A Minimal RISC-V processor in VHDL

Jesse E.J. op den Brouw*

The Hague University of Applied Sciences

December 31, 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 with the use of Quartus Prime Lite 21.1 and QuestaSim Intel Starter Edition 2021.2. 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.

There are two versions available. The standard processor executes each instruction in two clock cycles because the ROM is inferred using onboard RAM. ROM and RAM reads need an extra clock cycle. The processor has a simple, non-pipelined instruction decoder. The pipelined processor has a simple two-stage pipeline and executes each instruction in one clock cycle except for jump/branches taken, which requires two clock cycles. ROM and RAM reads need an extra clock cycle. This processor also has a basic Control and Status Registers (CSR) set.

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, 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 will be handled in software. The C-compiler will take care of that when supplied with the correct parameters. The aim is to synthesize for a minimum clock frequency of 50 MHz. The processor utilizes some simple I/O effectively making it a microcontroller.

There are two version: a non-pipelined version (`riscv`) which requires two/three clock cycles to complete an instruction and a pipelined version with a basic implementation of the Control and Status Registers (`riscv-pipe-csr`). This processor also requires two clock cycles to complete an instruction, but the next instruction is fetched while the current instruction is executed. Jumps, calls and branches taken requires two clock cycles because an new instruction has to be fetched. Also, an extra clock cycle is needed when reading ROM or RAM (data) because ROM and RAM is implemented using onboard RAM block (which are buffered with an output register). This processor has a basic Control and Status Registers set, offering CYCLE and CYCLEH (completed clock cycles), TIME and TIMEH (time since last reset, in microseconds) and INSTRET and INSTRETH (number of retired instructions).

The non-pipelined processor executes at top 0.5 MIPS/MHz because each instruction fetch and execution requires a clock cycle (2 clock cycles total). The pipelined processors execute at top 1 MIPS/MHz (jump and branches taken and reading ROM/RAM for data requires an extra clock cycle).

2 The processor

Both RISC-V processors consist of one *core* and 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.

- (only for `riscv-pipe-csr`) The Control and Status Registers contains a basic set of register. Currently only RDTIME(h), RDCYCLE(h) and RDINSTRET(h).

A block diagram of the non-pipelined processor is shown in Figure 1.

2.1 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.

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

2.2 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, half word 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 processor instantiates the ROM in onboard RAM. Rearranging half word and word data accesses in Big Endian format is handled by the ROM decoding unit.

Note: the Cyclone V has 3,153,920 bits of onboard 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.

