VexRiscv CPU v1.0

IP User Guide (Beta Release)

January 20, 2023

Contents

IP Summary

Introduction

The VexRiscv CPU is a 32 bit, AXI4 compliant soft processor designed to be used in applications that require fast computations on FPGAs in the form of soft SoCs.
It is a modern and complete soft processor that can be used to boot Operating Systems or used in a bare metal fashion. This soft processor is based on RISCV ISA, **RV32IM CPU** and hence support the entirety of RISCV's instructions. It is designed to be configurable based on the requirement and footprint size. The overall CPU is comprised of many different customizable plugins that have the access to the whole CPU.

Features

- Support IEEE 754 float and optionally double
- Implement Subnormal
- Implement exceptions flags
- The FPU can be shared between multiple CPU
- Fully pipe-lined, can produce one result per cycle for most operations as long there is no inter-dependencies
- Implement multiplication using multiple sub multiplication operations in parallel
- Division done with radix 4
- Square root done with radix 2
- Optional MMU
- Optional Instruction and Data Caches
- Optional interrupts and exception handling with the Machine and the User mode from the riscv-privileged-v1.9.1 spec.

Overview

VexRiscv CPU

VexRiscv is implemented via a 5 stage in-order pipeline on which many optional and complementary plugins add functionalities to provide a functional RISC-V CPU. It is currently implemented as a single core processor with JTAG support for debugging purposes along with the capability to handle various timers and interrupts. Being a modular design, this soft processor is made up of several smaller components that work together in harmony to make a functional processor. The internal modules in the CPU depends on how the CPU is configured from the IP Generator. A macro block diagram of the internals of the soft processor can be visualized in Figure 1.

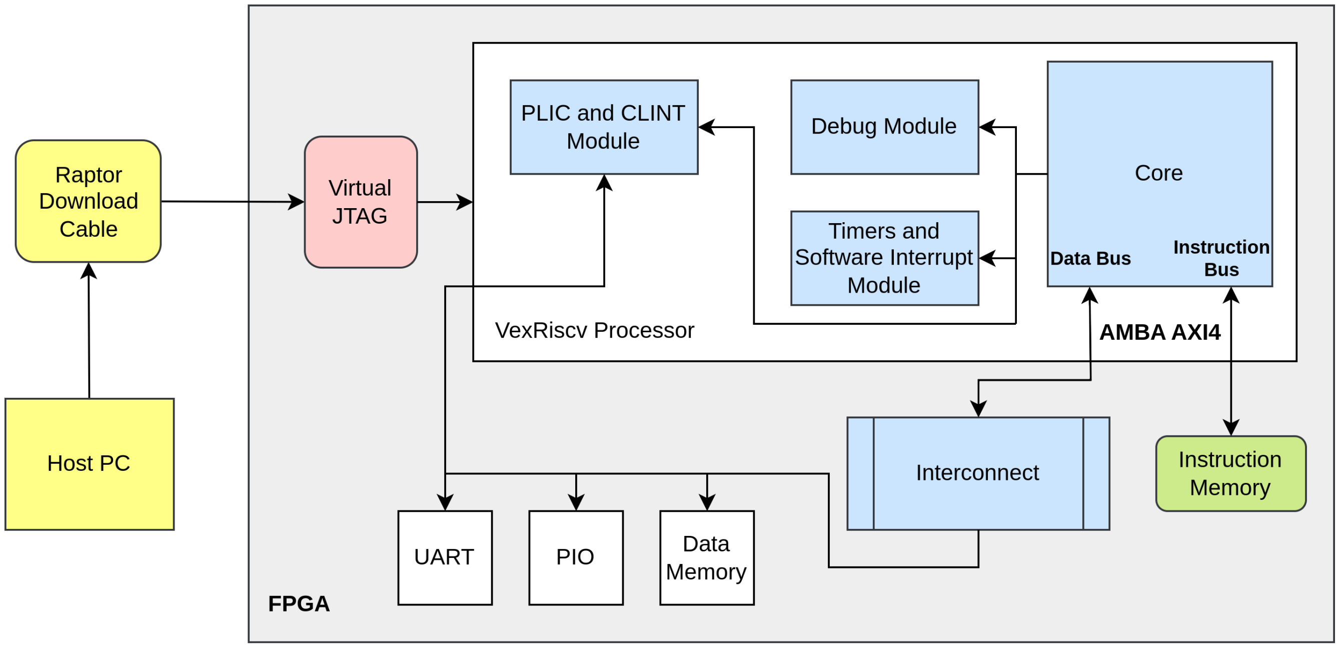


Figure 1. CPU Internals

Licensing

COPYRIGHT TEXT:

Copyright (c) 2022 RapidSilicon

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

IP Specification

The figure 2 shows the internal block diagram of the VexRiscv CPU. It is a pipelined processor with 5 stages, i.e., **Fetch, Decode, Execute, Memory and Writeback** as can be seen in Figure 2. The functionality of the integrated FPU can also be analyzed by the block diagram. It deals with non-SMP and SMP processors alike with a pipelined combination of a number of Loaders, Multipliers, Adders, Divisors and Square Root blocks. The CPU can either be uncached or have a cached configuration. All transactions in the uncached configuration are of non-burst type while burst mode is activated in the cached configurations. Keeping in mind that the cache is 4kB in size and its implementation is write through hence there is no write burst but only individual write transactions. More information on the VexRiscv modular integration can be read from the creator's manual from here.

