# A Minimal RISC-V processor in VHDL

Jesse E.J. op den Brouw*

The Hague University of Applied Sciences

April 5, 2022

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 execute the RV32IM unprivileged instruction set, and most of the instructions from the privileged instruction set. The processor incorporates a ROM, RAM, some simple I/O, a CSR and LIC (local interrupt controller). 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. Many C programs were successfully tested using the GNU C compiler. 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 has a simple two-stage pipeline and executes each instruction in two clock cycles, but the next instruction is fetched while the current instruction is executed. Jump/branches taken require two clock cycles. ROM and RAM reads need an extra clock cycle. This processor also has a basic Control and Status Registers (CSR) set, suitable to handle traps, and a hardware multiplier/divider.

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 core contains one HART (Hardware Thread). The processor is able to run a compiled C-program. C++ is currently not supported. The processor can handle the RV32IM Base Integer Instruction Set as set forward in "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA". The processor can handle traps (interrupts/exceptions). The aim is to synthesize for a minimum clock frequency of 50 MHz. The processor utilizes ROM, RAM and some simple I/O (including TIME and TIMECMP) effectively making it a microcontroller.

The processor 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 require 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 are implemented using onboard RAM block (which are buffered with an output register). The 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, shadowed from memory mapped registers), INSTRET and INSTRETH (number of retired instructions), and a basic set of CSRs to handle traps. A hardware multiplication requires three clock cycles to complete. A hardware division requires 16+2 clock cycles to complete.

The processor executes at top 1 MIPS/MHz (jump and branches taken and reading ROM/RAM for data requires an extra clock cycle, multiplications/divisions take more clock cycles). The processor only supports Machine mode (M mode).

# 2 The processor

The 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.

- The Control and Status Registers contains a basic set of register for trap handling in Machine mode.
- The Local Interrupt Controller determines which trap request is fed to the processor/CSR.
- The multiply/divide unit adheres the M-standard.

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

## 2.1 Registers

The register file 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.