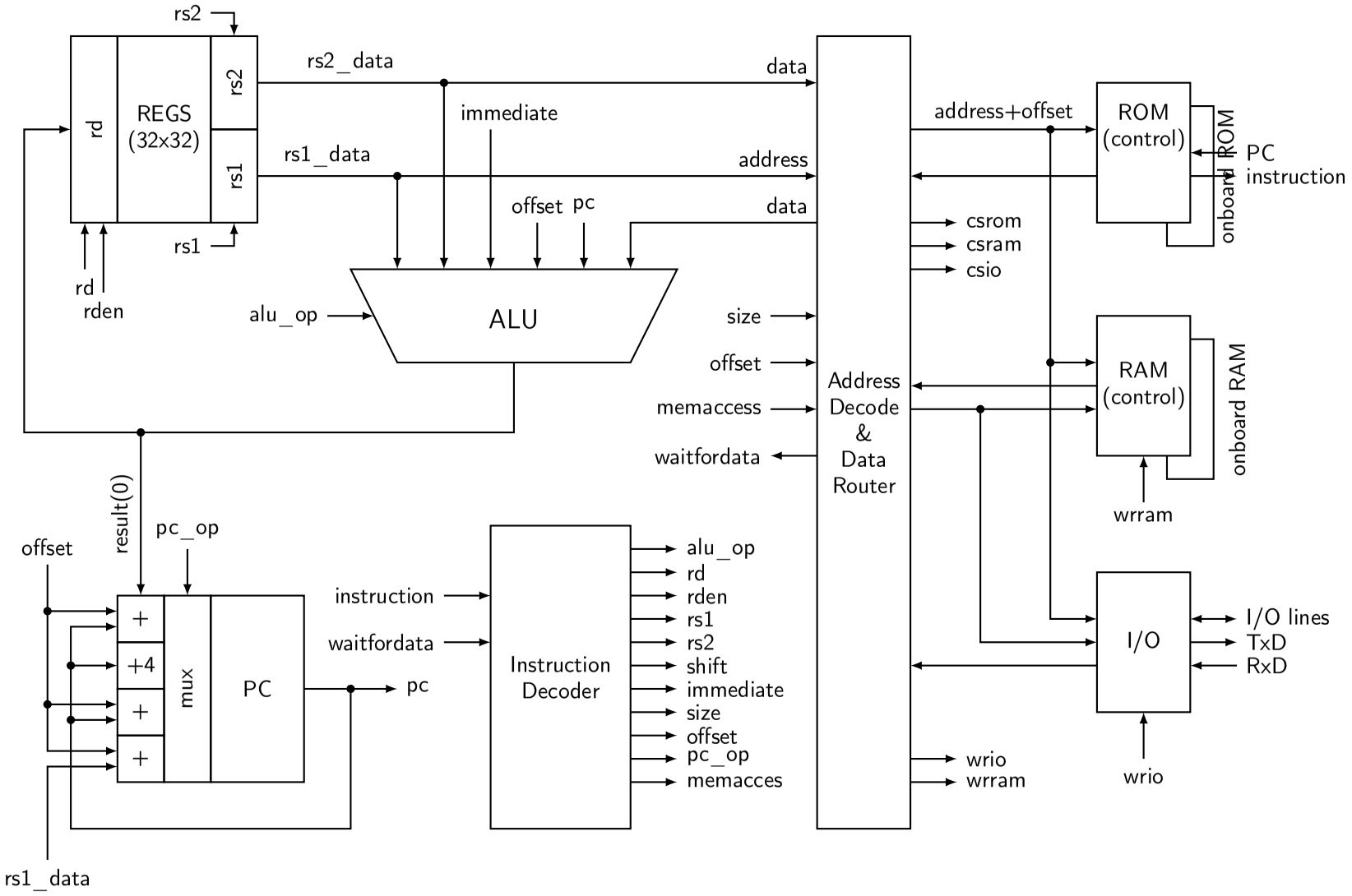


Figure 1: *The complete, non-pipelined RISC-V MCU.*

2.3 RAM

The RAM consists of bytes and is byte, half word 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, half word 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 onboard 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.

Reading the RAM (byte, half word, word) requires an extra clock cycles because the RAM output is buffered by a register. This is automatically handled by the processor.

2.4 I/O

Currently, the I/O consists of one 32-bit data input and one 32-bit data output, and a simple UART. More I/O (timers/counter, ...) 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 and not word size reads/writes, reads return undefined data whereas writes will not write data. Reading and writing I/O requires one clock cycle.

The UART can transmit and receive data at 8 bits, no parity and 1 stop bit only (8N1). Tested speeds are 9600 bps and 115200 bps. Several status flags are implemented to guide transmission.

2.5 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 half word 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. The ALU is the only building block that can write registers.

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

2.6 PC

The Program Counter contains the address of the currently executed instruction (non-pipelined) or the next instruction (pipelined). 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 C compiler 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.

2.7 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, the CSR and the register file.

In the standard processor, a simple three-state FSM is used, see Figure 2. All instructions require two clock cycles to be fetched and executed, and an extra clock cycle is needed when reading RAM or ROM.

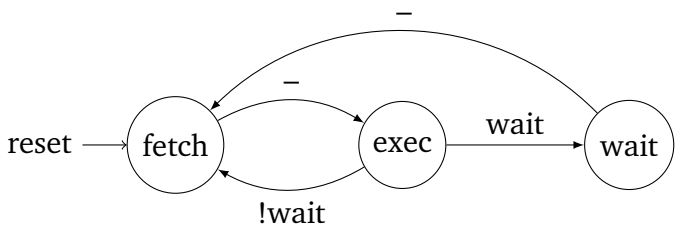


Figure 2: *FSM of the instruction decoder of the non-pipelined processor.*

The pipelined processor uses a simple three-state FSM, see figure 3. The PC points to +4 of the currently executing instruction. If a jump or branch taken occurs, the FSM inserts a penalty because the PC has to be loaded with the correct value and a new instruction must be fetched. Reading RAM and ROM requires an extra state. Note that penalty and wait cannot occur at the same time.

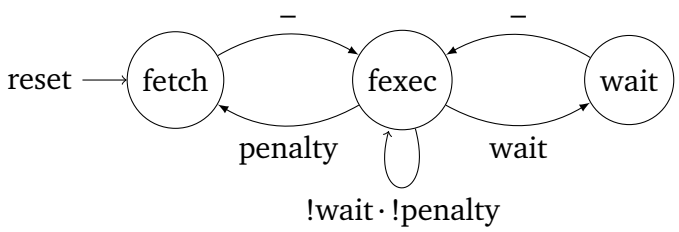


Figure 3: *FSM of the instruction decoder of the pipelined processor.*

2.8 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 64 kB. Unused ROM addresses return 0x0000000. The RAM starts at address 0x20000000 and length is 32 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 half word accesses) if needed. Please note that reading data from the RAM and ROM requires an extra clock cycle. Reading data from I/O requires no extra clock cycle.

