implements privileged architecture version 1.7 [3] with the following restrictions:

- Only Mbare addressing environment is supported.
- Only Machine and User modes are supported.
- Due to the constraint that instruction RAM must be mapped on lower portion of data address space, `MTVEC` CSR can be set only to trap vector lower location (`0x0000100`)
- The core is strictly single-threaded.
- `MRTS`, `MRTH`, `HRTS`, `WFI` and `SFENCE.VM` instructions are implemented as NOP's.

Control and Status Registers

CSR access from control port

Control port uses a 17-bit address, rather than the standard CSR 12-bit one, to comply with Debug module specs (see [4]), even when Debug module is replaced by Halt module.
A CSR's 12-bit address (not related to the Debug module) can be translated to a control port (CP) 17-bit address by

- Mapping the 12-bit address to CP address bits [13:2].

- Setting CP address bits [1:0] to “00”.
- Setting CP address bits [16:14] to “100”.

So, for instance, `MTVEC` CSR (address 0x301) is accessed from control port using 17-bit address 0x10c04 (100_0011_0000_0001_00)
When Halt module is not present, CP address bits [16:14] and bits [1:0] can be hard-wired to “100” and “00”, respectively.

Table 3: non-optional CSR's.

Addr.	Mnemonic	Notes
0xc00	`UCYCLE`	Count cycles (lower 32-bit)
0xc01	`UTIME`	Count cycles (lower 32-bit)
0xc02	`UINSTRET`	Count retired instructions (lower 32-bit)
0xc80	`UCYCLEH`	Count cycles (upper 32-bit)
0xc81	`UTIMEH`	Count cycles (upper 32-bit)
0xc82	`UINSTRETH`	Count retired instructions (upper 32-bit)
0x0c0	`USTATS`	Can be read/written, but access has no side effect
0xf00	`MCPUID`	Hard-wired to `0x00001000`
0xf01	`MIMPID`	Hard-wired to `0x00008000`
0xf10	`MHARTID`	Hard-wired to `0x00000000`
0x300	`MSTATUS`	Resets to `0x0000024e` (FS, XS and DS fields hard-wired to 0)
0x301	`MTVEC`	Resets to `0x00000100` (trap vector lower location)
0x302	`MTDELEG`	Hard-wired to `0x00000000`
0x304	`MIE`	Resets to `0x00000000`, only bit #7 is writable
0x321	`MTIMECMP`	
0x701	`MTIME`	Count cycles (lower 32-bit)
0x741	`MTIMEH`	Count cycles (upper 32-bit)
0x340	`MSCRATCH`	
0x341	`MEPC`	
0x342	`MCAUSE`	
0x343	`MBADADDR`	
0x344	`MIP`	
0x380	`MBASE`	Can be read/written, but access has no side effect
0x381	`MBOUND`	Can be read/written, but access has no side effect
0x382	`MIBASE`	Can be read/written, but access has no side effect
0x383	`MIBOUND`	Can be read/written, but access has no side effect
0x384	`MDBASE`	Can be read/written, but access has no side effect
0x385	`MDBOUND`	Can be read/written, but access has no side effect
0x780	`MTOHOST`	
0x781	`MFROMHOST`	
0x782	`MRV01CC`	RV01 core control (implementation-specific) register

Table 4: FPU CSR's (optional).

Addr.	Mnemonic	Notes
0x001	`FFLAGS`	Floating point Flags register
0x002	`FRM`	Floating point Rounding Mode register
0x003	`FCSR`	Floating point Control and Status register

Table 5: Debug module CSR's (optional, implementation-specific).

Addr.	Mnemonic	Notes
0x0000	`CCS`	Component Control and Status register
0x0010	`DTMIA`	DTM Interrupt Address register
0x0020	`DCS`	Debug Control and Status register
0x0030	`PCS`	PC Sample Register
0x0100	`SI`	Stuff Instruction Register
0x0110	`DJ`	Debug Jump register
0x0120	`PC`	PC register

Table 6: Halt module CSR's (optional, implementation-specific).

Addr.	Mnemonic	Notes
0x783	`MRV01HC`	RV01 halt control register
0x784	`MRV01HA`	RV01 halt address register
0x785	`MRV01RA`	RV01 resume address register

Notes:

1. Debug module CSR addresses are 17-bit long (rather than 12-bit long like the other CSR's), these CRS's are accessible only through the Debug/Halt port (which purposely has a 17-bit address input).
2. Halt module CSR's can be accessed like any other CSR, either using RISC-V ISA CSR's manipulating instructions or directly through the Debug/Halt port.
3. Either debug module CSR's or halt module CSR's must always be present, as these registers sets the fetch starting address and the condition controlling the transition from Run state to Debug/Halt state.

Non-optional, implementation-specific, CSR's description

Table 7: `MRV01CC` CSR bit field description

Bit(s)	Mnemonic	R/W	Description
0:31	`PXE`	R/W	Parallel Execution Enabled flag (1 → parallel execution enabled, 0 → parallel execution disabled). Clearing this bit at the beginning of a code section and setting at its end insures all instructions inside the code section are executed in strict sequential fashion (no two instructions are executed in parallel).
1:31	n.a.		Reserved for future use

Debug Module CSR's description

Please make reference to spec. proposal version 0.9 [4].