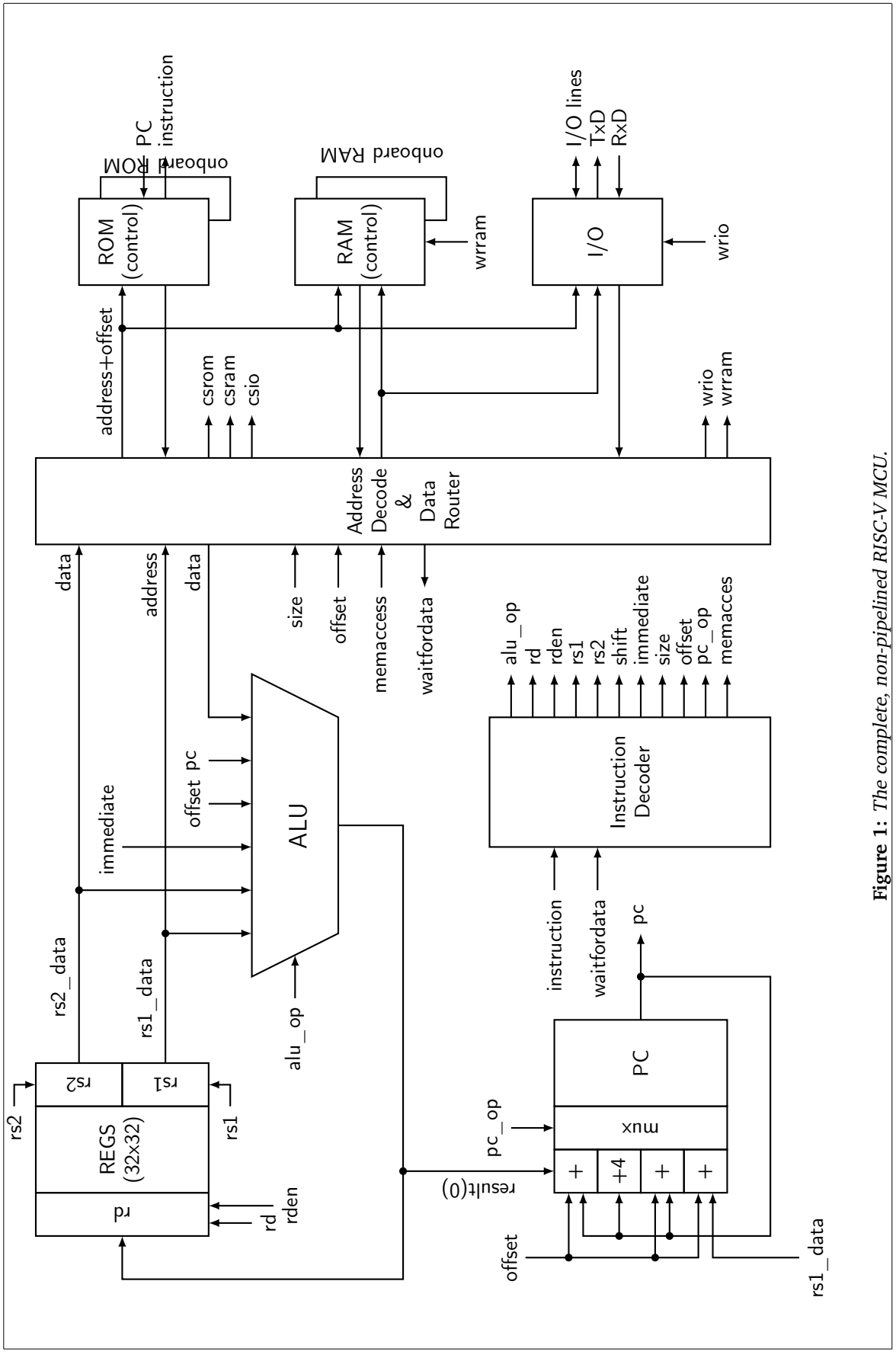



**Figure 1:** *The complete, non-pipelined RISC-V MCU.*

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.

## 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. The RAM cannot be used for program data.

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, a simple USART (well, really an UART), a simple timer and the TIME and TIMEH memory mapped time registers. 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. Note that not all I/O addresses are used.

The USART can transmit and receive data at 7, 8 or 9 bits, no/even/odd parity and 1 or 2 stop bits. Tested speeds are 9600 bps, 115200 bps and 230400 bps. Several status flags are implemented to guide transmission. Note: more identical USARTs will be added. Receive and transmitted character (local) interrupts are provided.

The timer has a 32-bit time register and increments on every clock cycle. It does not have a prescaler. It counts up to a compare match register, after which is will be loaded with 0 again. A compare match (local) interrupt is provided.

The I/O incorporates memory mapped TIMEH:TIME and TIMECMPH:TIMECMP registers. Whenever TIMEH:TIME (as viewed as a 64-bit register)is greater than or equal to TIMECMPH:TIMECMP (as viewed as a 64-bit register) an interrupt is asserted. The interrupt is negated if TIMEH:TIME is less than TIMECMPH:TIMECMP.

## 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. This is handled by the C compiler. 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. The ALU does not handle multiplications and divisions.

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 fetched instruction. The address is always on a 4-byte boundary although function calls and conditional jump (JAL, JALR en B$xx$ 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, RAM, ROM, I/O, the PC, the Address Decoder, the CSR, the register file and the md unit. Some signals are directly wired to the ROM, RAM and I/O.

The processor uses a five-state FSM, see figure 2. The PC points to +4 of the currently executing instruction, in a sequential instruction stream. 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 (Figure 3). Reading RAM and ROM requires an extra state (Figure 4). When a multiply, divide or remainder instruction is encountered, the FSM enters the `md` state and waits for the md-unit to complete (32+2 or 16+2 clock cycles). Note that penalty, wait and start cannot occur at the same time.

### 2.7.1 Trap handling in the instruction decoder

When an interrupt occurs, the FSM enters the `intr` state for fetching the trap vector. An interrupt can occur any time and is called asynchronous. When an interrupt occurs in the `fetch` state, the current instruction is discarded. When an interrupt occurs in the `fexec` state, the current instruction is either retired or discarded. In the latter case the instruction must be restarted. When occurring in the `wait` state, the instruction (load)