2.9 Control and Status registers

The RISC-V describes a set of 4096 control and status register in a separate address space. Currently only a basic set is implemented:

- CYCLE and CYCLEH – these 32-bit registers form a 64-bit value that contains the counted clock cycles since the last reset.
- TIME and TIMEH – these 32-bit registers form a 64-bit value that contains the counted wall-clock microseconds since the last reset.
- INSTRET and INSTRETH – these 32-bit registers form a 64-bit value that contains the counted retired instructions since the last reset.

Note that these registers are read-only. Writes are ignored. Other registers may be added.

2.10 Stack pointer

The stack pointer is fully implemented although the ISA does not provide 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 0x20008000 by default). The ISA postulates that the stack is aligned on 16-byte boundaries.

2.11 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. The pipelined version with CSR (`riscv-pipe-csr`) can handle the RV32 “Zicsr” instruction set.

3 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 ALMs available. It has 3080 kb of onboard RAM bits available which are used for RAM and ROM. Depending on the

program and used resources, the compiled RISC-V processor uses about 2000-2500 ALMs (about 12 % - 15 %) and 786,432 bits of internal memory (25 %). The clock frequency is 50 MHz which is sufficient for all program examples. This FPGA is mounted on a Terasic DE0-CV board. The board has 10 switches, 4 push buttons, 1 reset push button, 10 leds, and 6 seven-segment displays. 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.

It is possible to add the Quartus' Signal Tap (embedded) Logic Analyzer. Follow the instructions on https://people.ece.cornell.edu/land/courses/ece5760/Quartus/Signal_tap.html.

4 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, half word and word accesses. Simulation can be started from Quartus.

5 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, linker script and startup files
`DOCS` – Documentation
`HARDWARE` – the VHDL descriptions
`OLD` – yes, really old files for backup

Change directory to `CODE`. Make sure the RISC-V C compiler is available (see Section 6) and 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.

6 Setting up the GNU C compiler for *this* 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 and startup file are needed to setup the compiled code. It is possible to set up the C library for multiple RISC-V architecture versions and select a version during compilation. Building the C compiler (from Linux) is straightforward:

1. You need a current GNU C-compiler installed or your Linux box. You also need all essential building tools:

   ```
   apt install autoconf automake autotools-dev curl python3 ↩
      ↪libmpc-dev libmpfr-dev libgmp-dev gawk build-essential↩
      ↪ bison flex texinfo gperf libtool patchutils bc zlib1g↩
      ↪-dev libexpat-dev
   ```

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 ../configure --prefix=/opt/riscv32 --with-arch↩
   ↪=rv32i --with-abi=ilp32 --with-multilib-generator=``↩
   ↪rv32i-ilp32--;rv32e-ilp32e--''
```

   This will set the architecture to RV32I and RV32E (reduced registers), and the ABI to ilp32 and ilp32e (reduced registers). This means that integers, long integers and pointers use 32-bit entities. 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/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 --↩
   ↪enable-multilib --with-multilib-generator='rv32i-ilp32↩
   ↪--;rv32e-ilp32e--' --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)
```

6.1 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 ALM (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.

Using the above recipe, the toolchain is set up for both RV32I and RV32E. You need the specify the architecture and ABI during compile time of the RISC-V programs.

Now compile a C program with:

```
riscv32-unknown-elf-gcc -O2 -g -o flash flash.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`.

7 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/startup.c
  ```

  We supply our own linker file (`-T ../ldfiles/riscv.ld`) and we supply our own startup file (`../crt/startup.c`). 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.
  - `minmal.S` – Provides the entry symbol, loads the global pointer and stack pointer. Can be used with assembler programs.
  - `startup.S` – Provides the entry symbol, 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. Can be used with `malloc` et al.

- 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 5.

- 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 4 to 6.

