A Minimal RISC-V processor in VHDL

Jesse E.J. op den Brouw*

The Hague University of Applied Sciences

November 30, 2021

https://github.com/jesseopdenbrouw/riscv-minimal

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 documents describes a basic 32-bit 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 standard processor executes each instruction in one clock cycle because the ROM is realized with cells, except for reads from RAM which need two clock cycles. The alternative processor executes each instruction in two clock cycles because the ROM is inferred using onboard RAM, except for ROM and RAM reads which need an extra clock cycle. The processor has a simple, non-pipelined instruction decoder. Exceptions are currently not implemented. This will be for future development.
This is work in progress. Things will certainly change in the future.

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

Contents

1 Introduction

This document describes the buildup of a simple, single/dual 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 and registers.
- The Instruction Decoder decodes the currently executing instruction and provides control signals to other building blocks.

A block diagram is shown in Figure 1.

There are two versions available. The standard version uses Logic Cells to implement the ROM. This will limit the program size but each ROM access (instruction and data) only requires one clock cycle. The alternative version uses preprogrammed onboard RAM blocks to implement the ROM. Larger programs are possible but each ROM access (program and data) requires two clock cycles.

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 but may be handy when simulating the designs. Table 1 shows the names of the registers as they should be used.

A register can be written to, and two register can be selected for data and base address.