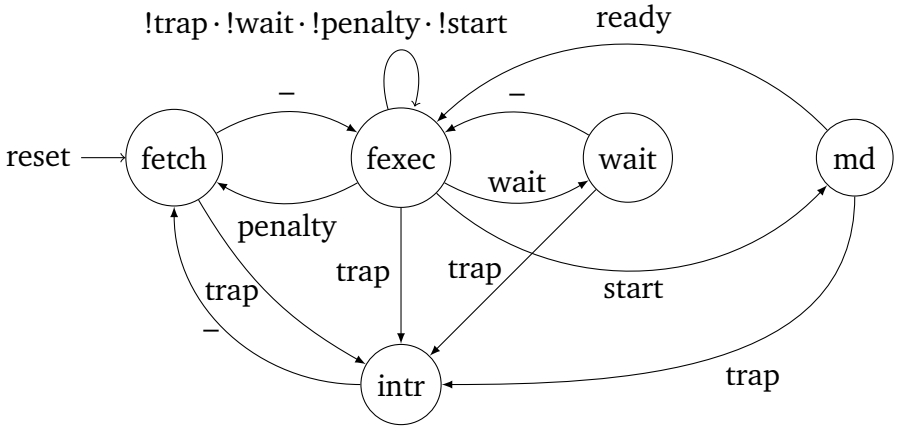



**Figure 2:** *FSM of the instruction decoder of the processor.*

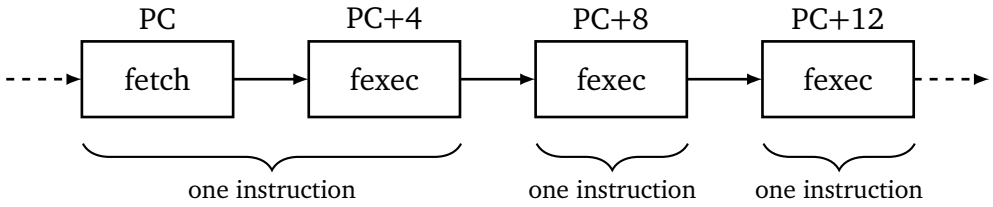



**Figure 3:** *State sequence for start up/penalty with instruction execution (no wait state).*

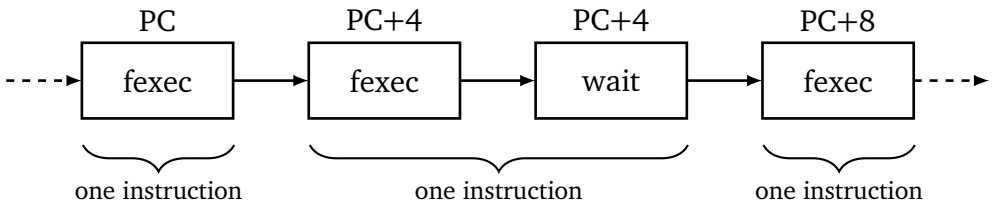



**Figure 4:** *State sequence for instruction execution (wait state).*

is retired. When occurring in the `md` state *and* the md unit is ready, the instruction (mul, div, rem) is retired. When occurring in the `md` state and the md unit is *not* ready the instruction is discarded and must be restarted. After loading the trap vector, the FSM enters the `fetch` state.

Exceptions are synchronous to executing instructions. When occurring in the `fetch` state, the instruction is discarded and the FSM enters the `intr` state. When occurring in the `fexec` state, the instruction is either retired or discarded (load, mul, div, rem). In this case the instruction must be restarted. Note that exceptions cannot occur in the `wait` and `md` states.

When a trap request is asserted, the trap vector is loaded in the next clock cycle. Then, in the second clock cycle, the instruction is fetched and in the third clock cycle the instruction is executed. On return from a trap handler, the original contents of the PC is restored and the FSM enters the `fetch` state.

## 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.

Note that instructions can only be fetched from ROM.

## 2.9 Control and Status registers

The RISC-V specification describes a set of 4096 control and status register in a separate address space. Basic event counters are 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 registers shadow the contents of the TIME and TIMEH registers from the I/O.
- `instret` and `instreh` – 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.

For trap handling, the following registers are implemented:

- `mvendorid` – this register is hardwired to all zero bits.
- `marchid` – this register is hardwired to all zero bits.
- `mimpid` – this register is hardwired to all zero bits.
- `mhartid` – this register is hardwired to all zero bits.
- `mconfigptr` – this register is hardwired to all zero bits.
- `mstatus` – the only implemented bits are `MIE`, `MPIE` and `MPP`, all other bits are hardwired zero.
- `misa` – hardwired to value 0x40001100, indicating 32-bit processing, RV32I base ISA and Integer Multiply/Divide extension.
- `mie` – for the lower 16 bits, only `MTIE` is implemented, all other bits are hardwired zero.
- `mtvec` – contains the trap handler (vector) address, can be used in direct and vectored mode.

- `mstatush` – this register is hardwired to all zero bits.
- `mscratch` – currently not used, but can be used by software trap handlers.
- `mepc` – contains the PC at point of trap of the *currently* executing instruction.
- `mcause` – contains the cause of the trap as set forward in "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture". For local interrupts, additional codes are used.
- `mtval` – contains the address on the address bus when a trap occurs, not always relevant.
- `mip` – contains the pending interrupts. For the lower 16 bits, only `MTIP` is implement. The upper 16 bits are used for local interrupts. This register is read-only.
- `mconfigptr` – this register is hardwired to all zero bits.

The `mcounteren` and `mcountinhibit` registers are not implemented, because U-mode is not implemented. The `menvcfg` and `menvcfgh` are not implemented, because only M-mode is implemented.

The trap handler (vector) address must be loaded by software at boot time. Both direct and vectored mode are supported. In direct mode all traps redirect to a trap handler that has to handle both interrupts and exceptions. The most significant bit of `mcause` is 1 when a trap occurred from an interrupt. In vectored mode, the addresses of *interrupt handlers* are loaded from a jump table. Exceptions are redirected to a single handler. Note that the address of the trap handler must be on a 4-byte boundary, and bit 0 has to be set to 1 for vectored mode.

Other CSRs are not implemented. Good synthesizers will remove not-used CSRs.

## 2.10 Local Interrupt Control unit

The Local Interrupt Controller is responsible for selecting which trap request must be serviced by the core. Interrupts have higher priority than exceptions. The state of the serviced trap is visible in the CSR.

The LIC can handle 16 local interrupts (numbered 16 to 31) and the Machine mode external timer interrupt (numbered 7). Other standard RISC-V interrupts (numbered 0 to 6 and 8 to 15) are not available. The external timer interrupt has the highest priority, followed (currently) by the USART and TIMER1 interrupts.

Exceptions are handled as set forward in Table 3.7 of "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture": instruction access fault, instruction address misaligned, ECALL (M mode only), EBREAK, load/store address misaligned, load/store access fault. Note that ECALL and EBREAK are user instructions and can be interrupted by an interrupt (i.e. when the ECALL or EBREAK is executed).