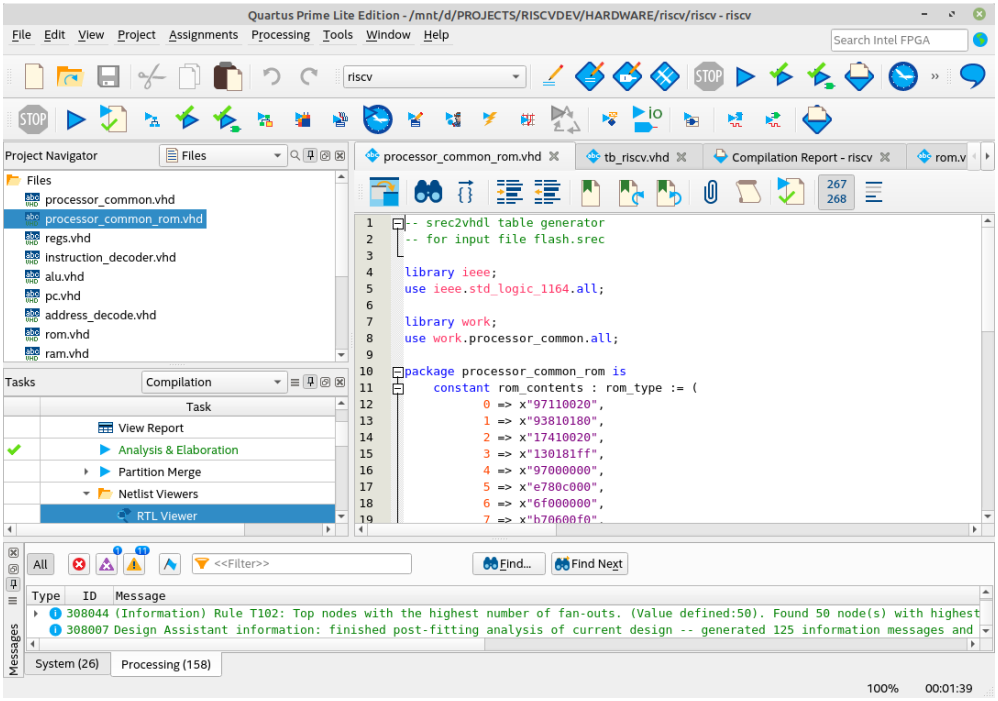


Figure 4: *Image of the Quartus project (1).*

8 VHDL files

The VHDL description is composed of the following files:

- `processor_common.vhd` – Common types and constants.