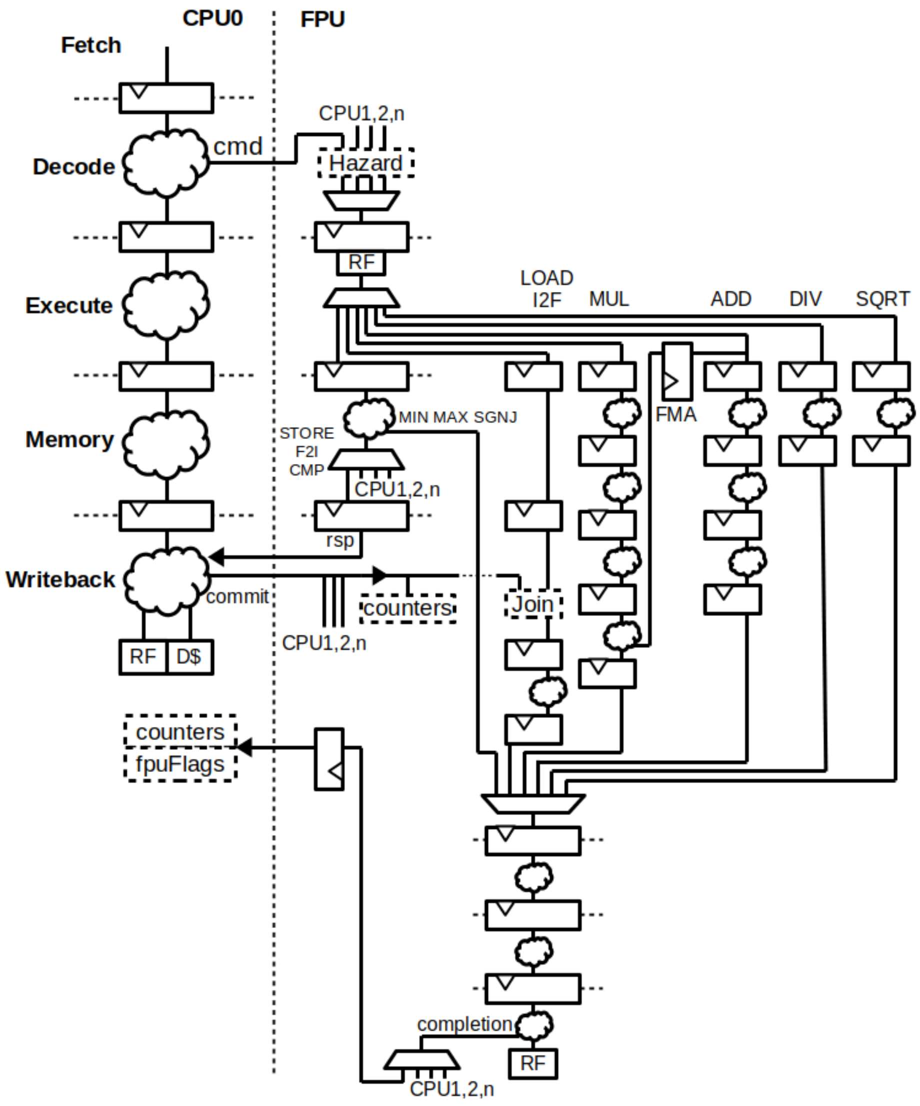


Figure 2. Top Module

Standards

The AXI4 interface is compliant with the AMBA® AXI Protocol Specification.

IP Support Details

The Table 1 gives the support details for VexRiscv CPU.

Compliance		IP Resources					Tool Flow		
Device	**Interface**	**Source Files**	**Constraint File**	**Testbench**	**Simulation Model**	**Software Driver**	**Analyze and Elaboration**	**Simulation**	**Synthesis**
GEMINI	AXI4	Verilog	SDC	Verilog/C	-	-	Raptor	Raptor	Raptor

Table 1. IP Details

Resource Utilization

The resource utilization for all the configurations for the VexRiscv processor are given in Table 2.

Tool	Raptor Design Suite		
FPGA Device	GEMINI		
Configuration		**Resource Utilization**	
Minimum Resource	Uncached CPU with MMU	**Resource**	**Utilized**
		LUT	1960
		Registers	1310
		BRAM	1
		DSP	4
Median Resource	Cached CPU with MMU	**Resource**	**Utilized**
		LUT	3090
		Registers	2441
		BRAM	6
		DSP	4
Maximum Resource	Cached CPU with MMU, PLIC and CLINT	**Resource**	**Utilized**
		LUT	3177
		Registers	2506
		BRAM	6
		DSP	4

Table 2. VexRiscv Resource Utilization

Ports

Table 3 lists the top interface ports of the VexRiscv CPU.

Signal Name	I/O	Description
AXI Clock and Reset		
clk	I	AXI4 Clock
rst	I	AXI4 RESET
Instruction Bus Channel		
AXI READ ADDRESS CHANNEL		
ibus_axi_arvalid	O	AXI4 Read address valid
ibus_axi_arready	I	AXI4 Read address ready
ibus_axi_araddr	O	AXI4 Read address
ibus_axi_arprot	O	AXI4 Read address Protection type
ibus_axi_arburst	O	AXI4 Read address Burst mode
ibus_axi_arlen	O	AXI4 Read address length of burst
ibus_axi_arsize	O	AXI4 Read address size of burst
ibus_axi_arlock	O	AXI4 Read address lock type
ibus_axi_arcache	O	AXI4 Read address cache
ibus_axi_arqos	O	AXI4 Read address quality of service identifier
ibus_axi_arregion	O	AXI4 Read address region identifier
ibus_axi_arid	O	AXI4 Read address ID
ibus_axi_aruser	O	AXI4 Read address user signal
AXI READ DATA CHANNEL		
ibus_axi_rvalid	I	AXI4 Read valid
ibus_axi_rready	O	AXI4 Read ready
ibus_axi_rdata	I	AXI4 Read Data
ibus_axi_rresp	I	AXI4 Read Response
ibus_axi_rlast	I	AXI4 Read last transfer
ibus_axi_rid	I	AXI4 Read ID
ibus_axi_ruser	I	AXI4 Read user signal
Data Bus Channel		
AXI WRITE ADDRESS CHANNEL		
dbus_axi_awvalid	O	AXI4 Write address valid
dbus_axi_awready	I	AXI4 Write address ready
dbus_axi_awaddr	O	AXI4 Write address
dbus_axi_awprot	O	AXI4 Write address Protection type
dbus_axi_awburst	O	AXI4 Write address Burst mode
dbus_axi_awlen	O	AXI4 Write address length of burst
dbus_axi_awsize	O	AXI4 Write address size of burst
dbus_axi_awlock	O	AXI4 Write address lock type
dbus_axi_awcache	O	AXI4 Write address cache
dbus_axi_awqos	O	AXI4 Write address quality of service identifier
dbus_axi_awregion	O	AXI4 Write address region identifier
dbus_axi_awid	O	AXI4 Write address ID
dbus_axi_awuser	O	AXI4 Write address User signal
AXI WRITE DATA CHANNEL		
dbus_axi_wvalid	O	AXI4 Write valid