Halt Module CSR's description

Table 8: `MRV01HC` CSR bit field description

Bit(s)	Mnemonic	R/W	Description
0	`HALTSTATE`	R	Halt state flag (1 → Halt state, 0 → Run state)
1	`START`	W	Start flag, writing it to '1' causes a transition from Halt to Run state and starts instruction execution from ISA reset address.

2	RESUME	W	Resume flag, writing it and START bit to '1' causes a transition from Halt to Run state and starts instruction execution from address stored in MRV01RA register.
3	HALTU	W	Halt Unconditionally flag, writing it to '1' causes an immediate transition from Run to Halt state and stops instruction execution.
4	HALTOBRK	R/W	Halt-on-Break flag, when set to '1', execution of a sbreak instruction causes an immediate transition from Run to Halt state and stops instruction execution.
5	HALTOADR	R/W	Halt-on-Break flag, when set to '1', execution of instruction located at address stored in MRV01HA register causes an immediate transition from Run to Halt state and stops instruction execution.
6:31	n.a.		Reserved for future use

Table 9: MRV01HA CSR bit field description

Bit(s)	R/W	Description
0:31	R/W	This register stores the address where instruction execution halts when halt-on-address mode is enabled. The instruction at this address is not executed.

Table 10: MRV01RA CSR bit field description

Bit(s)	R/W	Description
0:31	R/W	This register stores the address where instruction execution resumes when '1' is written to MRV01HC[START] and MRV01HC[RESUME] bits. This register allows instruction execution to be resumed from an arbitrary address. When instructions execution halts, this register stores the address of the last not executed instruction (i.e. MRV01RA content is coincident with MRV01HA one). If MRV01RA and MRV01HA registers hold the same address when instruction execution resumes and MRV01HC[HALTOADR]is set to '1', the resume command takes precedence over the halt-on-address one, in this way, inside a loop, instruction execution can be halted/resumed at the same instruction.

Core configuration options summary

General notes

The core design enables users to re-size or remove several features in order to achieve the desired tradeoff between performance and resource usage. All configuration options can be managed through VHDL generics available on top module RV01_TOP, allowing different core instances to be configured independently.

Size options

Instruction, data and I/O memories

The size of instruction and data memories (expressed in 32-bit words) can be set using generics IMEM_SIZE and DMEM_SIZE. The size of instruction memory can be set independently of data one.

Size values don’t need to be a power-of two, but if instruction/data memory size is actually a power-of-two, setting generic IMEM_SIZE_PO2/IMEM_SIZE_PO2 to ‘1’ permits to simplify circuitry performing address range checking.
When generic PARALLEL_EXECUTION_ENABLED is set to ‘1’, there’s the additional constraint that instruction memory size must be a multiple of 64-bit.
Because of the pipeline organization, instruction and data memories output directly feds the CPU internal logic, without a register to break the timing path, and therefore paths originating from instruction and data memory outputs tends to become more timing critical as their size increases (said in different words: larger instruction/data memory size tends to result in lower core max operating frequency).
The size of I/O memory can be set using generic IOMEM_SIZE and must be a power-of two. I/O modules (like PLIC one) can be present only if IOMEM_SIZE > 0 (the result of an attempt of accessing an I/O module memory-mapped register when IOMEM_SIZE = 0 is unpredictable).

Branch History Table (BHT)

The size of the branch history table included in branch and jal instructions prediction logic can be set using generic BHT_SIZE. This generic is meaningful only if BRANCH_PREDICTION_ENABLED generic is set to ‘1’.
The table is designed as a synchronous RAM module and can therefore be implemented using embedded RAM blocks.
When generic PARALLEL_EXECUTION_ENABLED is set to ‘1’, the table is physically split in two RAM blocks, in order to support prediction on two instructions per cycle.

Functional Options

Parallel instruction execution

Parallel instructions execution is enabled by setting generic PARALLEL_EXECUTION_ENABLED to ‘1’. When this option is enabled up to two instructions can be fetched and executed in each cycle, in strict program order (in-order execution constraint). From a design standpoint, the execution pipeline contains two or three “physical” pipelines: one, or two, simple instructions pipelines (which can executed only a limited set of instructions that occurs with high frequency in typical code) and one complex instruction pipeline (which can execute all remaining instructions). Logical pipeline #0 consists of a simple instructions pipeline plus a complex instructions pipeline, while logical pipeline #1 (which is present only when parallel instructions execution is enabled) consists of a simple instruction pipeline only. In the pair of instructions candidate to be executed in parallel, the instruction #0 (the older one) is always assigned to pipeline #0, while instruction #1 (the newer one) is always assigned to pipeline #1.
The set of instructions executed by the simple instructions pipeline includes: add, addi, sll, slli, srl, srli, sra, srai, and, andi, or, ori, xor, xori, plus load, stores, branches and jumps. All these instructions execute in one cycle, except for lw, which executes in two cycles and lh*/lb*, which execute in three cycles.
In general two instructions can be executed if parallel if instruction #1 belongs to the simple instructions set (instruction #0 can be any instructions) and there’s no data dependency between them (if instruction #1 takes instruction #0 result as operand, instruction #1 can’t be released together with instruction #0).
Stores can be treated as executing in parallel, in spite of the fact memory architecture doesn’t support two writes in the same cycle, thanks to a store buffer where stores are held until it is known if any previous instruction hasn’t raised an exception or requires being re-fetched. Being the store buffer

capable to accept two store instructions per cycle, stores can be released in parallel from the instruction queue until the buffer doesn't fills up.
When instruction #0 is an instruction manipulating CSR's (and therefore potentially impacting the way following instructions are executed), instruction #1 is not issued, e no other instruction is issued until the instruction manipulating CSR's complete its execution, this guarantees that any newer instruction "sees" the updated CSR's content.
When parallel instruction execution is enabled, sequential instructions execution can be enforced by clearing `MRV01CC[PXE]` bit.