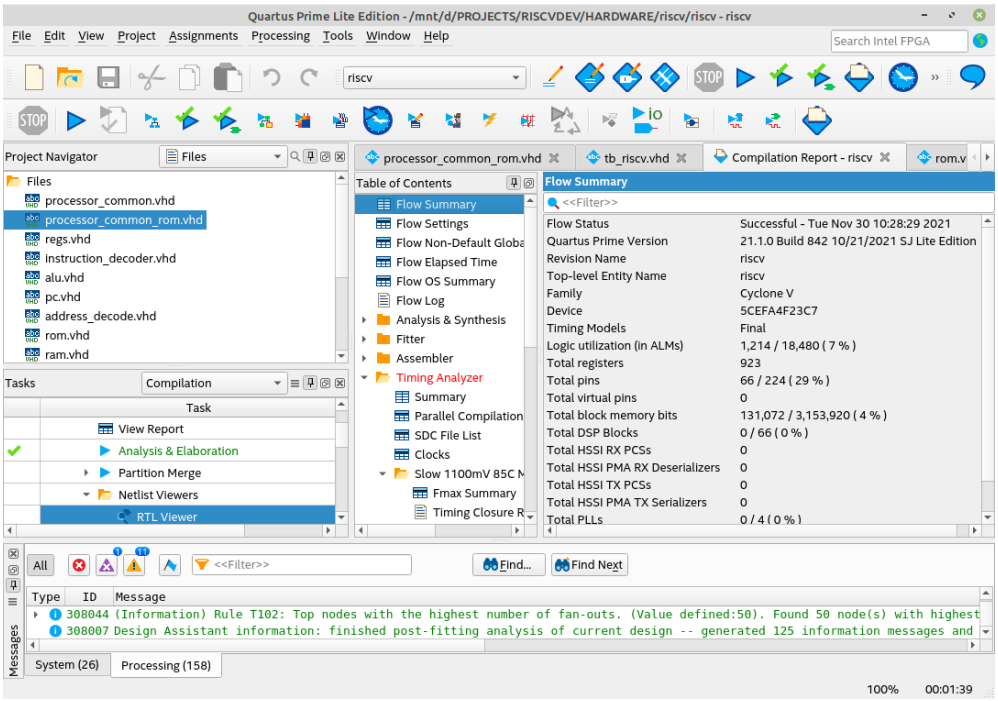


Figure 5: *Image of the Quartus project (2).*

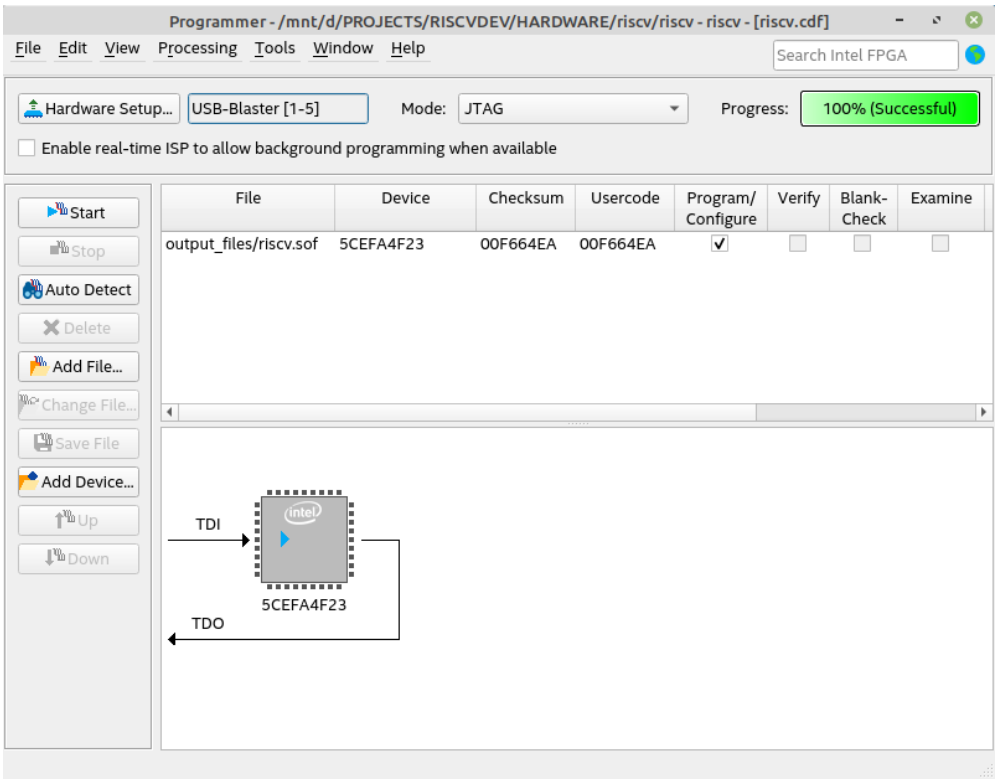


Figure 6: *Image of the programmer.*

- `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 interface between the core and the ROM contents.
- `rom_inst.vhd` – Interface between the ROM contents and the ROM interface. The ROM contents itself is in the file `processor_common_rom.vhd`. The ROM contents will be placed in onboard, initialized RAM blocks.
- `ram.vhd` – Description of the RAM interface between the address decoder and data router and the onboard RAM.
- `ram_inst.vhd` – Description of the onboard RAM. Quartus detects this and creates the RAM in onboard RAM blocks.
- `io.vhd` – Description of the I/O. It contains a 32-bit input register, a 32-bit output register and a simple 8-bit UART.
- `csr.vhd` – Description of the Control and Status registers (only for `riscv-pipe-csr`).
- `riscv.vhd` – Top-level description of the processor. Connects are the building blocks to a viable 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.

9 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 half words (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.

Note: uninitialized ROM contents is emitted as don't care, Quartus will set this to 0 by default.

10 Software programs

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

- `ldfiles` – contains the linker scripts. There is one script:
  - `riscv.ld` – default linker script: ROM = 64 kB, RAM = 32 kB, I/O = 16 kB.
- `crt` – contains the startup files.
- `bin` – contains the binary of `srec2vhdl`.
- `add64` – simple 64-bit addition.
- `assembler` – a simple assembler program.
- `clock` – a simple clock using the CSR TIME and TIMEH registers to fetch the time since last reset. Can only be used with the pipelined processor. Works on the board.
- `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).
- `ioadd` – adds the lower 5 switches to the upper 5 switches and displays the result on the leds. Tests addition, shifting and I/O (works on the board).
- `malloc` – example to test `malloc` and friends (seems to work).
- `monitor` – simple monitor program. Works on the board. Uses strings, USART, RAM, ROM, I/O and `sprintf`.
- `mult` – integer multiplication using the C library (works).
- `qsort_int` – sorts an integer array using the `qsort` C library function (seems to work).
- `riemann_left` – calculates the Riemann Left Sum of $\sin^2 x$ from 0 to 2π.
- `shift` – shifts (works)

- `sprintf` – prints integers, floats/doubles to a string. This is a big binary (seems to work).
- `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).
- `usart` – simple USART program (works on the board).
- `usart_sprintf` – simple program that prints an integer, a pointer, a float and a double to the terminal, this is a big binary (works on the board).