dbus_axi_wready	I	AXI4 Write ready
dbus_axi_wdata	O	AXI4 Write Data
dbus_axi_wstrb	O	AXI4 Write Strobe
dbus_axi_wlast	O	AXI4 Write last transfer
dbus_axi_wuser	O	AXI4 Write User signal
AXI WRITE RESPONSE CHANNEL		
dbus_axi_bvalid	I	AXI4 Write Response valid
dbus_axi_bready	O	AXI4 Write Response ready
dbus_axi_bresp	I	AXI4 Write Response
dbus_axi_bid	I	AXI4 Write Response ID
dbus_axi_buser	I	AXI4 Write Response User signal
AXI READ ADDRESS CHANNEL		
dbus_axi_arvalid	O	AXI4 Read address valid
dbus_axi_arready	I	AXI4 Read address ready
dbus_axi_araddr	O	AXI4 Read address
dbus_axi_arprot	O	AXI4 Read address Protection type
dbus_axi_arburst	O	AXI4 Read address Burst mode
dbus_axi_arlen	O	AXI4 Read address length of burst
dbus_axi_arsize	O	AXI4 Read address size of burst
dbus_axi_arlock	O	AXI4 Read address lock type
dbus_axi_arcache	O	AXI4 Read address cache
dbus_axi_arqos	O	AXI4 Read address quality of service identifier
dbus_axi_arregion	O	AXI4 Read address region identifier
dbus_axi_arid	O	AXI4 Read address ID
dbus_axi_aruser	O	AXI4 Read address user signal
AXI READ DATA CHANNEL		
dbus_axi_rvalid	I	AXI4 Read valid
dbus_axi_rready	O	AXI4 Read ready
dbus_axi_rdata	I	AXI4 Read Data
dbus_axi_rresp	I	AXI4 Read Response
dbus_axi_rlast	I	AXI4 Read last transfer
dbus_axi_rid	I	AXI4 Read ID
dbus_axi_ruser	I	AXI4 Read user signal
JTAG PORTS		
jtag_tms	I	JTAG Mode select
jtag_tdi	I	JTAG Data in
jtag_tdo	O	JTAG Data out
jtag_tck	I	JTAG Clock
DEBUG PORTS		
debugReset	I	Debug Mode In
debug_resetOut	O	Debug Mode Out
CLINT PORTS		
clint_awvalid	I	AXI4-Lite Write valid
clint_awaddr	I	AXI4-Lite Write Address
clint_awprot	I	AXI4-Lite Write Address Protection
clint_awready	O	AXI4-Lite Write Address Ready

clint_wvalid	I	AXI4-Lite Write Valid
clint_wready	O	AXI4-Lite Write Ready
clint_wdata	I	AXI4-Lite Write Data
clint_wstrb	I	AXI4-Lite Write Strobe
clint_bready	I	AXI4-Lite Write Response Ready
clint_bvalid	O	AXI4-Lite Write Response Valid
clint_bresp	O	AXI4-Lite Write Response
clint_arvalid	I	AXI4-Lite Read Address Valid
clint_arready	O	AXI4-Lite Read Address Ready
clint_araddr	I	AXI4-Lite Read Address
clint_arprot	I	AXI4-Lite Read Address Protection
clint_rready	I	AXI4-Lite Read Ready
clint_rvalid	O	AXI4-Lite Read Valid
clint_rresp	O	AXI4-Lite Read Response
PLIC PORTS		
plic_awvalid	I	AXI4-Lite Write valid
plic_awaddr	I	AXI4-Lite Write Address
plic_awprot	I	AXI4-Lite Write Address Protection
plic_awready	O	AXI4-Lite Write Address Ready
plic_wvalid	I	AXI4-Lite Write Valid
plic_wready	O	AXI4-Lite Write Ready
plic_wdata	I	AXI4-Lite Write Data
plic_wstrb	I	AXI4-Lite Write Strobe
plic_bready	I	AXI4-Lite Write Response Ready
plic_bvalid	O	AXI4-Lite Write Response Valid
plic_bresp	O	AXI4-Lite Write Response
plic_arvalid	I	AXI4-Lite Read Address Valid
plic_arready	O	AXI4-Lite Read Address Ready
plic_araddr	I	AXI4-Lite Read Address
plic_arprot	I	AXI4-Lite Read Address Protection
plic_rready	I	AXI4-Lite Read Ready
plic_rvalid	O	AXI4-Lite Read Valid
plic_rresp	O	AXI4-Lite Read Response
plicInterrupts	I	PLIC Interrupts
OTHER PORTS		
timerInterrupt	I	Timer Interrupt
externalInterrupt	I	External Interrupt
softwareInterrupt	I	Software Interrupt

Table 3. VexRiscv Interface

Registers Address Space

Since this is a RISCV compliant CPU, it supports all the compatible registers found in the RISCV ISA. For the complete register address space, refer to the RISCV Privileged Specification. The CSRs can be configured via a firmware to enable the processor for various sorts of tasks and functions.

IO Range

The entirety of the Address Range can be used as the IO Range for the uncached CPU configuration but for the cached configuration, IO Range is divided into cached and uncached regions which can be referred from Table 4.