Delayed Instructions Execution

Delayed instructions execution is enabled by setting generic `DELAYED_EXECUTION_ENABLED` to '1'. When this option is enabled, instructions belonging to a subset of high frequency instructions that execute in one cycle are allowed to be issued from the instructions queue even if one, at least, of their operand is not yet available. Such instructions are actually executed when the missing operand is produced by an older instruction (this is possible because the simple ALU required to execute these instruction is replicated in all stages of the execution pipeline).
This option doesn't break the in-order execution paradigm because it still guarantee than no instruction can be issue/executed/completed before an older one.

Branch and Jump-PC instructions prediction

Branch and Jump-PC (`jal`) instructions prediction is enabled by setting generic `BRANCH_PREDICTION_ENABLED` to '1'. Prediction is based on 2-bit saturating counters stored in a Branch History Table (BHT) along with the predicted target address. This simple prediction mechanism is used to predict `jal` instructions too.
Prediction takes place when the instruction is in IF2 stage, while prediction verification occurs in IX2 stage, leading to a mis-prediction penalty of three cycles (the penalty for a correct prediction is, instead, zero cycles).
If parallel instructions execution is enabled, up to two branch or `jal` instructions can be predicted, or have their prediction checked, in each cycle.

Jump-register instructions prediction

Jump-register (`jalr`) instructions prediction is enabled by setting generic `JALR_PREDICTION_ENABLED` to '1'. Prediction is based on a 4-entry Return Address Stack (RAS), supported by a 2-entry verification queue (which store prediction verification info while the `jalr` instruction moves along the pipeline).
Prediction takes place when the instruction is in IF2 stage, while prediction verification occurs in IX2 stage, leading to a mis-prediction penalty of three cycles (the penalty for a correct prediction is, instead, zero cycles).
If parallel instructions execution is enabled, both instructions in IF2 stages are checked for `jalr`, but, if both are jumps only the older instruction (e.g. the one in pipeline #0) is predicted, the newer one being treated as it's mis-predicted, resulting in a jump penalty of three cycles .
It's worth noting that only `jalr` instructions presumably used to return from a function call (e.g. those writing GPR r0 and having zero immediate) are predicted, the other `jalr` instructions are ignored when in stage IF2 and executed in stage IX2 (in other words they're treated as they're always mis-predicted, resulting in a constant jump penalty of three cycles).

FPU

Current version of the core doesn't support a Floating Point Unit, so `FPU_PRESENT` generic must be left set to '0'.

Debug module

A Debug Module based on spec. proposal version 0.9 (see [4]) can be included in the core by setting generic `DM_PRESENT` to '1', the Debug Module being replaced by the simpler Halt Module when the generic is, instead, set to '0' (in this way one of the two modules is always present but the two are never present at the same time).
The available Debug module implements only the portion of the debug interface covered by chapter #6 ("RISC-V Debug Module") of the specs document, except for authentication functionalities (not available), freeze mode (not available) and the capability to access GPR's as CSR's (not available).
WARNING: being the Debug module based on a proposed spec (never officially approved), it must be treated as an "experimental" option.

Halt module

A Halt Module can be included in the core by setting generic `DM_PRESENT` to '0', the Halt Module being replaced by the more complex Debug Module when the generic is, instead, set to '1' (in this way one of the two modules is always present but the two are never present at the same time)
The Halt module, when present, manages the transitions between Halt and Run states, allowing user to start/restart instruction execution from the ISA reset address or from any arbitrary address, and to stop instruction execution:

- Unconditionally (i.e. on the first instruction entering the execution pipeline after `MRV01HC`[`HALTU`]) is set to '1') , or
- When the instruction at a target address (specified by `MRV01HA` CSR) is fetched, or
- When a `sbreak` instruction is fetched.

Halt module functionalities are controlled by CSR's described in Tables 8-10.
Although user logic can write Halt module CSR's in any moment, it's strongly suggested to not do it while core is in Run state, except for unconditionally halting instruction execution by setting `MRV01CC[HALTU]` bit to '1' (after all, there's little value in doing so while the core is already in Halt state).
When the core is in Halt state, user logic can access the core instruction and data memories, loading a new program or reading/writing program data.

PLIC

A Programmable Local Interrupt Controller (PLIC) module implemented according to RISC-V ISA version V2.1 [2] is included in the core by setting generic `PLIC_PRESENT` to '1'. The PLIC module is placed (when present) inside the top-level module RV01_TOP and is managed through memory-mapped registers (located in I/O memory, which size must be > 0) accessible using load/store instructions.
The PLIC module interacts with user logic via the core `EI_REQ_i` input(s), if the PLIC memory-mapped registers are properly set, when one of this inputs get asserted, an external interrupt exception is raised in the core CPU module, which manages the interrupt sources by reading/writing the PLIC registers by mean of load/store instructions (usually part of the interrupt service routine).
In order to support the PLIC module, the core uses the slightly modified machine-level exception codes table reported below here.