Note: we use a lot of the `volatile` keyword to emit the variables to RAM for easy inspection. You will see compiler warnings from C library functions.

Note that the floating point programs loads (huge) functions from the C library and possibly creates a binary that is too large to fit in the ROM. In that case, the linker will issue an error and does not build the binary. You have to update the data sizes in the VHDL description and update the linker script with suitable data sizes.

10.1 Monitor program

A simple monitor program is available, see the `monitor` directory in software examples. It makes use of the USART. You need a terminal program like Putty and a USART to USB device for your computer. Currently the following commands are available:

- `l` – set Little Endian (default),
- `b` – set Big endian.
- `rw <address>` – read word at address (4-byte boundary),
- `rh <address>` – read half word at address (2-byte boundary),
- `rb <address>` – read byte at address,
- `ww <address> <data>` – write word at address (4-byte boundary),
- `wh <address> <data>` – write half word at address (2-byte boundary),
- `wb <address> <data>` – write byte at address,

`<address>` and `<data>` in hexadecimal and Big Endian format. For ROM and RAM read accesses, best set to Little Endian. For I/O accesses, set Big Endian. Make sure to read and write I/O as words. Note that ROM cannot be written.

11 Address ranges and memory sizes

By default, the ROM starts at address 0x00000000 and and has a size of 64 kB (16 k words). The Program Counter starts at address 0x00000000. The RAM starts at address 0x20000000 and has a size of 32 kB (8 k words). The stack pointer is set to one address above the last RAM byte, by default at 0x20008000. The I/O starts at address 0xF0000000 and has a size of 16 kB (4 k words).

The ROM, RAM and I/O 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 ROM
-- NOTE: the ROM is word (32 bits) size.
-- NOTE: data is in Little Endian format (as by the toolchain)
--       for half word and word entities
--       Set rom_size_bits as if it were bytes
-- NOTE: rom_size_bits must be <= 16
constant rom_size_bits : integer := 16;
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 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 := 15;
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 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 := 6;
constant io_size : integer := 2**(io_size_bits-2);
type io_type is array (0 to io_size-1) of data_type;
```

In this setting, the ROM is 64 kB long and the RAM is 32 kB long. Please note that both ROM and RAM bits may not exceed 3,153,920 bits of onboard RAM. For increased ROM and RAM size, typical values may be 128 kB ROM and 64 kB RAM.

Note that we do not use full address decoding for ROM, RAM and I/O. This means that, for example, the ROM is visible multiple times in the address space. This is called *memory foldback*. For the ROM this is at 64 kB intervals. So the contents of address 0x00000000 is also available at address 0x00010000.

12 Future plans

Some future plans:

- Implement exceptions and interrupts in general. This will bring the possibility to add more I/O, such as timers.
- Implement the M standard: multiplier and divider. Multiplication can be achieved with the onboard multipliers (most FPGAs have those onboard and the synthesizer will use them) 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.
- Test more functions of the standard and mathematical libraries. Now only a few functions are tested.

13 Author's note

I managed to create the 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.

A Port I/O

The processor is equipped with a single 32-bit input and 32-bit output port. There is no data direction register. The reason for this is that the tri-state buffers for bi-direction must be in the top-level entity of the hardware design, making it impossible to use the processor in a greater design. Note that all accesses on the I/O are in Big Endian. This means that bit 31 of the input will be placed in bit 31 of the used variable.

The I/O registers are directly addressable of by a struct. See the listing below.

```
#ifndef _IO_H
#define _IO_H

#include <stdint.h>

#define IO_BASE (0xf0000000UL)

#define GPIOA_PIN  (*(volatile uint32_t*)(IO_BASE+0x00000000UL))
#define GPIOA_POUT (*(volatile uint32_t*)(IO_BASE+0x00000004UL))

typedef struct {
        volatile uint32_t PIN;
        volatile uint32_t POUT;
} GPIO_struct_t;

#define GPIOA_BASE (IO_BASE+0x00000000UL)

#define GPIOA ((GPIO_struct_t *) GPIOA_BASE)

#define USART_DATA (*(volatile uint32_t*)(IO_BASE+0x00000020UL))
#define USART_BAUD (*(volatile uint32_t*)(IO_BASE+0x00000024UL))
#define USART_CTRL (*(volatile uint32_t*)(IO_BASE+0x00000028UL))
#define USART_STAT (*(volatile uint32_t*)(IO_BASE+0x0000002CUL))

typedef struct {
        volatile uint32_t DATA;
        volatile uint32_t BAUD;
        volatile uint32_t CTRL;
        volatile uint32_t STAT;
} USART_struct_t;