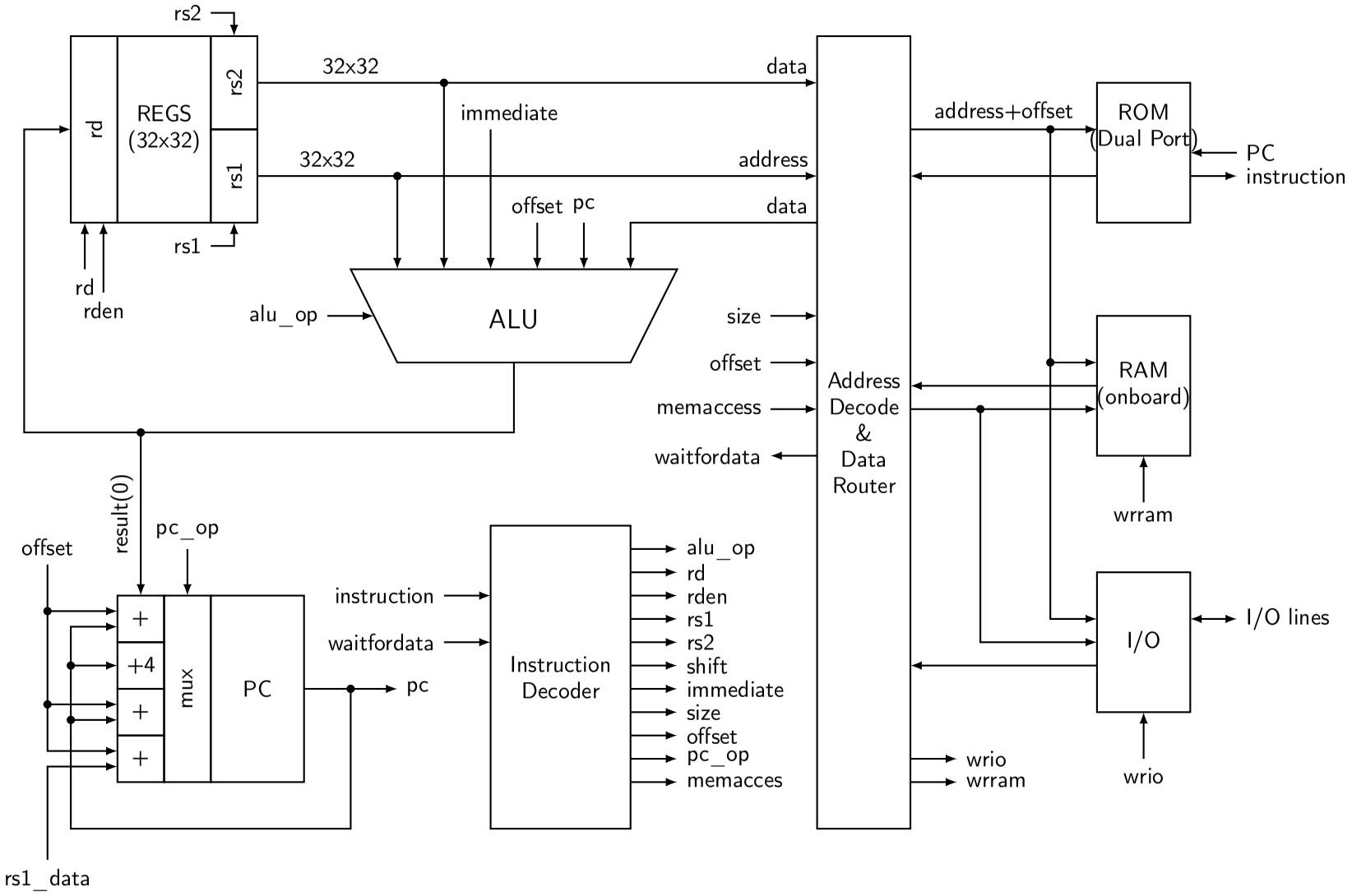

Figure 1: *The complete RISC-V MCU.*

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

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 Litte 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 standard processor instantiates the ROM in cells, which limits the size of the program. The alternative processor instantiates the ROM in onboard RAM, so bigger programs are possible. Rearranging half word and word data accesses in Big Endian format is handled by the ROM decoding unit.

Note: the alternative processor uses onboard RAM to simulate ROM. Because of this, each read from ROM (instruction and data) requires two 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 of 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 en 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 and conditional jump (JAL, JALR en Bxx 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, the Address Decoder and the register file.

In the standard processor, a simple two-state Finite State Machine (FSM) is used. Almost all instructions are executed in one clock cycle. Only reads from RAM require two clock

cycles. The FSM takes care of that.

In the alternative processor, a simple three-state FSM is used. All instructions require two clock cycles to be fetched and executed, and an extra clock cycle is needed when reading RAM or ROM. The FSM takes care of that.

The instruction decoder is non-pipelined. That simplifies the design but slows down the computational speed.

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 32-bit linear address space for ROM, RAM and I/O. In the default setting, ROM starts at address 0x00000000 and the length is 16 kB. Unused ROM addresses return don't cares (in simulation) and in hardware the data returned is implementation-defined. The RAM starts at address 0x20000000 and the length can be up to 262,144 bytes, in powers of 2. Default is 16 kB. The I/O starts at address 0xF0000000 and the length is 16 kB by default.

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 and ROM requires an extra clock cycle. Reading data from I/O requires one clock cycle.

10 Stack pointer

The stack pointer is fully implemented although the ISA does not provided pushes and pops. The stack pointer is used to allocate local variables and is updated with each allocation and deallocation. As usual, the stack grows downwards (to lower addresses) on allocations and upwards (to higher addresses) on deallocations. Therefore the stack pointer is set to the highest RAM address + 1 on startup (which is 0x20004000 by default). The ISA postulates that the stack is aligned on 16-byte boundaries.

11 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.

12 The FPGA

For this project, we use the Cyclone V FPGA from Intel (formerly Altera). See https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v.html. The used Cyclone V is the 5CEFA4F23C7 which has 18480 cells available. It has 3080 kb

of onboard RAM bits available which are partially used for RAM and ROM (in the alternative processor). Depending on the program and used resources, the compiled RISC-V processor uses about 2000-2500 cells (about 12 % - 15 %). This FPGA is mounted on a Terasic DE0-CV board, see http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=921. For downloading the program file, the onboard USB-Blaster is used.