Table 11: machine-level exception codes (interrupts only)

Interrupt	Exception Code	Description
1	0x0	Software Interrupt
1	0x1	Timer Interrupt
1	**0x2**	**External Interrupt**
1	>0x2	Reserved

In current core version PLIC memory-mapped register are located at address `PLIC_ABASE`, specified as a constant in core top-level module VHDL file RV01_TOP.vhd.

Table 12: PLIC module address space

Address (32-bit word offset from PLIC_ABASE)	Mnemonic	Description
`0:EI_SRC_CNT-1`	`ISR[0:EI_SRC_CNT-1]`	Interrupt Source Register(s)
`EI_SRC_CNT`	`IPTR`	Interrupt Priority Threshold Register
`EI_SRC_CNT+1`	`IEBR`	Interrupt Enable Bit Register
`EI_SRC_CNT+2`	`ICR`	Interrupt Control Register

Table 13: ISR bit field description

Bit(s)	R/W	Description
0:7	R/W	Interrupt Id.
8:15	R/W	Interrupt priority
16:31	n.a.	Reserved for future use

Table 13: IPTR bit field description

Bit(s)	R/W	Description
0:7	n.a.	Reserved for future use
8:15	R/W	Priority threshold
16:31	n.a.	Reserved for future use

Table 14: IEBR bit field description

Bit(s)	R/W	Description
0:31	R/W	Source individual enable bits (only bits `0:EI_SRC_CNT-1` are meaningful)

Table 15: ICR bit field description

Bit(s)	R/W	Description
0:7	R	Interrupt Id.
8	W	Interrupt Claim
9	W	Interrupt Complete
10:31	n.a.	Reserved for future use

Flow of operations

Using Debug module

After reset signals `RST_i` has been de-asserted, the core is in Debug state (a condition similar to Halt state), as flagged by `CCS[HALTED]` bit value of ‘1’ (no instruction execution is in progress) and

therefore user logic can safely writes programs or data to the core internal memories through the DMA port.
While the core is in Debug state, user logic can also access the Debug module CSR's and set the program starting address and/or the program execution halting condition by writing `CCS` and `DPC` CSR's
When the DMA operations and CSR's setup are complete, user logic can start program execution from address stored in `DPC` register by setting `CCS[RESUME]` bit to '1'.
The core remains in Run state until the halting condition set in the Debug module CSR's get satisfied, or forever, if no halting condition has been specified (in such case the core can still be halted setting `CCS[HALT]` bit to '1').
When the core returns to Debug state, user logic can:

- Read program output data.
- Write new program input data.
- Restart or resume program execution.
- Load a new program and the related data…

The Debug module supports many other functionalities, beside the minimal ones used here, the curious reader may find them described in greater detail in the spec. proposal V0.9 [4].

Using Halt module

After reset signals `RST_i` has been de-asserted, the core is in Halt state, as flagged by `MRV01HC[HALTSTATE]` bit value of '1' (no instruction execution is in progress) and therefore user logic can safely writes programs and data to the core internal memories through the DMA port.
While the core is in Halt state, user logic can also access the Halt module CSR's and set the program starting address and/or the program execution halting condition by writing `MRV01CC` and `MRV01HA` CSR's.
When the DMA operations and CSR's setup are complete, user logic can start program execution from ISA reset address by setting `MRV01HC[START]` bit to '1'. The core responds to this write operation by entering Run state and setting `MRV01HC[HALTSTATE]` bit to '0' (user logic can monitor this bit to check the core state). If `MRV01HC[RESUME]` bit is set to '1' at the same time of `MRV01HC[START]`, program execution starts from the address stored in `MRV01RA` CSR.
The core remains in Run state until the halting condition set in the Halt module CSR's get satisfied, or forever, if no halting condition has been specified (in such case the core can still be halted setting `MRV01HC[HALTU]` bit to '1').
When the core returns to Halt state, user logic can:

- Read the program output data.
- Write new program input data.
- Restart or resume program execution.
- Load a new program and the related data…

Performance

Dhrystone benchmark

The "full optional" version of the core (e.g. with parallel execution, delayed execution, branch/`jal` prediction and `jalr` prediction all enabled, and a 512-entry BHT) delivers a Dhrystone score of ~1.72 Dmips/MHz, using the executable file dhrystone.riscv.exe compiled for Sodor processor (the only changes applied to this executable file were needed to use hardware multiplication and division in place of software ones).

The “bare bone” version of the core (e.g. without parallel execution, delayed execution, branch/`jal` prediction and `jalr` prediction) delivers a Dhrystone score of ~1.09 Dmips/MHz (-57%), using the same executable file of above.

To the purpose of trading performance for resource usage, delayed execution and parallel execution should be disabled before branch/jump prediction, as parallel execution benefits get largely lost without the smoother flow of instructions provided by branch/jump prediction. Furthermore, delayed execution should be disabled before parallel execution, as the latter delivers a larger performance gain.