## 2.11 Multiply/Divide Unit

The processor is equipped with a hardware multiply/divide unit. All multiply/divide instructions are supported (`MUL`, `MULH`, `MULHSU`, `MULHU`, `DIV`, `DIVU`, `REM`, `REMU`) and the result is fed to the ALU. A multiplication takes three clock cycles (one clock-in, one multiply, one clock-out). For division, two versions are available. By default, the divider needs 18 clocks (one clock-in, 16 divide, one clock-out) for a division using a poor man's radix-4 division unit. As an alternative, a simple radix-2 division unit can be selected taking 34 clock cycles (one clock-in, 32 divide, one clock-out) to do the division. Thus, the radix-4 divider unit is faster, but needs more cells. The radix-2 divider unit is slower, but needs less cells. You have to enable the M-standard support in the compiler. The multiplier uses special DSP units in the Cyclone V. Most regular FPGAs have onboard multipliers.

## 2.12 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 boot (which is 0x20008000 by default). The ISA postulates that the stack is aligned on 16-byte boundaries for the I standard.

## 2.13 Implemented instructions

For the processor, all RV32IM Unprivileged instructions are implemented with the exception of the FENCE instruction. This instruction act as a no-operation (NOP). ECALL and EBREAK are supported and execute an exception. MRET is used to return from an exception or an interrupt. WFI is not supported and act as a no-operation (NOP).

# 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 2700 ALMs (about 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. Note that the Signal Tap uses onboard memory.

# 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:

```
1 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
`RETIRED` – Descriptions that are not used anymore

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`.

If you want, you can compile the processor with the standard program incorporated, which is by default, flashing the onboard leds. 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 in `CODE` 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 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`.

If you wish to enable the multiply/divide instructions, issue:

```
../configure ../configure --prefix=/opt/riscv32 --with-arch=rv32im --with-abi=ilp32
```

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, saving 512 memory element. 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

We tested a large amount of programs with and without trap handling. We tested all 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-pipe-csr-md-lic/↩
↪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 5 to 7.

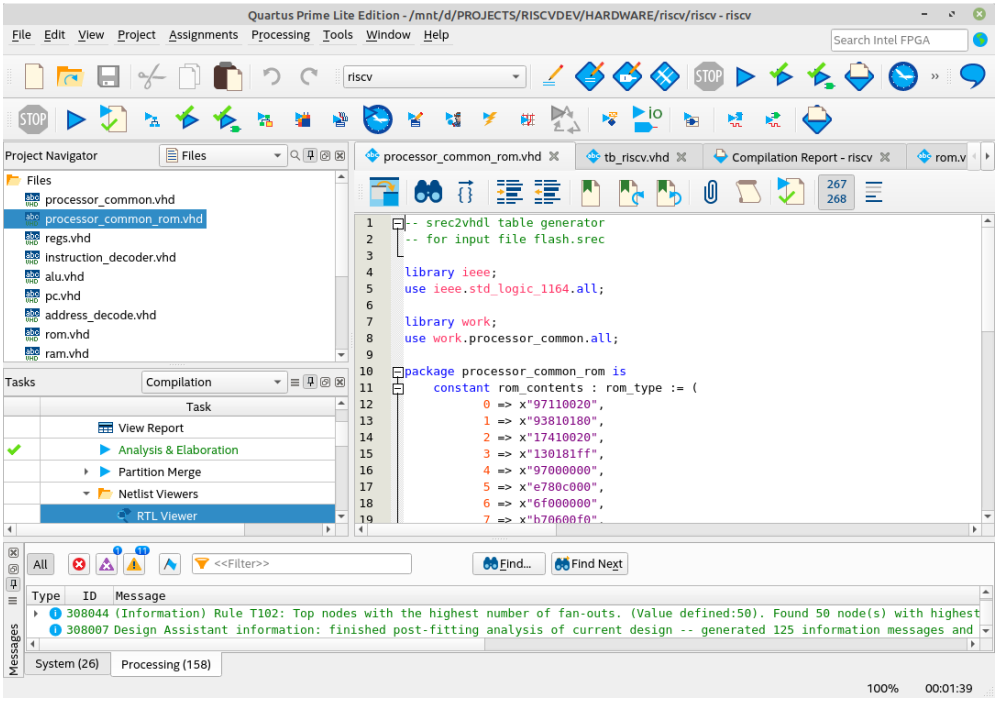



**Figure 5:** *Image of the Quartus project (1).*

# 8 Implemented system calls

The `sbrk/brk` system call, used for allocating RAM memory, is implemented. Note that there is a limited amount of RAM. Note that `sbrk` is not called by the user. Use `malloc`

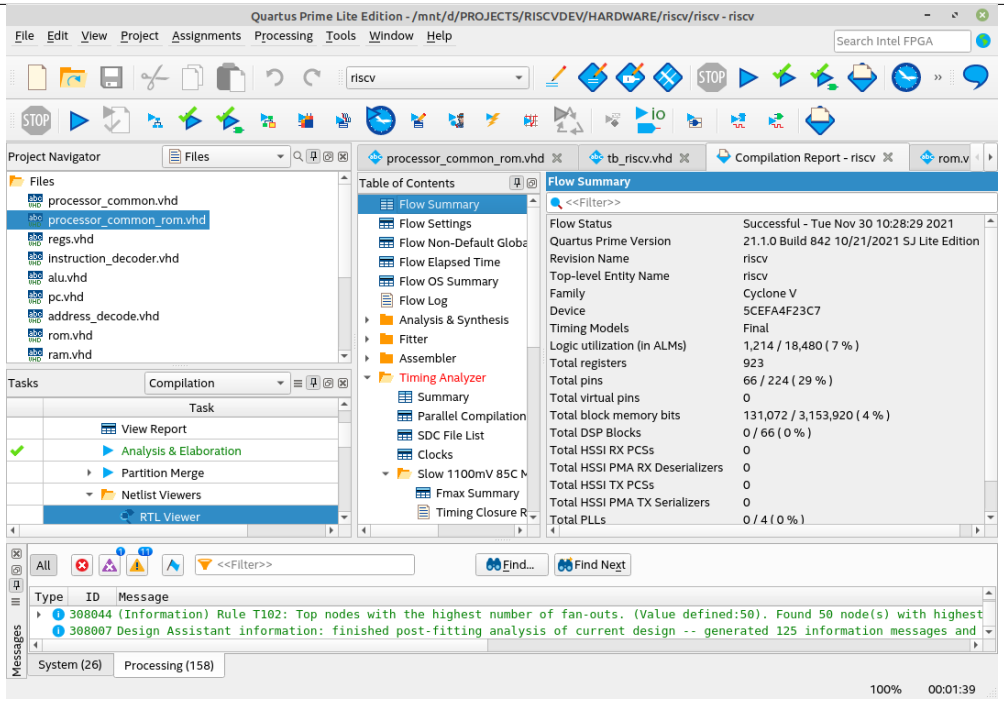



**Figure 6:** *Image of the Quartus project (2).*

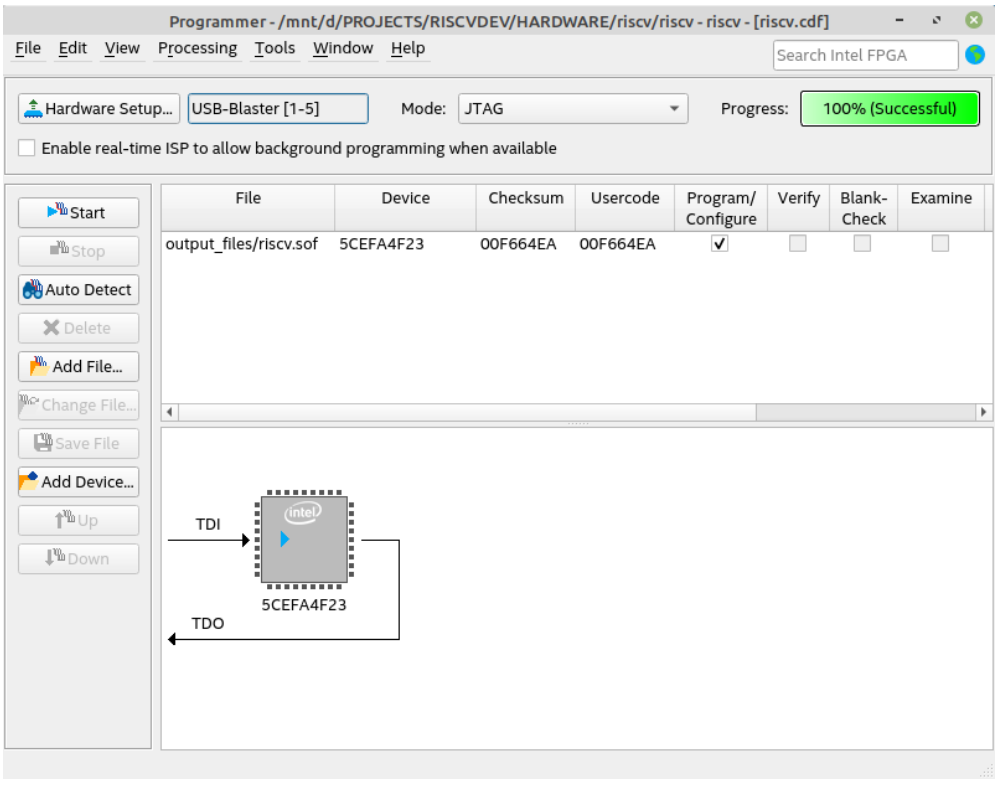



**Figure 7:** *Image of the programmer.*

et al.

The `gettimeofday` system call is implemented. It returns the seconds and microseconds

since the last reset of the processor. You need to call the `gettimeofday` C function for proper handling.

The `read` and `write` system calls are implemented but in turn they call the userland functions `__io_getchar` and `__io_putchar` functions to read or write a character. Normal use is for the latter two to transmit or receive via the USART. When implemented, `printf` and `scanf` can be used.

Other system calls return an error because they cannot fulfill the requested operation, such as `open`. Note that some system calls are in fact not implemented and return undetermined behavior.

Note: when using traps, the system calls are handled by a trap handler (by using ECALL). This is the default behavior of the tool chain. When not using traps, the system calls are rerouted to functions in a library. You need to set up your software properly. See the software examples.

# 9 Using trap handlers in software

We provide (see software examples `interrupt_direct` and `interrupt_vectored`) a basic implementation of trap handlers. The `universal_handler` handles all traps (interrupts and exceptions) in direct mode. The entry point (the address loaded in the `mtvec` CSR) must be set in the `main` function, as is enabling traps. In vectored mode, interrupts are redirected to their own handlers via a jump table. The start address of the jump table must be set in the `main` function, and traps must be enabled. The external timer has its own handler called `external_timer_handler`. TIMER1 has its own handler called `timer1_cmpt_handler`. The USART has its own handler called `usart_handler`. This handler is used for both receive and transmit interrupts. Note that negating the interrupt request must be done by software in the respective handlers. The interrupt requests is *not* negated by hardware.

# 10 VHDL files

The VHDL description is 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 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, uninitialized RAM blocks.