13 Simulation

The designs can be simulated fully, using QuestaSim Intel Starter or ModelSim Intel Starter. You need a (free) license for QuestaSim. During simulation, all essential signals can be viewed, as is the RAM. The RAM is viewed as 32-bit entries, so we need to do some manual calculations to correctly find byte, halfword and word accesses. Simulation can be started from Quartus.

14 Setting up the GNU C compiler for RISC-V

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 or your Linux box.
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 to the directory with

   ```
   cd riscv-gnu-toolchain
   ```

6. Make the `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 ipl32. 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   -mcmodel=medlow' '↩
       ↪CXXFLAGS_FOR_TARGET=-Os   -mcmodel=medlow'
    Thread model: single
    ```

```
Supported LTO compression algorithms: zlib
gcc version 11.1.0 (GCC)
```

15 Cloning the RISC-V project

Now we have to clone the RISC-V project. It incorporates 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 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`. Make sure the RISC-V C compiler is in your path environment variable. 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.

To test one of the programs, change directory to 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`. Now start Quartus and start the compilation. After a successful compilation, you can program the Cyclone V on a DE0-CV board.

16 Register subset

It is possible to compile the toolchain to only use register `x0` to `x15`. This is called the RISC-V E extension. As a positive side effect, the register file can be cut down from 32 registers to 16 registers. This will lower the cell count and possible speed up the device. A negative side effect is that the pressure on register allocation is higher, possibly increasing instruction count when saving registers on the stack.

To configure the GNU C compiler for the E extension issue the command:

```
../configure --prefix=/opt/riscv32 --with-arch=rv32e --with-abi=↩
   ↪ilp32e
```

and build the compiler.

Now compile a C program with:

```
riscv32-unknown-elf-gcc -O0 -g -o string string.c -Wall -T ../↩
   ↪ldfiles/riscv.ld -march=rv32e -mabi=ilp32e -nostartfiles --↩
   ↪specs=nano.specs ../crt/startup.c
```

Make sure to use `-march=rv32e` and `-mabi=ilp32e`.

17 Compiling a C program by hand

Note: only very simple C programs can be compiled for the processor at this time. We tested some simple looping (with `for`) and reading/writing the I/O. We did test the use of the C library (malloc et al, floats and double calculation, some trigonometry functions from the mathematical library), but more tests are needed.

Compiling a program requires the following steps:

- In the program directory `CODE`, create a new directory and change to that directory.
- Create a C program file, we assume `flash.c`.
- Now issue the command:

  ```
  riscv32-unknown-elf-gcc -O2 -g -o flash flash.c -Wall -T ↩
     ↪../ldfiles/riscv.ld -march=rv32i -nostartfiles --specs↩
     ↪=nano.specs ../crt/minimal.S
  ```

  We supply our own linker file (`-T ../ldfiles/riscv.ld`) and we supply our own startup file (`../crt/minimal.S`). Make sure to use `-nostartupfiles`↩ ↪ otherwise the default startup file will be linked and errors will report. There a three startup files:

  - `empty.S` – Empty startup file only providing the entry symbol. Can be used with assembler programs.
  - `minimal.S` – Loads the global pointer and stack pointer and calls `main`. On return of `main`, it waits in an endless loop. Can be used with minimalistic C programs.
  - `startup.c` – Full support for C programs.

- Next issue the command:

  ```
  riscv32-unknown-elf-objcopy -O srec flash flash.srec
  ```

  This will create an S-record file in Motorola hex-format.

- Next issue the command:

  ```
  ../bin/srec2vhdl -wf flash.srec flash.vhd
  ```

  This will create a VHDL file with the ROM encoded as 32-bit Little Endian quantities. Note: the taylor-made `srec2vhdl` has to be compiled before. See Section 15.