Sample timing diagrams

Core reset

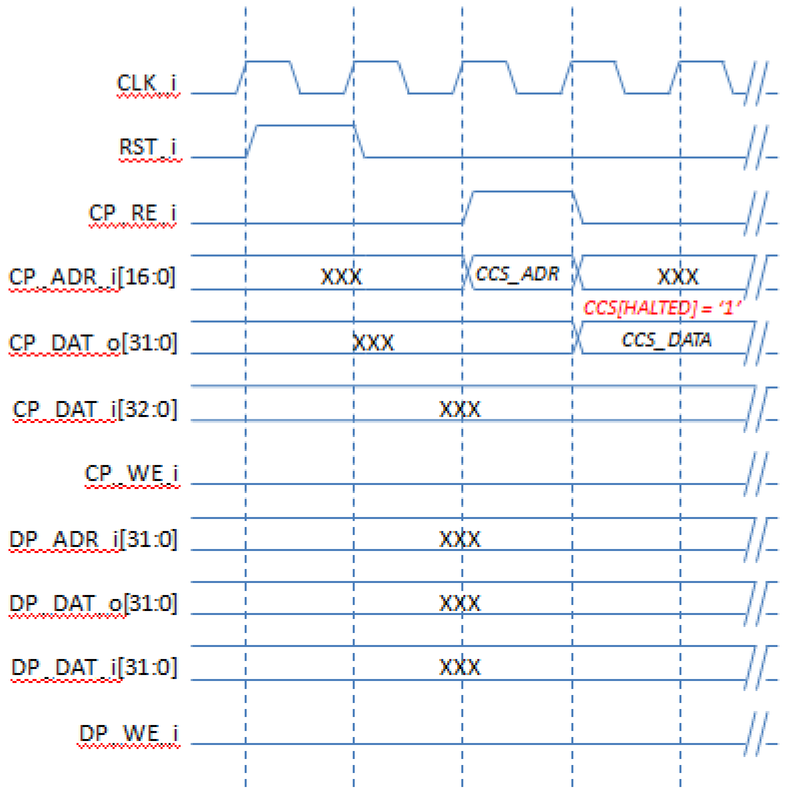

Fig. 1: reset (with Debug module) timing diagram. After reset assertion, Debug module `CCS` register is read to check core is halted (`CCS[HALTED]` = '1').

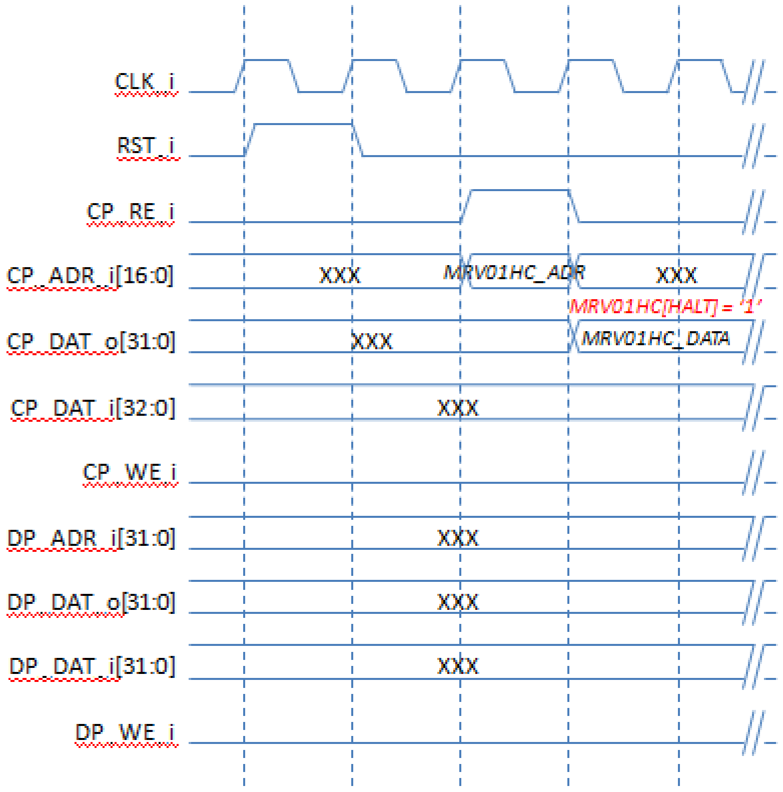

Fig. 2: reset (with Halt module) timing diagram. After reset assertion, Halt module `MRV01HC` register is read to check core is halted (`MRV01HC[HALT]` = '1').

