

A Simple RISC-V processor in VHDL

Jesse E.J. op den Brouw*

The Hague University of Applied Sciences

November 15, 2021

Abstract

The RISC-V Instruction Set Architecture (ISA) is an open source instruction set for a processor. This means that anybody can create a processor that uses this instruction set. There are already processors available such as E2-core from SiFive. More freeware cores are available on several platforms (e.g. on GitHub). This document describes a basic RISC-V core in VHDL. The core can only execute the RV32I unprivileged instruction set. The processor incorporates a ROM, RAM and some simple I/O. It is targeted for implementation on an FPGA. It is tested on an Intel Cyclone V with a DE0-CV development board from Terasic. The GNU C-compiler for RISC-V is used for software development. Currently only simple C programs can be compiled and run. C++ is currently not supported.

This processor is not intended as a replacement for commercial available processors. It is intended as a study object for Computer Science students. The processor executes each instruction in one clock cycle except for reads from RAM which need two clock cycles. The processor has a simple, non-pipelined instruction decoder. Exceptions are currently not implemented. This will be for future development.

*J.E.J.opdenBrouw@hhs.nl

1 Introduction

This document describes the buildup of a simple, single clock cycle, one core, RISC-V processor, completely written in VHDL. The processor is able to run a simple compiled C-program. C++ is currently not supported. The processor can handle the RV32I Base Integer Instruction Set as set forward in “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA”. The RV32M Instruction Set is currently not supported, so multiplications and divisions have to be handled in software. The toolchain will take care of that when supplied with the correct parameters. The aim is to synthesize for a minimum clock frequency of 50 MHz.

This RISC-V processor consists of the following building blocks:

- The registers contain intermediate data for calculations.
- The ROM contains the program instructions and constant (read-only) data.
- The RAM contains read-write data (mutable data).
- The I/O is an interface with the outside world.
- The ALU is responsible for almost all computations in the processor.
- The PC is used to point to the currently executing instruction.
- The Address Decoder and Data Router is an interface between the memory (ROM, RAM, I/O) and the ALU.
- The Instruction Decoder decodes the currently executing instruction and provides control signals to other building blocks.

2 Registers

The processor consists of thirty-two 32-bit registers denoted by `x0` to `x31`. Internally, the registers use Big Endian format. Register `x0` (alias `zero`) is hardwired to all zeros. Writing this register has no effect. Reading this register returns all zero bits. Normally, the `x`-names are not used. Table 1 shows the names of the registers as they should be used.

3 ROM

The ROM consists of bytes and is only word addressable for instructions. The ROM is byte, half word and word addressable when reading constant data. Half word and word entries are in Little Endian format. When reading data from the ROM, halfword accesses must be on 2-byte boundaries and word accesses must be on 4-byte boundaries. This simplifies the decoding circuitry. The ROM returns undefined data if an access is not aligned. The ROM is

Table 1: *RISC-V registers and their purpose.*

Register	Name	Purpose	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–x7	t1–t2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporaries	Caller

instantiated in cells, which limits the size of the program. Rearranging half word and word data accesses in Big Endian format is handled by the ROM decoding unit.

Note: it is possible to put the ROM in the onboard RAM and supply a MIF file (Memory Initialization File), but then each read from the ROM would require 2 clock cycles. See Section 4.

4 RAM

The RAM consists of bytes and is byte, halfword and word addressable. Half word and word entries are in Little Endian format. The RAM itself is made up of word (i.e. 32-bit) entries and is instantiated with onboard RAM blocks. Due to this fact, halfword accesses are only permitted on 2-byte boundaries and word accesses are only permitted on 4-byte boundaries. The RAM returns undefined data if an access is not aligned. Writes will not take place if an access is unaligned. This simplifies the decoding circuitry. For the Cyclone V a maximum of 65536 words of RAM can be instantiated. Rearranging half word and word data accesses in Big Endian format is handled by the RAM decoding unit.

Note: the Cyclone V has 3,153,920 bits of RAM available. Because of the 32-bit entries a maximum of 2,097,152 (65536 x 32) bits can be instantiated. This is equivalent to 262,144 bytes.

Writing the RAM (byte, half word or word) requires 1 clock cycle. Reading the RAM (byte, half word, word) requires 2 clock cycles because the RAM output is buffered by a register. This is automatically handled by the processor.

5 I/O

Currently, the I/O consists of one 32-bit data input and one 32-bit data output. More I/O (timers/counter etc.) will be added in the future, but most I/O requires the use of interrupts (timer overflow etc.). Note that the I/O can only be accessed as words and the addresses must be on 4-byte boundaries. If not on a 4-byte boundaries, reads return undefined data whereas writes will not write data.

6 ALU

The Arithmetic and Logic Unit (ALU) handles all computations on data. It can add, subtract, do logic operations such as AND, OR and XOR, can shift data left or right, and sign extend byte and halfword data. Some operations require two registers, some only use one register. Furthermore the ALU is also used to determine if a conditional branch should be taken. Note that the RISC-V programmer's model does not incorporate status flags as some other architectures do. This requires some extra instructions when adding or subtracting double word (64-bit) data. The ALU is also used to compute the return address from unconditional function calls (JAL and JALR instructions). The data is in Big Endian format.

Note that the computation of jump target addresses is handled by the Program Counter (PC)

7 PC

The Program Counter contains the address of the currently executed instruction. The address is always on a 4-byte boundary although function calls (JAL and JALR instructions) can be on non 4-byte boundaries (the toolchain will always create 4-bytes boundaries). The PC (or rather the VHDL description of the PC) handles the address calculations of jumps and branches taken.