- Next issue the command:

  `cp flash.vhd ../../HARDWARE/riscv/processor_common_rom.vhd`

  This will copy the VHDL file to the RISC-V processor ROM file.

- Now start the compilation of the VHDL code in Quartus Prime Lite and program the compiled file. This file has the extension `.sof`. See Figures 2 to 4.

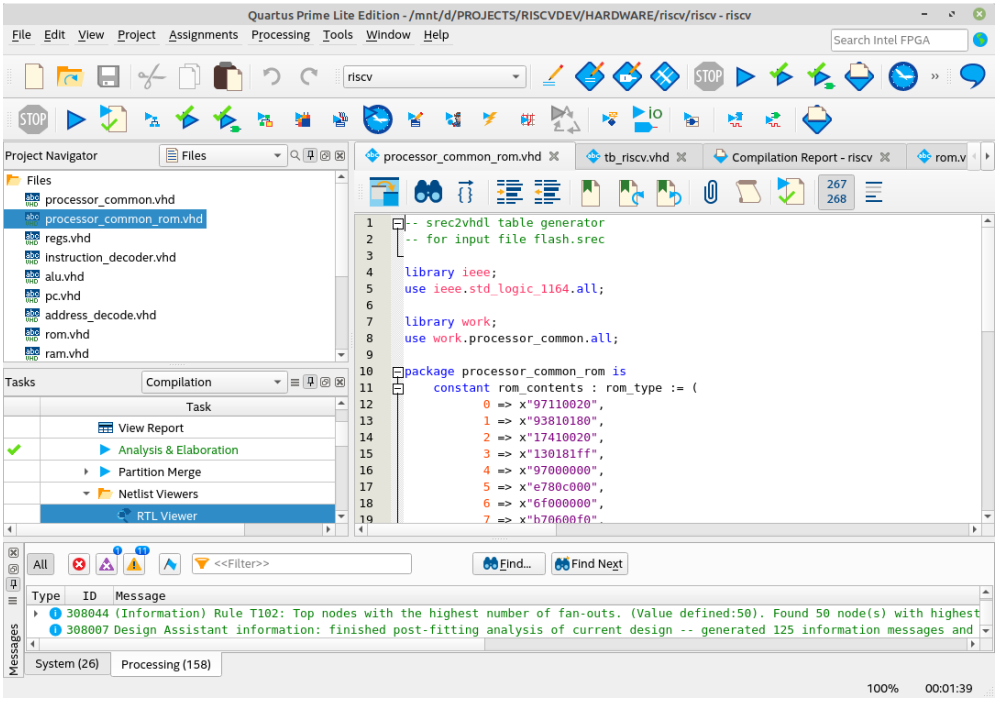

Figure 2: *Image of the Quartus project (1).*

18 VHDL files

Both descriptions are composed of the following files:

- `processor_common.vhd` – Common types and constants.
- `processor_common_rom.vhd` – Description of the ROM contents.
- `address_decode.vhd` – The address decoder and data router to the memory (ROM, RAM, I/O).
- `alu.vhd` – Description of the ALU.
- `pc.vhd` – Description of the Program Counter.
- `instruction_decoder.vhd` – The instruction decoder.
- `regs.vhd` – Description of the register file.
- `rom.vhd` – Description of the ROM.