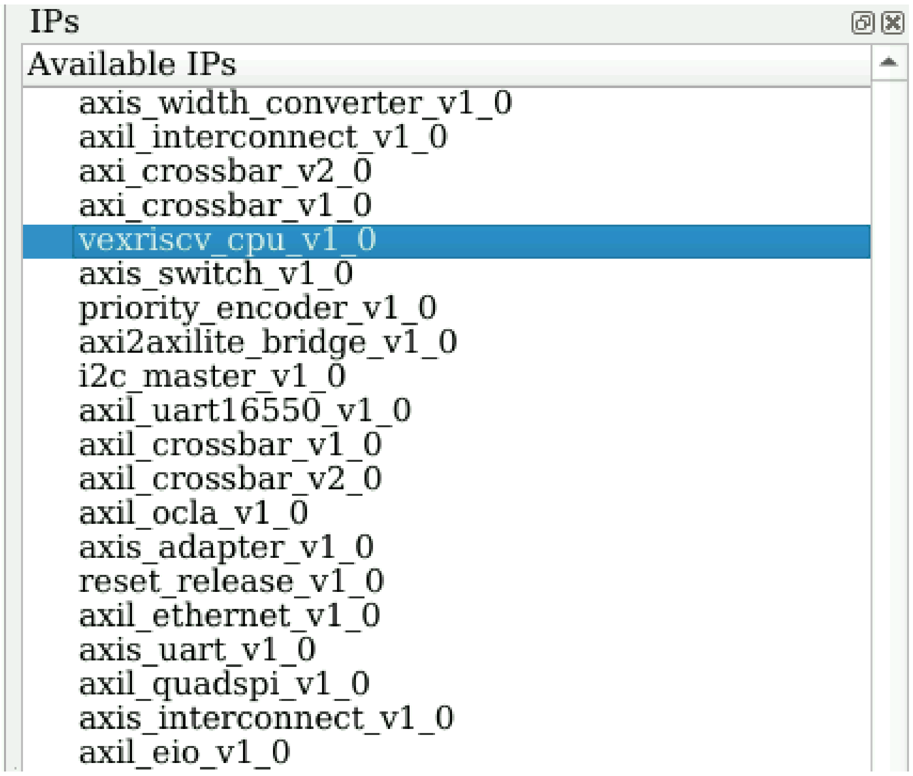
Configuration	Cached	Uncached
Base Variant	-	Entire Range
Cached with MMU	0x00000000-0xEFFFFFFF	>0xF0000000
Cached with MMU, PLIC and CLINT	0x00000000-0xEFFFFFFF	>0xF0000000

Table 4. IO Range

Design Flow

IP Customization and Generation

VexRiscv CPU is a part of the Raptor Design Suite Software. Three different configurations of the CPU can be generated from the Raptor's IP configurator window by selecting it from the IP Catalog as shown in Figure 3.



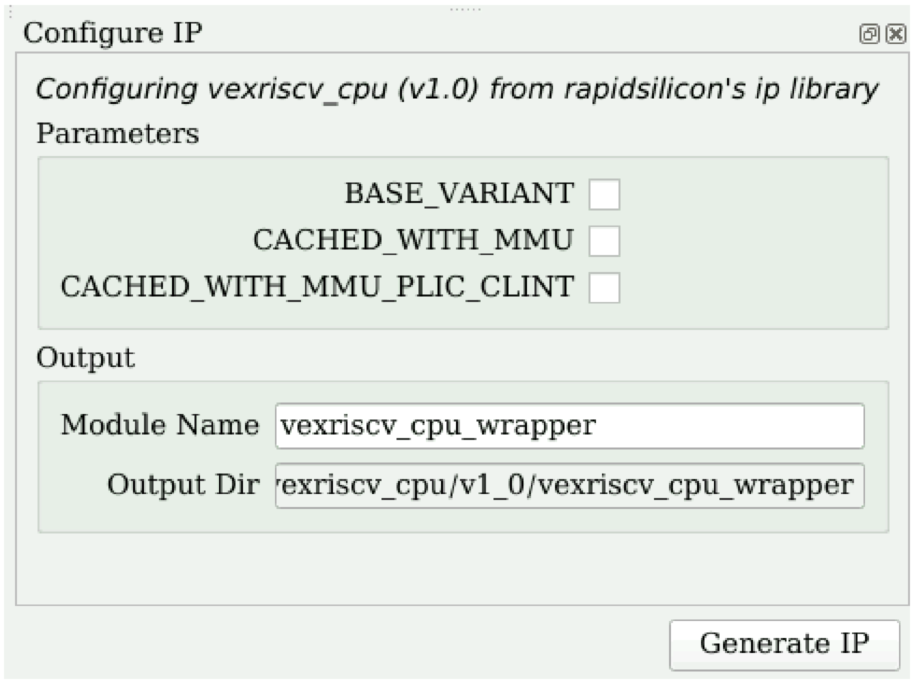
```
axis_width_converter_v1_0
axil_interconnect_v1_0
axi_crossbar_v2_0
axi_crossbar_v1_0
vexriscv_cpu_v1_0
axis_switch_v1_0
priority_encoder_v1_0
axi2axilite_bridge_v1_0
i2c_master_v1_0
axil_uart16550_v1_0
axil_crossbar_v1_0
axil_crossbar_v2_0
axil_ocla_v1_0
axis_adapter_v1_0
reset_release_v1_0
axil_ethernet_v1_0
axis_uart_v1_0
axil_quadspi_v1_0
axis_interconnect_v1_0
axil_eio_v1_0
```

Figure 3. IP list

Parameters Customization: From the IP configuration window, the required configuration of the CPU can be selected that suits the user application requirement. As can be seen in the picture below, user can select one of the three available configurations for the VexRiscv CPU. The details of which are given below: -

- **Base Variant** This base variant of the CPU is a cacheless design with no MMU. This design has the smallest footprint and hence is low-end FPGA friendly. Since the whole CPU is cacheless, the entire ioRange can be accessed atomically.
- **Cached with MMU** This variant of the CPU has individual caches on both the Instruction and Data Bus and hence help improving the overall speed of the processor for cache friendly operations at the cost of more memory requirement on the FPGA. For the uncached ioRange of this variant, refer to the earlier subsection of **ioRanges**.
- **Cached with MMU, PLIC and CLINT** This variant of the is the most memory extensive among the list of configurations. This adds seperate modules for PLIC and CLINT along with the caches and MMU as in the previous configuration. For the uncached ioRange of this variant, refer to the earlier subsection of **ioRanges**. This processor can be used when implementing own drivers and programs for the PLIC and CLINT modules for the RISCV platform.