#define USART_BASE (IO_BASE+0x00000020UL)

#define USART ((USART_struct_t *) USART_BASE)

#endif
```

To read the inputs, use:

```
uint32_t input;

input = GPIOA->PIN;
```

To set the outputs, use:

```
uint32_t output = 0xff00ff00;

GPIOA->POUT = output;
```

B UART Code

The UART can currently transmit and receive data with one stop bit, 8 data bits and 1 stop bit. No parity is supported. This is called 8N1. Transmission is tested with a baud rate of 9600 bps and 115200 bps. There are no auxiliary control signals (e.g. RTS and CTS). There is no embedded FIFO to buffer incoming data. The UART is programmable using I/O registers, see Appendix C.

To initialize the UART, use the code in the listing below:

```
/* Frequency of the DE0-CV board */
#define F_CPU (50000000UL)
/* Transmission speed */
#define BAUD_RATE (9600UL)

/* Initialize the Baud Rate Generator */
void usart_init(void)
{
    /* Set baud rate generator */
    USART->BAUD = F_CPU/BAUD_RATE-1;
}
```

To send a single character with waiting, use the code in the listing below:

```
/* Send one character over the USART */
void usart_putc(int ch)
{
    /* Transmit data */
    USART->DATA = (uint8_t) ch;

    /* Wait for transmission end */
    while ((USART->STAT & 0x10) == 0);
}
```

To send a null-terminated string, use the code in the listing below:

```
/* Send a null-terminated string over the USART */
void usart_puts(char *s)
{
    if (s == NULL)
    {
        return;
    }

    while (*s != '\0')
    {
        usart_putc(*s++);
    }
}
```

To wait for a character reception, use the code in the listing below:

```
/* Get one character from the USART in
 * blocking mode */
int usart_getc(void)
{
        /* Wait for received character */
        while ((USART->STAT & 0x04) == 0);

        /* Return 8-bit data */
        return USART->DATA & 0x000000ff;
}
```

To receive a string from the UART, including some line editing, use the code in the listing below:

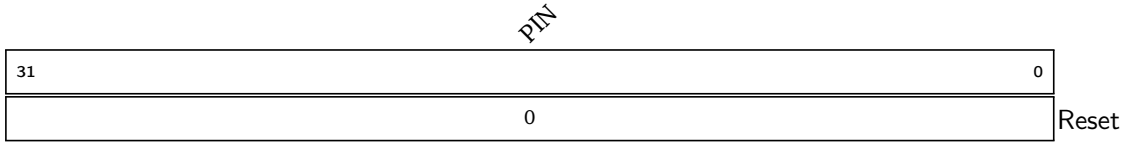
```
/* Gets a string terminated by a newline character from usart
 * The newline character is not part of the returned string.
 * The string is null-terminated.
 * A maximum of size-1 characters are read.
 * Some simple line handling is implemented */
int usart_gets(char buffer[], int size) {
    int index = 0;
    char chr;

    while (1) {
        chr = usart_getc();
        switch (chr) {
            case '\n':
            case '\r':  buffer[index] = '\0';
                        usart_puts("\r\n");
                        return index;
                        break;
```