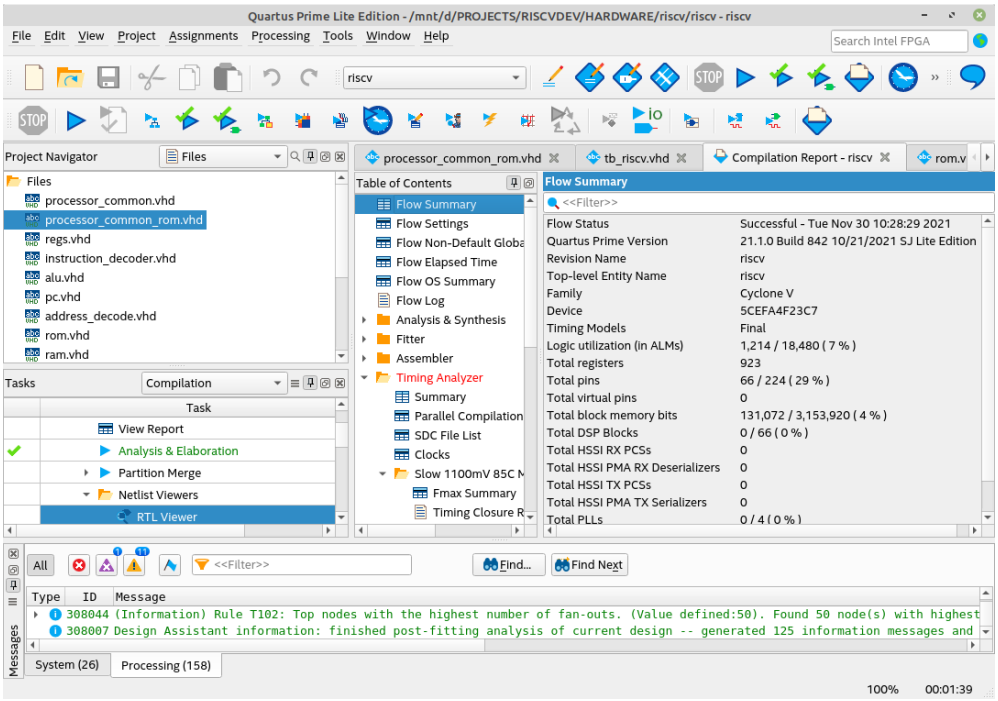

Figure 3: *Image of the Quartus project (2).*

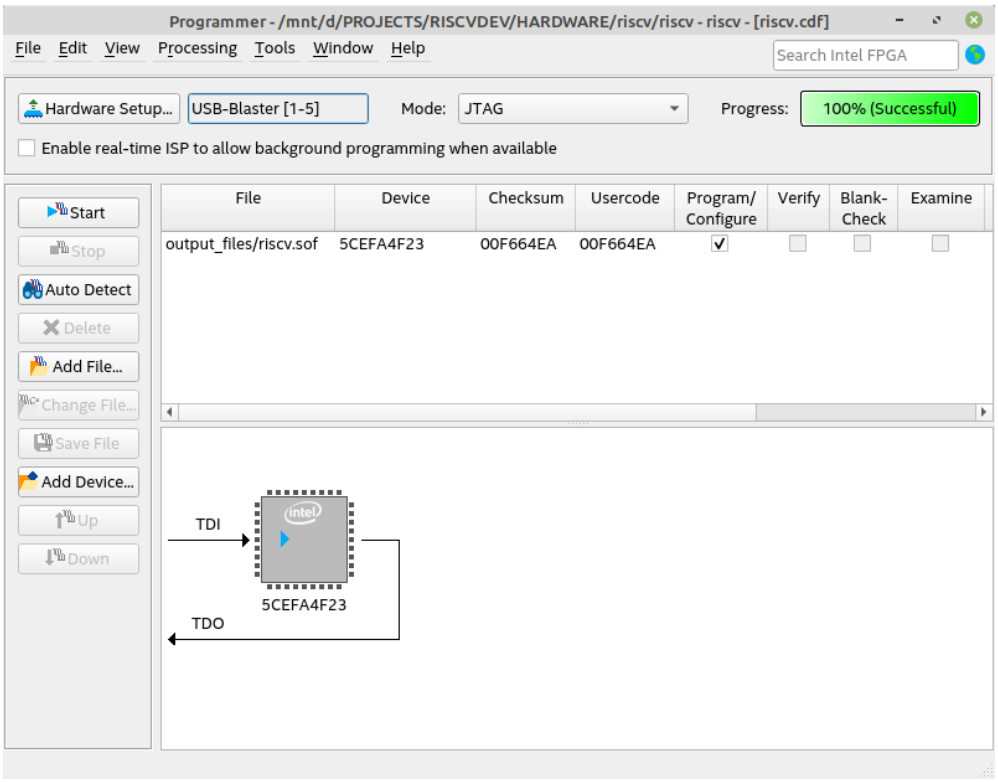

Figure 4: *Image of the programmer.*

- `ram.vhd` – Description of the RAM interface between the address
- `ram_inst.vhd` – Description of the onboard RAM. Quartus detects this and cre-

ates the RAM in onboard RAM blocks.

- `io.vhd` – Description of the I/O
- `riscv.vhd` – Top-level description of the processor.
- `riscv.sdc` – Constraints file. Sets the target clock frequency.
- `tb_riscv.vhd` – VHDL testbench to simulate the design.
- `tb_riscv.do` – QuestaSim/Modelsim command script.

The alternative version as one more file:

- `rom_inst.vhd` – Description of the ROM. The ROM contents will be placed in onboard, initialized RAM blocks. The file `rom.vhd` is then a frontend between the address decoder and data router and the onboard ROM.

19 srec2vhdl

This is a homebrew utility to convert a Motorola S-record file into a VHDL file suitable for inclusion of the processor. The program is called with:

```
srec2vhdl [-fbwhqv] [-i <arg>] inputfile [outputfile]
```

`inputfile` is the S-record file, created by the `objdump` program. `outputfile` is the VHDL outputfile. When omitted, `stdout` is used. There are a number of options:

- `-f` makes a full output that directly can be used. If not used, only the ROM table contents itself is produced.
- `-w` ROM contents is in words (32 bits).
- `-h` ROM contents is in halfwords (16 bits).
- `-b` ROM contents is in bytes (8 bits).
- `-v` Verbose output
- `-q` Quiet output, only error messages are displayed.
- `-i <arg>` indents each line with `<arg>` spaces.

20 Software programs

In de `CODE` directory, there are a number of software programs available:

- `ldfiles` – contains the linker scripts.
- `crt` – contains the startup files.