- `io.vhd` – Description of the I/O. It contains a 32-bit input register, a 32-bit output register, an USART, a simple timer and the TIME and TIMECMP memory mapped registers
- `csr.vhd` – Description of the Control and Status registers.
- `md.vhd` – Description of the multiply/divide unit.
- `riscv.vhd` – Top-level description of the processor. Connects all 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.

There are two architectures for the ALU: a non-optimized, easy-to-follow architecture and an optimized architecture. First impression is that the optimized architecture saves about 500 ALMs in respect to the non-optimized architecture.

There are two architectures for the multiply/divide unit: one with a radix-2 divider and one with a radix-4 divider. The radix-4 divider uses more cells than the radix-2 divider but is faster.

# 11 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 [-fbwhqv0] [-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.
- `-0` Output unused ROM data as 0 instead of don't care.
- `-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, except when the `-0` option is used. Don't cares are set to 0 by Quartus on default.

# 12 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. For use in the simulator.
- `assembler` – a simple assembler program. For use in the simulator.
- `basel_problem` – a program that calculates the sum of the inverses of the squares of natural numbers, up to 1000. For use in the simulator. Used to test de divider.
- `clock` – a simple clock using the CSR TIME and TIMEH registers to fetch the time since last reset. Works on the board.
- `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).
- `hex_display` – program that reads 8 switches from the board and display them as a 2-digit hexadecimal value on the 7-segment display. Works on the board.
- `interrupt_direct` – program to test the interrupt handling using direct mode. Works on the board.
- `interrupt_vectored` – program to test the interrupt handling using vectored mode. Works on the board.
- `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` (and therefore `malloc` et al.).
- `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 $2\pi$. For use in the simulator. The result must be $\pi$.
- `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_printf` – simple program that prints an integer, a pointer, a float and a double to the terminal using `printf`, this is a big binary (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.

When using floats and doubles in `sprintf`/`printf`, you need to supply the linker with the `-u _printf_float` option. When using floats and doubles in `sscanf`/`scanf`, you need to supply the linker with the `-u _scanf_float` option. Also, using `printf` and `scanf` creates big binaries.

## 12.1 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.

# 13 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 data_type;
-- 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 := 8;
constant io_size : integer := 2**(io_size_bits-2);
type io_type is array (0 to io_size-1) of data_type;
```