```
            /* Backspace key */
            case 0x7f:
            case '\b':  if (index>0) {
                            usart_putc(0x7f);
                            index--;
                        } else {
                            usart_putc('\a');
                        }
                    break;
            /* control-U */
            case 21:    while (index>0) {
                            usart_putc(0x7f);
                            index--;
                        }
                        break;
            /* control-C */
            case 0x03:   usart_puts("<break>\r\n");
                        index=0;
                        break;
            default:    if (index<size-1) {
                            if (chr>0x1f && chr<0x7f) {
                                buffer[index] = chr;
                                index++;
                                usart_putc(chr);
                            }
                        } else {
                            usart_putc('\a');
                        }
                        break;
        }
    }
    return index;
}
```

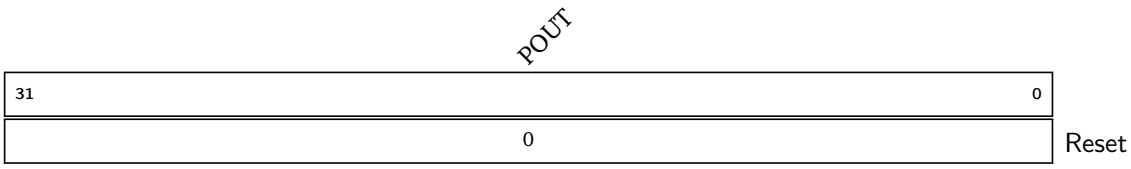
C I/O registers

This is a list of currently supported I/O addresses. The default start address is 0xF0000000. The offset is given in bytes. Note that the I/O can only be accesses on 4-byte boundaries and on word size accesses.



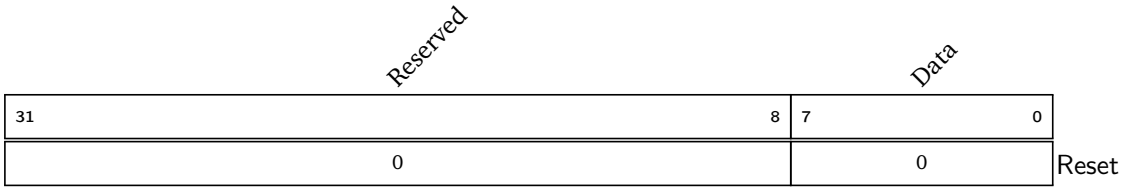
Register C.1: PORT INPUT REGISTER PIN *(0x00)*

Note: This I/O register can only be read.



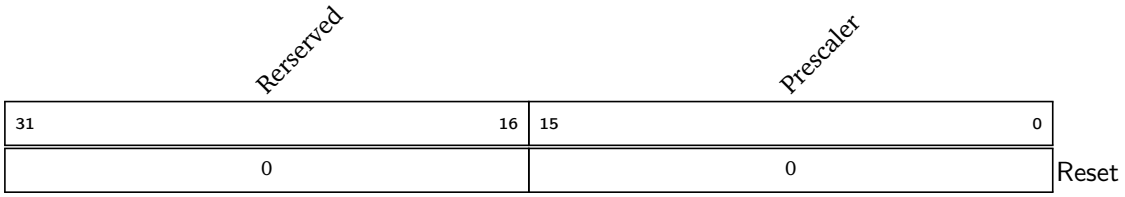
Register C.2: PORT OUTPUT REGISTER POUT *(0x04)*

Write The data is written to the output pins.
Read The last entered data is read back.



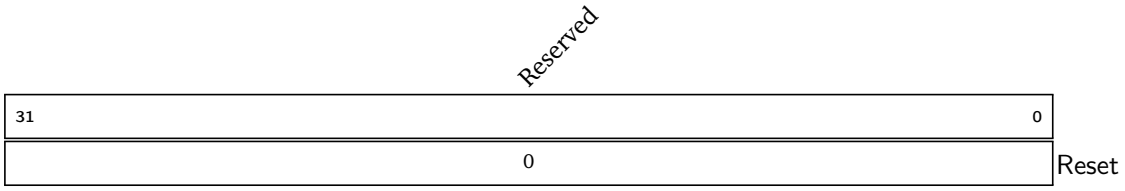
Register C.3: USART DATA REGISTER USART_DATA *(0x20)*

Write The data is written to an internal buffer and transmitted.
Read The last received data is read.



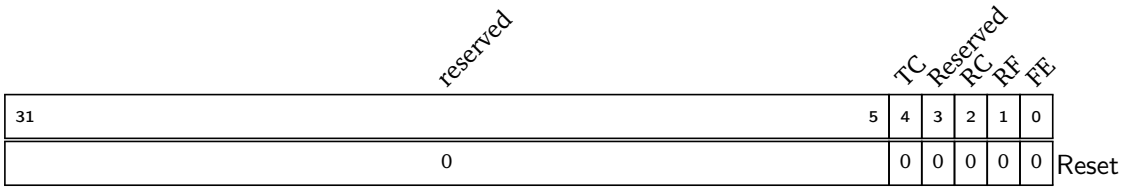
Register C.4: USART BAUD RATE REGISTER USART_BAUD *(0x24)*

Prescaler $\text{Baud rate} = \dfrac{f_{system}}{\text{prescaler} + 1}$



Register C.5: USART CONTROL REGISTER USART_CTRL *(0x28)*

Note: currently no bits are assigned to the Control Register.



Register C.6: USART STATUS REGISTER USART_STAT *(0x2c)*

TC Transmit completed. Set directly to 1 when a character was transmitted. Automatically cleared when writing new character to the data register or when writing 0 in the TC bit in USART_STAT.

RC Receive completed. Set to 1 when a character was received. Automatically cleared when data register is read or when writing 0 in the RC bit in USART_STAT.

RF Receive failed. Set to 1 when failed receiving (invalid start bit). Automatically cleared when data register is read or when writing 0 in the RF bit in USART_STAT.

FE Frame error. Set to 1 when a low is detected at the position of a stop bit. Automatically cleared when data register is read or writing a 0 in the FE bit in USART_STAT.