It is to be noted here that all the three configurations of the VexRiscv processor are mutually exclusive and hence only one configuration can be activated at one time and the source files are made available to the user with the top wrapper that instantiates the require configuration of the VexRiscv processor. The configurator window can be seen in Figure 4.


Configure IP

Configuring vexriscv_cpu (v1.0) from rapidsilicon's ip library

Parameters

BASE_VARIANT ☐
CACHED_WITH_MMU ☐
CACHED_WITH_MMU_PLIC_CLINT ☐

Output

Module Name: vexriscv_cpu_wrapper
Output Dir: 'exriscv_cpu/v1_0/vexriscv_cpu_wrapper

Figure 4. IP Configuration

Design Flow

The required configuration of the CPU is generated from the Raptor Suite based on the application of the processor. From there, processor goes through a series of steps to be able to be used in the required application. These steps essentially ready the processor for the task at hand and hence comprise of both the hardware and software flow, both of which are described in detail in Figure 5.

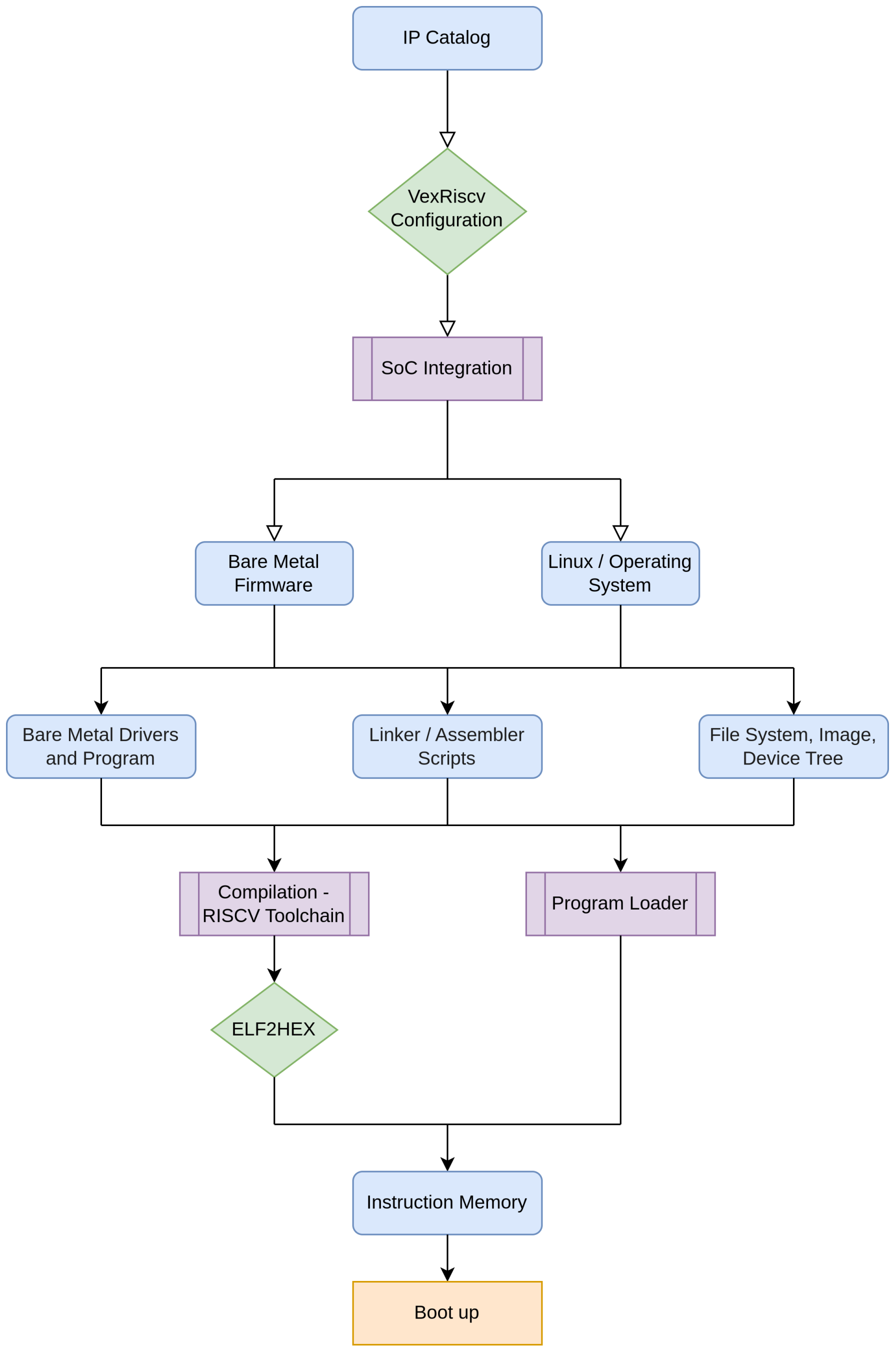


Figure 5. VexRiscv Design Flow

Software Development Cycle

The VexRiscv processor, being RISCV compliant, executes and runs the RISCV ISA. Meaning that to run a bare-metal program on the VexRiscv processor, it requires instructions written with RISCV linker and assembler scripts. Since this is a Verilog HDL based IP, it can read these instructions in either binary or hexadecimal format. To prepare the hex file for the VexRiscv processor, following steps are to be executed:

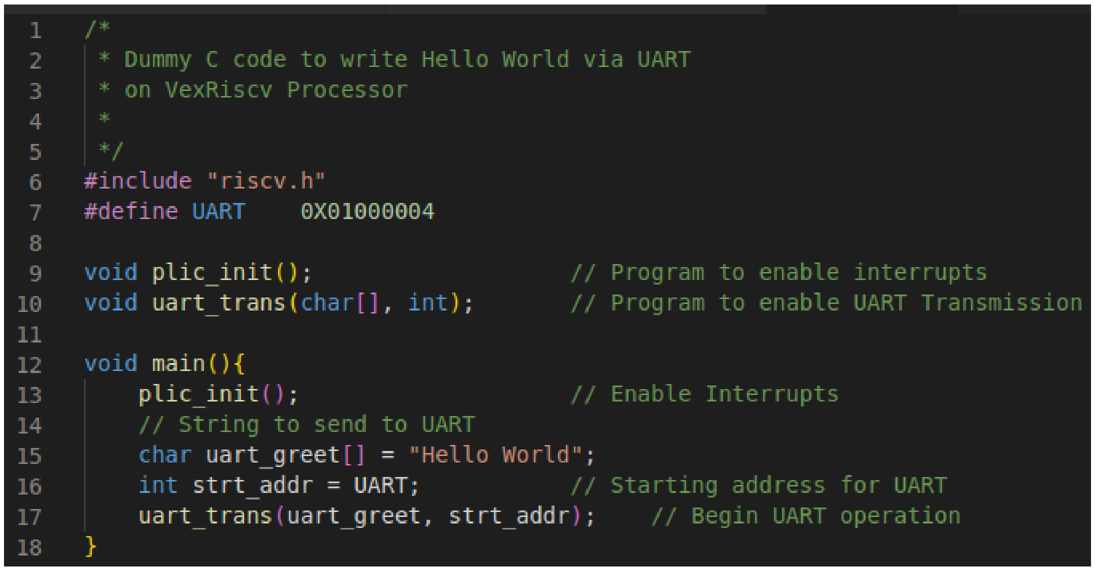
- The **RISCV toolchain** required for the compilation of the software can be downloaded and installed via the following commands making sure that the downloaded toolchain is compatible with the host OS.

```
$ wget https://static.dev.sifive.com/dev-tools/freedom-tools/v2020.12/riscv64-unknown-
    elf-toolchain-10.2.0-2020.12.8-x86_64-linux-centos6.tar.gz
$ tar -xvf riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-centos6.tar.gz
$ export PATH=$PATH:$PWD/riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86_64-linux-
    centos6/bin/
```

- The assembler is responsible for translating higher level language's instructions into RISCV ISA.
- The linker then is responsible for linking of all the object files together in accordance with the VexRiscv address configuration, during the compilation.
- The main program can be written in C language, to keep things simple, and later on linked with the assembler via the linker script.
- This makes for at least three files to run a bare metal program on the VexRiscv processor, i.e., a **dummy.c** file containing our fundamental code that is to be run on the VexRiscv processor,

```
$ riscv64-unknown-elf-gcc -c -march=rv32im  -mabi=ilp32 -g -O3  -MD -fstrict-volatile-
    bitfields    -o build/src/dummy.o src/dummy.c
```

A snippet for a piece of code written for the VexRiscv CPU can be seen in the Figure 6.



```
/*
 * Dummy C code to write Hello World via UART
 * on VexRiscv Processor
 *
 */
#include "riscv.h"
#define UART    0X01000004

void plic_init();                   // Program to enable interrupts
void uart_trans(char[], int);       // Program to enable UART Transmission

void main(){
    plic_init();                    // Enable Interrupts
    // String to send to UART
    char uart_greet[] = "Hello World";
    int strt_addr = UART;           // Starting address for UART
    uart_trans(uart_greet, strt_addr);    // Begin UART operation
}
```

Figure 6. Dummy Code for VexRiscv

- A **crt.S** file for the RISCV assembly root instructions,

```
$ riscv64-unknown-elf-gcc -c -march=rv32im  -mabi=ilp32 -g -O3  -MD -fstrict-volatile-
    bitfields  -o build/src/crt.o src/crt.S -D__ASSEMBLY__=1
```

- And a **linker.ld** script to link all the object files together according to the address configuration of the soft processor,

```
$ riscv64-unknown-elf-gcc -march=rv32im  -mabi=ilp32 -g -O3  -MD -fstrict-volatile-
    bitfields  -o build/$(DESIGN_NAME).elf build/src/dummy.o build/src/crt.o -march=
    rv32im  -mabi=ilp32 -nostdlib -lgcc -mcmodel=medany -nostartfiles -ffreestanding -
    Wl,-Bstatic,-T,./libs/linker.ld,-Map,build/$(DESIGN_NAME).map,--print-memory-usage
```

- After these three commands, an executable .elf file is made available in the local directory. This .elf file is then converted into a Verilog readable **hex** file by **elf2hex** tool that can be downloaded directly from the following commands,

```
$ wget https://github.com/sifive/elf2hex/archive/refs/tags/v20.08.00.00.tar.gz
$ tar -xvf v20.08.00.00.tar.gz
$ export PATH=$PATH:$PWD/v20.08.00.00/bin/
```

- Finally the hex can be generated from the elf file by typing out the following command in the terminal after the elf file has been created,

```
$ riscv64-unknown-elf-elf2hex --bit-width 32 --input $(path to .elf file) --output $(
    path where to create .hex file)
```

After the above command is executed, the newly formed hex file can be loaded in the Instruction Memory for the VexRiscv processor and the Processor will start executing these commands when booted up.