CSR's access

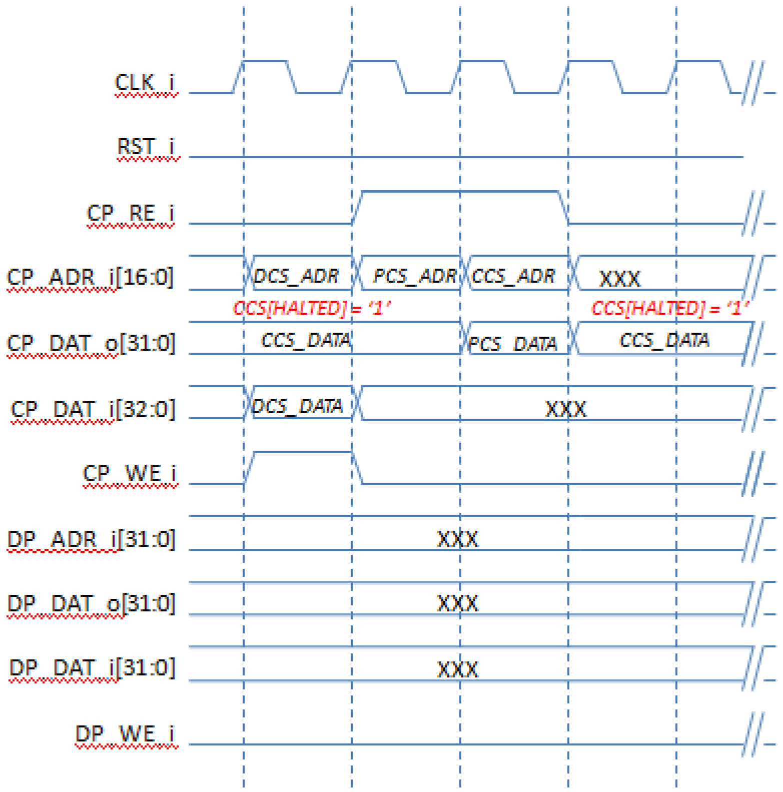

Fig. 3: CSR's access (with Debug module) timing diagram. The core remains halted while reading/writing CSR's, as shown by `CCS[HALTED]` = '1'

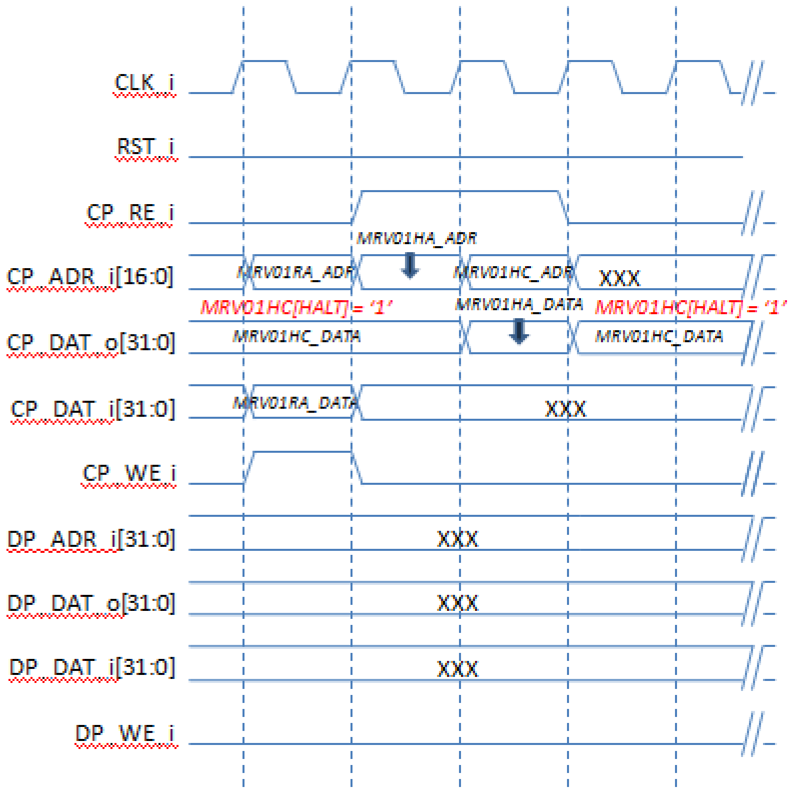

Fig. 4: CSR's access (with Halt module) timing diagram. The core remains halted while reading/writing CSR's, as shown by `MRV01HC[HALT]` = '1'

DMA access

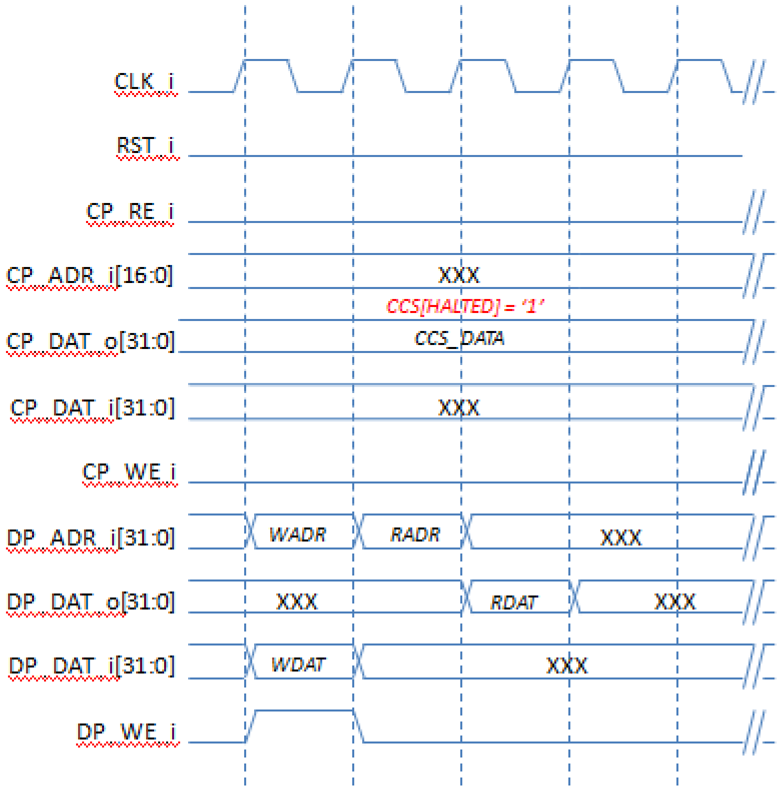

Fig. 5: DMA access (with Halt module) timing diagram. The core remains halted while reading/writing internal memory, as shown by `MRV01HC[HALT]` = '1'