Note that you also have to make changes to the linker script. In the file `riscv.ld`, at the top you will find the following lines. Change the origins in accordance with the VHDL description.

```
ENTRY( _start )

MEMORY
{
   ROM (rx)    : ORIGIN = 0x00000000, LENGTH = 64K
   RAM (rw)    : ORIGIN = 0x20000000, LENGTH = 32K
   IO (rw)     : ORIGIN = 0xf0000000, LENGTH = 16K
}
```

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.

# 14 Future plans (or not) and issues

Some future plans:

- We are *not* planning the C standard.
- Plans to create a hardcoded bootloader to load programs when the processor is loaded in an FPGA. We need the modify the instruction fetch hardware. Instructions can currently only be fetched from ROM.
- We strive to implement SPI and I2C, and PWM.
- We are considering implementing a 5-stage instruction pipeline (this will take time ;-)). This will speed up the clock speed.
- Implement Supervisor Mode (this will also take some time ;-)).
- Smaller (in cells) divide unit.
- Test more functions of the standard and mathematical libraries. Now only a few functions are tested.
- It is not possible to print `long long` (i.e. 64-bit) using `printf` et al. When using the format specifier `%lld`, `printf` just prints `ld`.
- Further optimize the ALU for size and speed.

# 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 or by a struct. See the listing below.

```
1  #ifndef _IO_H
2  #define _IO_H
3
4  #include <stdint.h>
5
6
7  /* Base address of the I/O */
8  #define IO_BASE (0xf0000000UL)
9
10 /* General purpose I/O */
11 #define GPIOA_PIN  (*(volatile uint32_t*)(IO_BASE+0x00000000UL))
12 #define GPIOA_POUT (*(volatile uint32_t*)(IO_BASE+0x00000004UL))
13
14 typedef struct {
```

```
15          volatile uint32_t PIN;
16          volatile uint32_t POUT;
17  } GPIO_struct_t;
18
19  #define GPIOA_BASE (IO_BASE+0x00000000UL)
20
21  #define GPIOA ((GPIO_struct_t *) GPIOA_BASE)
22
23
24  /* USART (USART1) */
25  #define USART_DATA (*(volatile uint32_t*)(IO_BASE+0x00000020UL))
26  #define USART_BAUD (*(volatile uint32_t*)(IO_BASE+0x00000024UL))
27  #define USART_CTRL (*(volatile uint32_t*)(IO_BASE+0x00000028UL))
28  #define USART_STAT (*(volatile uint32_t*)(IO_BASE+0x0000002CUL))
29
30  typedef struct {
31          volatile uint32_t DATA;
32          volatile uint32_t BAUD;
33          volatile uint32_t CTRL;
34          volatile uint32_t STAT;
35  } USART_struct_t;
36
37  #define USART_BASE (IO_BASE+0x00000020UL)
38
39  #define USART ((USART_struct_t *) USART_BASE)
40
41
42  /* TIMER (TIMER1) */
43  typedef struct {
44          volatile uint32_t CTRL;
45          volatile uint32_t STAT;
46          volatile uint32_t CNTR;
47          volatile uint32_t CMPT;
48  } TIMER_struct_t;
49
50  #define TIMER1_BASE (IO_BASE+0x00000080UL)
51  #define TIMER1 ((TIMER_struct_t *) TIMER1_BASE)
52
53
54  /* RISC-V system timer (in I/O) */
55  #define TIME (*(volatile uint32_t*)(IO_BASE+0x000000f0UL))
56  #define TIMEH (*(volatile uint32_t*)(IO_BASE+0x000000f4UL))
57  #define TIMECMP (*(volatile uint32_t*)(IO_BASE+0x000000f8UL))
58  #define TIMECMPH (*(volatile uint32_t*)(IO_BASE+0x000000fcUL))
59
```

```
typedef struct {
        volatile uint32_t time;
        volatile uint32_t timeh;
} TIME_struct_t;

typedef struct {
        volatile uint32_t timecmp;
        volatile uint32_t timecmph;
} TIMECMP_struct_t;

#endif
```

To read the inputs, use:

```
uint32_t input;

input = GPIOA->PIN;
```

To set the outputs, use:

```
uint32_t output = 0xff00ff00;

GPIOA->POUT = output;
```

Modifying bits of a I/O register must be done by a read-modify-write cycle. This is *not* atomically handled and can interfere with interrupts! For example, to set bit 2 of POUT, use:

```
GPIOA->POUT |= 0x04;
```

# B USART Code

The USART can send and receive data with one start bit, 7/8/9 data bits, N/E/O parity and 1 or 2 stop bits. Transmission is tested with a baud rate of 9600 bps, 115200 bps and 230400 bps. Send and receive speeds are equal as is the number of data bits, parity and the number of stop bits. There are no auxiliary control signals (e.g. RTS and CTS). There is no embedded FIFO to buffer incoming data. The USART is programmable using I/O registers, see Appendix E. Note that using a system frequency of 50 MHz, the baud rate cannot be lower than 763 bps.

To initialize the USART, 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 USART, including some simple 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;
}
```

For working with the USART on board of the processor, you need an USB-to-USART device with TTL (3.3 V) converter. An example is shown in Figure 8.

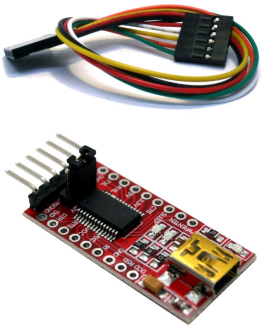


**Figure 8:** *An USB-to-USART converter. Make sure that the voltages do not exceed 3.3 V.*

# C TIMER1 code

Based on a frequency of 50 MHz, the following code will set TIMER1 interrupt on 0,5 s intervals:

```
/* Activate TIMER1 with a cycle of 2 Hz */
/* for a 50 MHz clock. Use interrupt. */
TIMER1->CMPT = 24999999;
/* Bit 0 = enable, bit 4 is interrupt enable */
TIMER1->CTRL = (1<<4)|(1<<0);
```

# D The external `time` registers

The `time` registers (TIME, TIMEH, TIMEMCP and TIMECMPH) can be accessed via the I/O and are therefore memory mapped. The following code reads in the current time and increments the compare registers with a certain *delta*. Whenever TIMEH:TIME is greater than or equal to TIMEH:TIME, an interrupt request is asserted. Negating the interrupt request is accomplished by writing a value greater than the time registers to the compare registers.