- `bin` – contains the binary of `srec2vhdl`.
- `add64` – simple 64-bit addition.

- `assembler` – a simple assembler program.
- `ctest` – scratch C-program test.
- `double` – some floating point double computations (seems to work).
- `flash` – flash the DE0-CV board leds (works on the board)
- `float` – some floating point float computations (seems to work).
- `global` – test for globals and local statics with initialization (seems to works).
- `malloc` – example to test `malloc` and friends (seems to work).
- `mult` – integer multiplication (works).
- `shift` – shifts (works)
- `string` – some string functions (seems to work).
- `syscalls` – implementing stubs for common system calls (seems to work). Note: `sbrk` works for `malloc`, needs more testing.
- `testio` – simple program that copies the input (switches) to the output (leds) (works on the board).
- `trig` – some float trigonometry functions (seems to work).
- `ioadd` – adds the lower 5 switches to the upper 5 switches and displays the result on the leds. Tests addition, shifting and I/O.

Note: we use a lot of the `volatile` keyword to emit the variables to RAM for easy inspection.

Note that the floating point programs loads (huge) functions from the C library and possibly creates a binary that is to large to fit in the ROM. In that case, the linker will issue an error and does not build the binary.

21 Address ranges and memory sizes

By default, the ROM starts at address 0x00000000 and and has a size of 16 kB (4 k words). The Program Counter starts at address 0x00000000. The RAM starts at address 0x20000000 and has a size of 16 kB (4 k words). The stack pointer is set to one address above the last RAM byte, by default at 0x20004000. The I/O starts at address 0xf0000000 and has a size of 16 kB (4 k words).