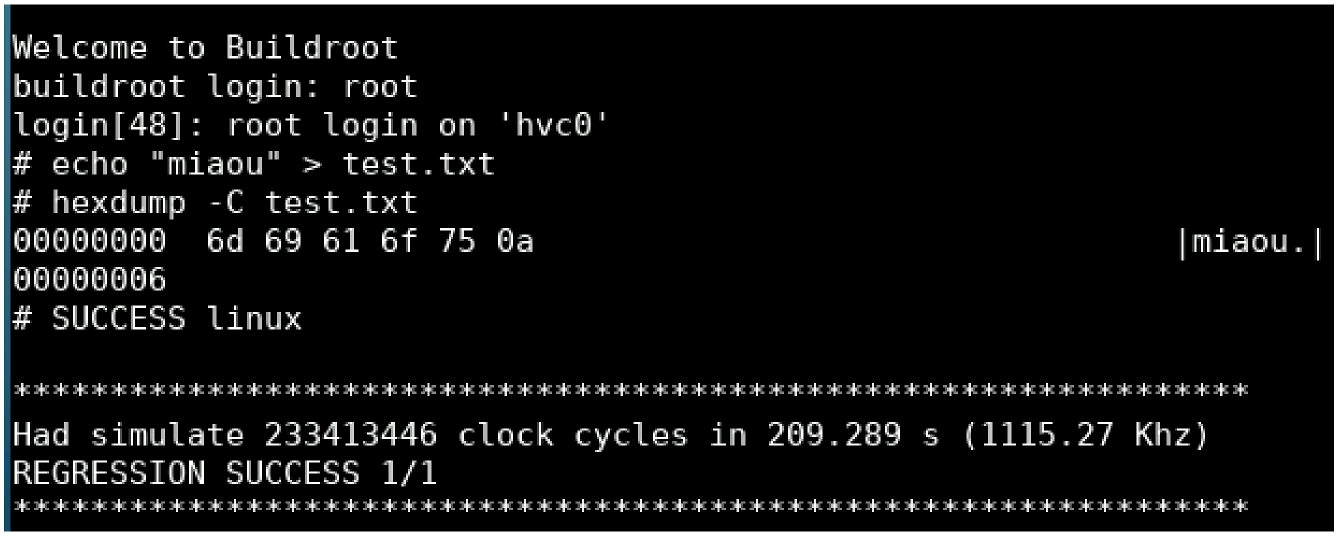
VexRiscv Linux Boot

The Boot Sequence

Booting Linux on the VexRiscv CPU requires the presence of a MMU and to keep things fast enough, cached configuration is required. For this purpose and to keep the configuration basic, the second configuration of the VexRiscv CPU, i.e., Cached with MMU, is made use of. A RISCV linker script is utilized to bring all the required files, for Linux boot, together. This linker script can also be replaced by a compatible BIOS such as **OpenSBI**. The bundled simulator is utilized to simulate the whole boot sequence to get to a Linux Terminal in an attempt to show what the VexRiscv processor is capable of. With the correct assembler and linker scripts, this can also be achieved on an FPGA board by burning the required files onto an SD card.
Any compatible version of Linux kernel can be used but for this example, the **Linux version 4.20.17** is bundled with the Raptor Suite along with all the required files for the boot up which include: -

- Linux version 4.20.17
- A device tree for hardware description
- A rootfs for the file system
- An Emulator for the simulator

The boot sequence running in the bundled simulator can be shown in Figure 7. The prompt can be seen at the end of the boot sequence when Linux is completely booted up and a pre-defined regression is run on the booted up Linux, all in simulation.



```
Welcome to Buildroot
buildroot login: root
login[48]: root login on 'hvc0'
# echo "miaou" > test.txt
# hexdump -C test.txt
00000000  6d 69 61 6f 75 0a                                 |miaou.|
00000006
# SUCCESS linux

****************************************************************
Had simulate 233413446 clock cycles in 209.289 s (1115.27 Khz)
REGRESSION SUCCESS 1/1
****************************************************************
```

Figure 7. Linux Boot

To fire up Linux on simulation using the bundled Verilator, the user just needs to provide the paths of all the required files, stated above, to the C++ file for Verilator. An example of such a file can also be found in the **/sim** directory. The AXI protocol needs to be followed, and the Verilator can simulate an example of a working and complete Linux experience. An example of the Verilator command for the simulation can also be seen in the Makefile present in the /sim directory. This C++ file can then be transformed into a linker script, if required, to be able to boot Linux on an FPGA board on this VexRiscv processor.

Modularity

Since the CPU itself is made up of various plugins and hence can have various configurations depending upon the required functionality, Linux too can be build with different various required drivers and IOs by modifying the device tree and the rootfs file system. This means that in theory, a complete Linux system can be made solely on simulation using this VexRiscv processor prior to moving towards actual hardware which can save both cost and time waiting for the hardware to be ready. This also means that this exact design can then be shifted to an FPGA by generating the bitstream via the Raptor Suite.

Example Design

Overview

Any of these CPU configurations can be used in a multitude of ways among which in an SoC is maybe the most common use for any CPU. One such example is shown in Figure 8 where the processor with PLIC and CLINT is utilized in an SoC. A pictorial representation for the SoC can be found below with a bried elaboration on the components attached: -

- **AXI RAM** acting as the ROM for the CPU
- **AXI2AXILite bridge** to be able to attach AXI-Lite IOs
- **AXILite Interconnect** for routing traffic between the IOs and the CPU
- **AXI RAM** acting as an attached storage device on IO
- **AXILite UART** connected as the second IO to generate interrupts to be handled by PLIC

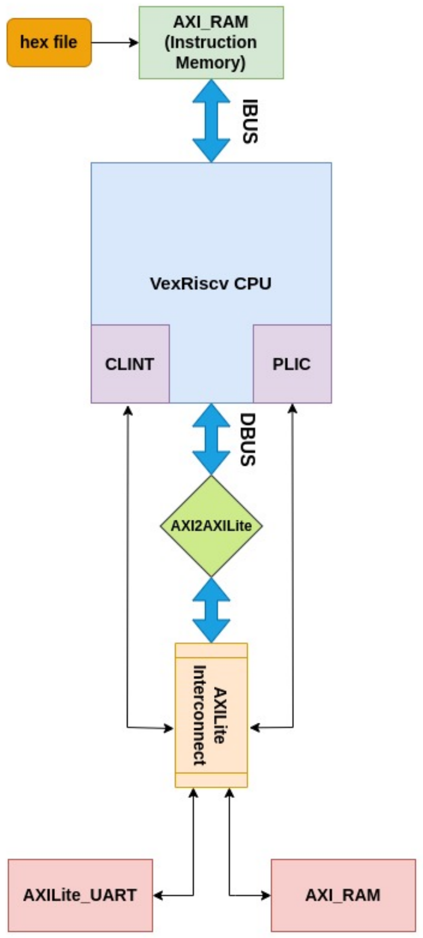


Figure 8. VexRiscv SoC

Simulating the Example Design

The design can be readily simulated with the bundled simulator by importing the design in the Raptor Suite and hitting the simulate button. The dumped waveform can be analyzed to see the internal workings of the CPU in the SoC configuration a part of which demonstrating the working of the UART in the SoC is shown in Figure 9. As it can be seen, the generated interrupt travels back to the VexRiscv processor and is handled accordingly deasserting the interrupt.