8 Instruction Decoder

The instruction decoder decodes the instruction supplied by the ROM as pointed by the PC. An instruction is 4-bytes wide and in Little Endian order. The instruction decoder provides all control signals for the ALU, the PC and Address Decoder. It incorporates a simple two-state Finite State Machine (FSM). Almost all instructions are executed in one clock cycle. Only reads from RAM require two clock cycles. The FSM takes care of that. The instruction decoder is non-pipelined. That simplifies the design.

9 Address Decoder and Data Router

The Address Decoder and Data Router routes reads and writes to the ROM (only reads), RAM and the I/O. The processor uses a linear address space for ROM, RAM and I/O. In the default setting, ROM starts at address 0x00000000 and the length is implementation-defined, it depends on the program. Unused ROM addresses return don't cares (in simulation) and in hardware the data returned is implementation-defined. Currently a maximum of 16384 bytes of ROM is supported. The RAM starts at address 0x20000000 and the length can be up to 262144 bytes, in powers of 2. The I/O starts at address 0xF0000000 and the length is implementation-defined.

When data is read, the data is collected from the accessed memory and put on an internal bus to the ALU. The ALU can perform sign extension (byte and halfword accesses) if needed. Please note that reading data from the RAM requires two clock cycles. Reading data from ROM and I/O requires one clock cycle.

10 Implemented instructions

All RV32I Unprivileged instructions are implemented with the exception of FENCE, ECALL and EBREAK instructions. These instructions act as a no-operation (NOP). This is because exceptions are not implemented.

11 Setting up the GNU C compiler

The processor can run simple compiled C-programs that are compiled using the GNU C-compiler for RISC-V. Besides that, a separate linker script is needed to setup the compiled code. Building the C compiler (for Linux) is straightforward:

1. You need a current GNU C-compiler installed.
2. You need the texinfo package. On Ubuntu et al. issue `apt install texinfo`
3. In your home directory, enter the command
`git clone --recursive https://github.com/riscv/riscv-gnu-toolchain`
4. Wait for the cloning to end (takes a long time, about 30 minutes on a Zbook G5 2020 with a 10 MB/s internet connection)
5. Change with `cd riscv-gnu-toolchain`
6. Make build directory with `mkdir build; cd build`
7. Check the current configuration with `../configure --help | grep abi`
It should say:

`--with-abi=lp64d` Sets the base RISC-V ABI, defaults to lp64d

The toolchain is currently configured for 64-bit RISC-V. That is not what we want.

8. Enter:

```
../configure --prefix=/opt/riscv32 --with-arch=rv32i --with-abi=ilp32
```

This will set the architecture to RV32I and the ABI to ilp32. This means that integers, long integers and pointers use 32-bit entries. The destination directory is /opt/riscv32

9. Now enter the make command as root: make

MAKE SURE TO ENTER THIS COMMAND AS root, because the toolchain is put in /opt/riscv32. This takes a long time (about 45 minutes on a Zbook G5). At some points the compilation seems to hang, but it is just compiling complicated C-files. By the way, you will see a lot of warnings.

10. Now that the toolchain is setup, we have to put the path into the \$PATH environment variable so enter: `export PATH=/opt/riscv32/bin:$PATH`

11. Check if the compiler is available: `riscv32-unknown-elf-gcc -v`

It should say something like:

Using built-in specs.

`COLLECT_GCC=riscv32-unknown-elf-gcc`

`COLLECT_LTO_WRAPPER=/opt/riscv32/libexec/gcc/riscv32-unknown-elf/11.1.0/lto-wrapper`

Target: riscv32-unknown-elf

Configured with: `/mnt/d/PROJECTS/RISCVDEV/riscv-gnu-toolchain/build/./riscv-gcc/configure --target=riscv32-unknown-elf --prefix=/opt/riscv32 --disable-shared --disable-threads --enable-languages=c,c++ --with-system-zlib --enable-tls --with-newlib --with-sysroot=/opt/riscv32/riscv32-unknown-elf --with-native-system-header-dir=/include --disable-libmudflap --disable-libssp --disable-libquadmath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=../../riscv-gcc --disable-multilib --with-abi=ilp32 --with-arch=rv32i --with-tune=rocket 'CFLAGS_FOR_TARGET=-Os -mmodel=medlow' 'CXXFLAGS_FOR_TARGET=-Os -mmodel=medlow'`

Thread model: single

Supported LTO compression algorithms: zlib

gcc version 11.1.0 (GCC)

12 Cloning the RISC-V project

Now we have to clone the RISC-V project. It incorporated the full Quartus Prime Lite project with the processor written in VHDL. It also incorporates some simple C program examples and a taylor-made program to convert a RISC-V executable to a VHDL table suitable for the ROM. Create a working directory (and change to that directory) and issue the command:

```
git clone --recursive https://github.com/jesseopdenbrouw/riscv-minimal
```

In the created directory, you will see the following directories:

CODE – Sample software programs

DOCS – Documentation

HARDWARE – the VHDL description

OLD – yes, really old files for backup

Change directory to CODE. Now enter the command `make`. It will compile all programs and the taylor-made conversion program. To clean up the programs, issue the command `make clean`.

Next, start your Quartus Prime Lite software and open the project in the HARDWARE directory. Now start a build by clicking on the play-symbol. It should compile a standard setting (this takes a long time). When finished, you can download the FPGA contents to the DE0-CV board.

Next, dive in one of the directories and copy the file with `.vhd` extension to the directory containing the VHDL description under the name `processor_common_rom.vhd`.

The start Quartus and start the compilation. After a successful compilation, you can program the Cyclon V on a DE0-CV board.