The ROM may be moved to another start location. The Program Counter is started at the correct address. The placement of the ROM is in 256 MB intervals, which are the 4 most significant bits of a 32-bit address. The same holds for the RAM and the I/O. To move the ROM, open the VHDL file `processor_common.vhd` and go down to the end of the file. There you will see the following lines:

```
-- The highest nibble (4 bits) of the ROM, RAM and I/O
```

```
-- This will set the memories at 256 MB intervals
constant rom_high_nibble : std_logic_vector(3 downto 0) := x"0";
constant ram_high_nibble : std_logic_vector(3 downto 0) := x"2";
constant io_high_nibble : std_logic_vector(3 downto 0) := x"F";
```

Change the start locations of the memories by changing the constants. Make sure the memories do not overlap. To change the sizes of the memory, look for the lines as shown below:

```
-- The RAM
-- NOTE: the RAM is 4x byte (8 bits) size, supporting
--       32-bit Big Endian storage,
--       so we have to recode to support Little Endian.
--       Set ram_size_bits as if it were bytes
-- NOTE: ram_size_bits must be <= 16
constant ram_size_bits : integer := 14;
constant ram_size : integer := 2**(ram_size_bits-2);
-- The type of the RAM block, there are 4 blocks instantiated
type ram_type is array (0 to ram_size-1) of std_logic_vector(7 ↩
   ↪downto 0);

-- The ROM
-- NOTE: the ROM is word (32 bits) size.
-- NOTE: data is in Little Endian format (as by the toolchain)
--       for halfword and word entities
--       Set rom_size_bits as if it were bytes
-- NOTE: rom_size_bits must be <= 16
constant rom_size_bits : integer := 14;
constant rom_size : integer := 2**(rom_size_bits-2);
type rom_type is array(0 to rom_size-1) of std_logic_vector(31 ↩
   ↪downto 0);
-- The contents of the ROM is loaded by processor_common_rom.vhd

-- The I/O
-- NOTE: the I/O is word (32 bits) size, Big Endian
--       there is no need to recode the data
--       The I/O can only handle word size access
--       Set io_size_bits as if it were bytes
constant io_size_bits : integer := 3;
constant io_size : integer := 2**(io_size_bits-2);
type io_type is array (0 to io_size-1) of data_type;
```

Set the memories in address bit lengths. So, `16` means $2^{16} = 65,536$ bytes (524,288 bits). This can be done for ROM and RAM. The standard processor uses Logic Cells to implement the ROM, so only small values are applicable. The alternative processor uses onboard RAM so higher values are possible. Please note that in this case both ROM and RAM bits may not exceed 3,153,920 bits.

Next, open de linker script in `<...>/ldfiles/riscv.ld`.

Find the lines as showed below:

```
MEMORY
{
   ROM (rx)    : ORIGIN = 0x00000000, LENGTH = 16K
   RAM (rwx)   : ORIGIN = 0x20000000, LENGTH = 16K
   IO (rw)     : ORIGIN = 0xf0000000, LENGTH = 16K
}
```

Change the start addresses at the top op the file and changes the sizes if needed. Then recompile your programs.

22 Future plans

Some future plans:

- Implement exceptions and interrupts in general. This will bring the possibility to add more I/O, such as timers. This needs implementation of the "Zicsr" standard.
- Implement the M standard: multiplier and divider. Multiplication can be achieved with the onboard multipliers but division has to be handled with a multi-cycle FSM.
- Implement more General Purpose I/O (pins), with data direction registers. On the Cyclone V this is an issue, since the tri-state buffers must be in the top level of the design. This makes it hard to implement this processor as part of greater design.
- Implement more functions of the standard library. Now only a few functions work.

23 Author's note

I managed to create this basic RISC-V processor within one week, including compiling the GNU C compiler and the created C program examples. Of course, this is not the fastest core available, but it gives a good example on designing a RISC-V processor yourself. Next in line is to make the standard C library work. In the mean time, files will change, so be sure to grab the latest GitHub repository clone.