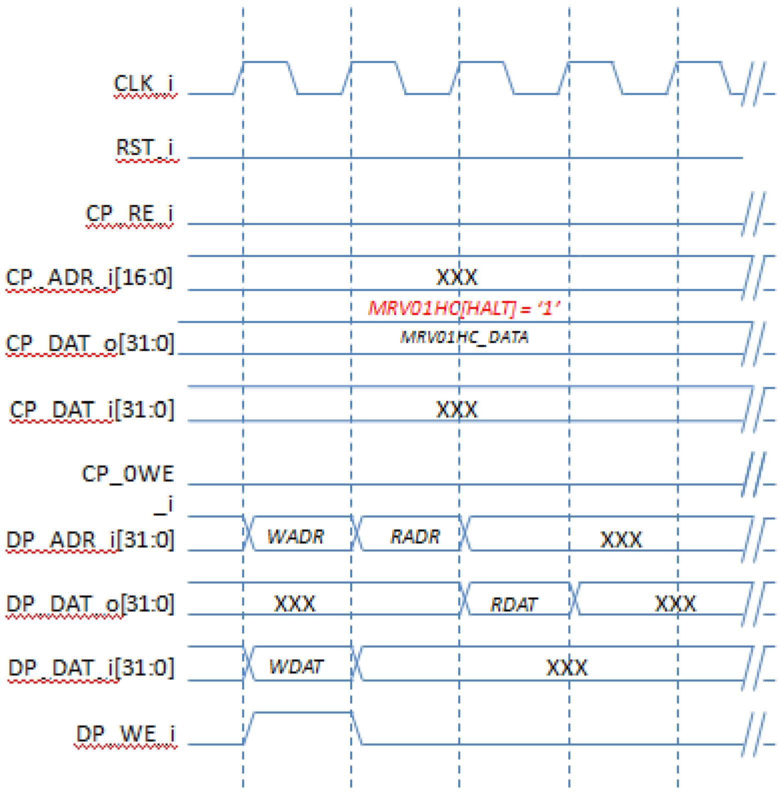

Fig. 6: DMA access (with Halt module) timing diagram. The core remains halted while reading/writing internal memory, as shown by `MRV01HC[HALT]` = '1'

Starting/Resuming execution

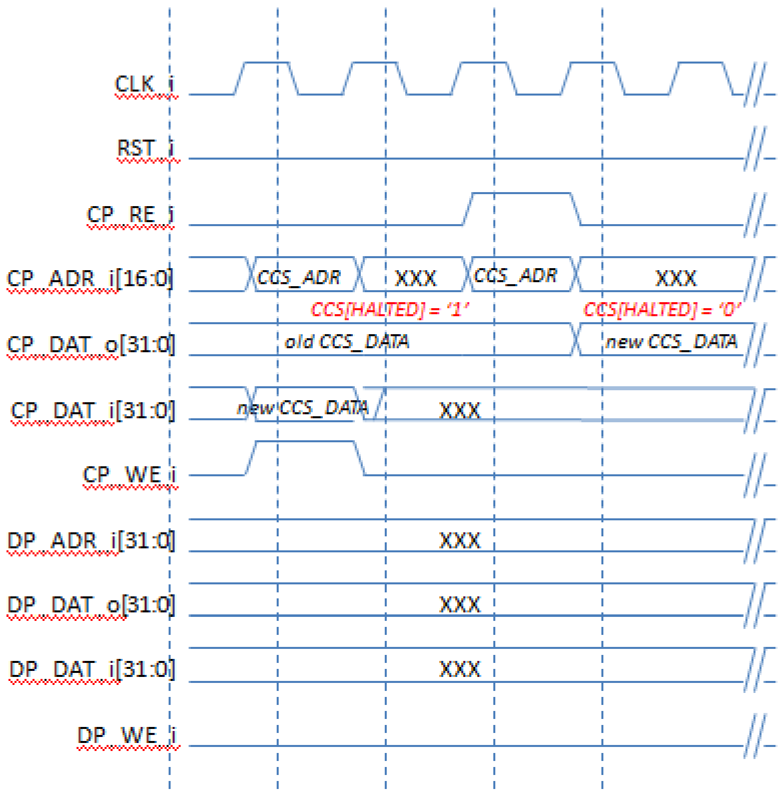

Fig. 7: Start/resume execution (with Debug module) timing diagram. After `CSS` CSR, the core starts/resumes execution, as shown by `CSS[HALTED]` level going from '1' to '0'