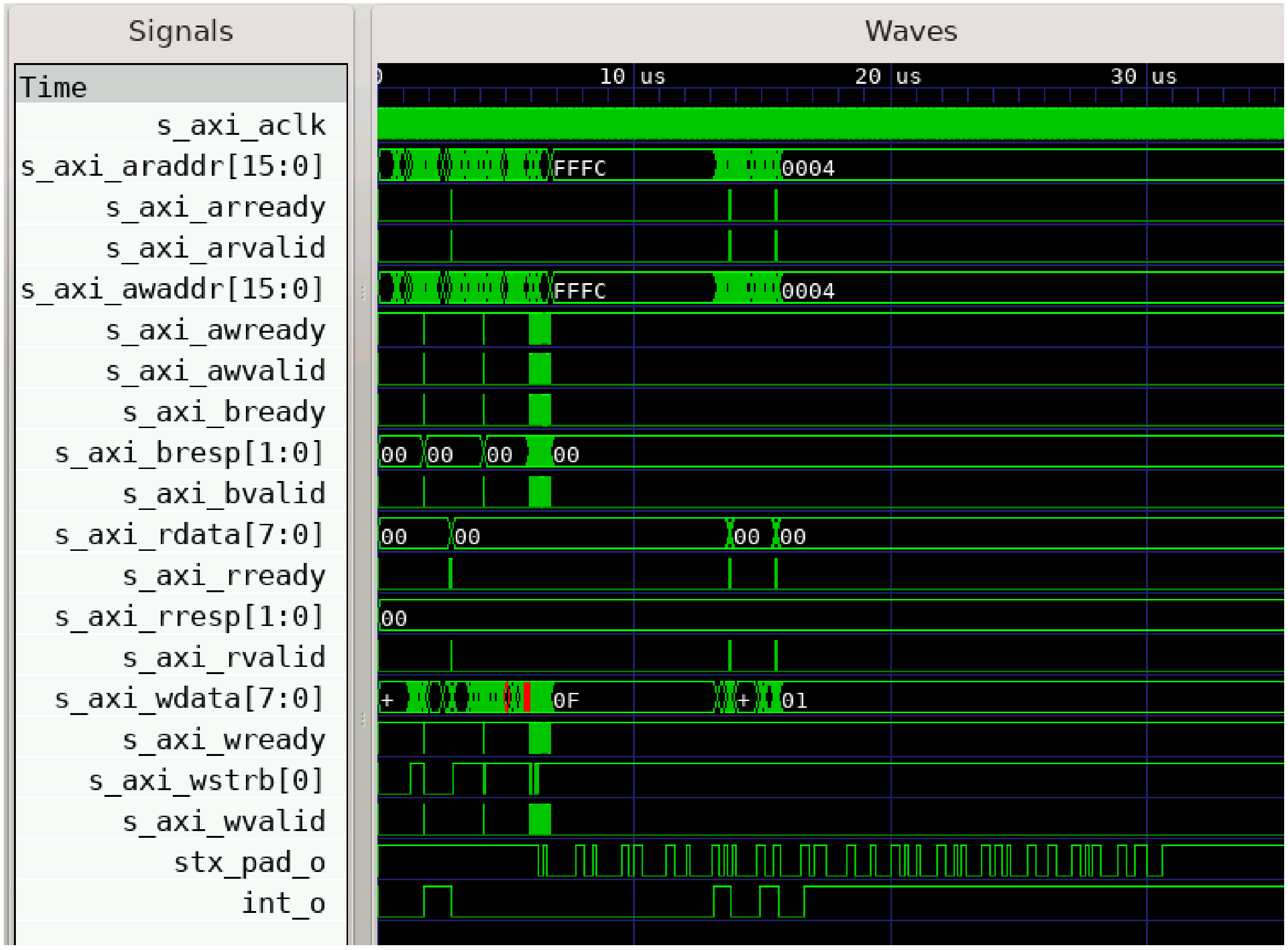


Figure 9. VexRiscv SoC Wavedump

Synthesis and PnR

Raptor Suite is armed with tools for **Synthesis** along with **Post and Route** capabilities and the generated post-synthesis and post-route and place netlists can be viewed and analyzed from within the Raptor. The generated bitstream can then be uploaded on an FPGA device to be utilized in hardware applications.

Test Bench

Test benches for all three of these configurations are packed with the Raptor Suite and can be simulated at ease with the bundled simulator. Details for the types of simulations can be found below: -

- **Base Variant** The test bench for this configuration is a bare-metal firmware written in RISCV ISA loaded directly on the CPU in .hex format, that performs some specific operations on an attached IO device.
- **Cached with MMU** This configuration of the processor is simulated via a firmware written in C++ that boots up this CPU with Linux Kernel.
- **Cached with MMU, PLIC and CLINT** This configuration is simulated via a bare-metal firmware written in RISCV ISA loaded directly on the CPU in .hex format, that performs specific operations on attached IOs while handling all IO interrupts as well as the inter core interrupts.

The bitstream for all of these designs, and much more, can be generated via the Raptor Suite and uploaded on an FPGA device to be used in hardware applications.

Revision History

Date	Version	Revisions
January 20, 2023	0.01	Initial version VexRiscv User Guide Document