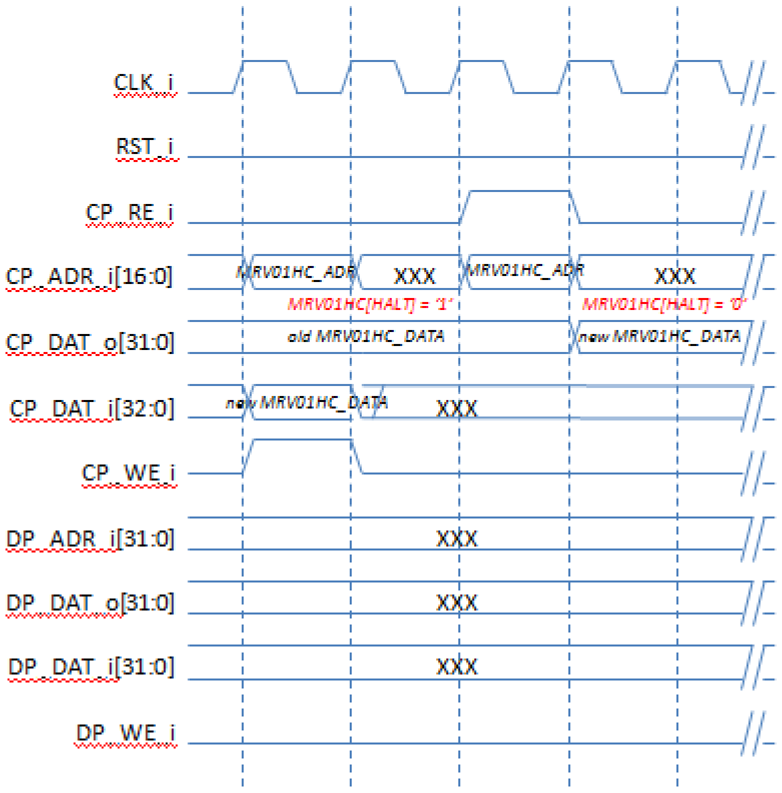

Fig. 8: Start/resume execution (with Halt module) timing diagram. After writing `MRV01HC` CSR, the core starts/resumes execution, as shown by `MRV01HC[HALT]` level going from '1' to '0'

Appendix A: Xilinx© Vivado 17.3 synthesis test

Top-level module: RV01_TOP_SYN (synthesis test-bench) from file RV01_TOP_SYN.vhd.
Configuration parameters:

- IMEM_SIZE = 8192 (32KB).
- DMEM_SIZE = 8192 (32KB).
- IOMEM_SIZE = 0.
- IMEM_SIZE_PO2 = '1'.
- DMEM_SIZE_PO2 = '1'.
- IMEM_LOWM = '1'.
- BHT_SIZE = 512.
- EI_SRC_CNT = 8 (*).
- EI_TRIG_TYPE = LEVEL (*).
- EI_REQ_MAXCNT = 16 (*).
- PARALLEL_EXECUTION_ENABLED = '1'.
- DELAYED EXECUTION ENABLED = '1'.
- BRANCH PREDICTION ENABLED = '1'.
- JALR PREDICTION ENABLED = '1'.
- FPU_PRESENT = '0'.
- DM_PRESENT = '0'.
- PLIC_PRESENT = '0'.

(*) = This value is un-relevant as PLIC module is not present.

Target device: xc6vlx75t-2ff484.
Target Fmax: 143MHz (default synthesis options)..
Results:

- Number of slice registers: 1574 (1%)
- Number of slice LUTs: 4284 (9%)
- Number of RAMB36: 18 (11%)
- Number of DSP48E1s: 4 (1%)

Appendix B: Simulation & Implementation Hints

- The suggested starting point, after downloading the project from OpenCores site, is to run the self-test module simulation (related VHDL files are located in VHDL/SELF_TEST folder, the top-level file, including the test bench, being VHDL/RV01_selftest_TB.vhd). This simulation allows verifying that all design files are available and can be compiled correctly, and provides also an example of core top-level module instantiation and interfacing to user logic. The self-test module is of very simple use, including only four I/O signals: two input signals (CLK_i and RST_i) and two output ones (DONE_o and PASS_o), all active-high.
- Once self-test module simulation runs successfully, a simple test on HW can be performed by implementing the self-test module itself on an FPGA board (so far the module has been successfully implemented on a Xilinx Artix-7 FPGA's). The simplest approach is to connect self-test module CLK_i and RST_i inputs to the board FPGA clock and cpu/user reset pins, and

to connect self-test module `DONE_o` and `PASS_o` outputs to board LED's (if available). Pay attention to polarity of input/output signals (self-test module signals are active-high, while some board signal may be active-low). A more sophisticated test may consist in adding a clock generator core (a PLL) and derive self-test module reset signal from the clock generator lock/stable signal.

- Self-test module consists of a RV01 core with Halt module (and without PLIC module), plus some self-test control logic and ROM memories. After reset is de-asserted, the control logic loads a sample program (a Dhrystone test) into the core internal memories from ROM ones, configures the Halt module to stop at a specific address (located at the end of the Dhrystone test core loop) and then starts sample program execution. When the core stops execution at the address specified by the Halt module, the self-test control logic checks loop iteration results by inspecting core memory content through the DMA port, and then resumes program execution from last not-executed instruction, leaving halt condition un-changed (in this way the core executes another Dhrystone loop iteration before stopping again). The Dhrystone loop is executed `CNTR_MAX` time (`CNTR_MAX` value being specified in `RV01_selftest.vhd file`). When core stops execution after last iteration, the self-test control logic configures the Halt module to stop execution at Dhrystone test last instruction and then resumes execution one last time, allowing the sample program to complete. When this event occurs, the self-test control logic assert the module outputs `DONE_o` and (if no error was reported during result checks) `PASS_o`.
- When targeting Virtex-5 FPGA family (or older ones), add "-use_new_parser yes" to synthesis other options (the default parser used for such FPGA families doesn't synthesize some portion of VHDL code correctly). Check synthesis report file to verify option has been properly specified.

Appendix C: Bibliography

1. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0", riscv-spec-v2.0.pdf.
2. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1", riscv-priv-spec-1.7.pdf.
3. "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.7", riscv-spec-v2.1.pdf
4. "RISC-V External Debug Support, Version 0.9jan27", riscv-debug-spec_v0.9.pdf.