```
/* Fetch current time */
register uint64_t cur_time;
cur_time = ((uint64_t)TIMEH << 32) | (uint64_t)TIME;
/* Add delta */
cur_time += DELTA;
```

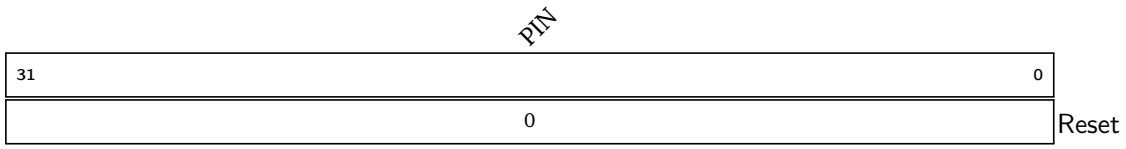
```
/* Set TIMECMP to maximum */
TIMECMPH = -1;
TIMECMP = -1;
/* Store new TIMECMP */
TIMECMP = (uint32_t)(cur_time & 0xffffffff);
TIMECMPH = (uint32_t)(cur_time>>32);
```

Normally, this code is placed in the interrupt handler. These registers are read-only shadowed by the CSR time registers.

# E I/O registers

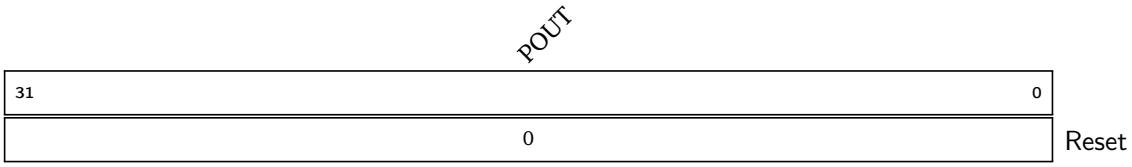
This is a list of currently supported I/O addresses. The default start address is 0xF0000000. The offset is given in bytes. Note that the I/O can only be accesses on 4-byte boundaries and on word size accesses.

## E.1 General purpose I/O




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

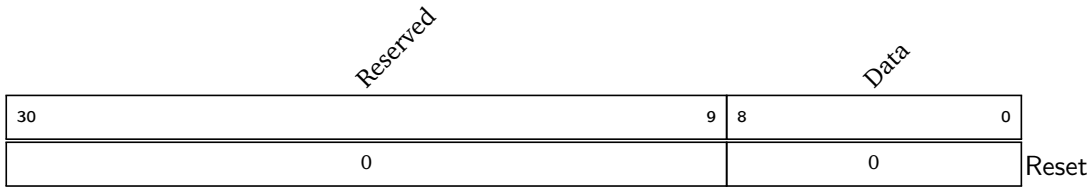
**Note:** This I/O register can only be read.




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

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

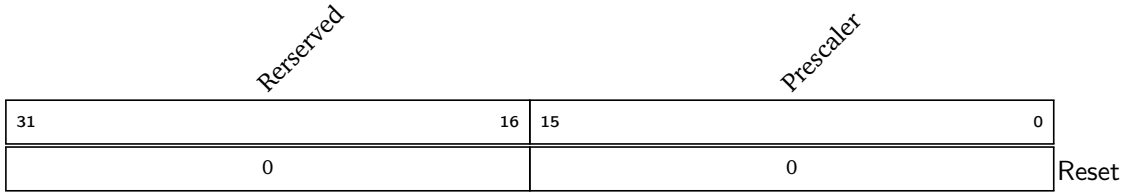
## E.2 USART




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

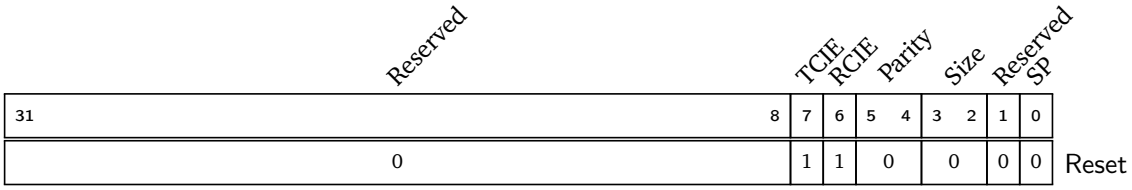
| | |
|---|---|
| **Write** | The data is written to an internal buffer and transmitted. |
| **Read** | The last received data is read. |

Size depends on the Size field in the USART Control Register.



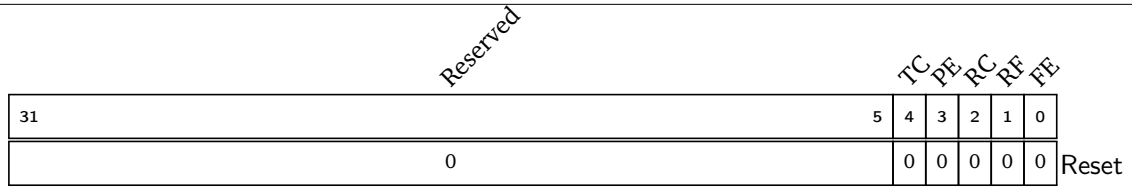

**Register E.4:** USART BAUD RATE REGISTER USART_BAUD *(0x24)*

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




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

| | |
|---|---|
| **TCIE** | Transmit character interrupt enable. |
| **RCIE** | Receive character interrupt enable. |
| **Parity** | 00: none, 10: even, 11: odd. |
| **Size** | 00: 8 bits, 10: 9 bits, 11: 7 bits, excluding the parity. |
| **SP** | 0: one stop bit, 1: two stop bits. |




**Register E.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.

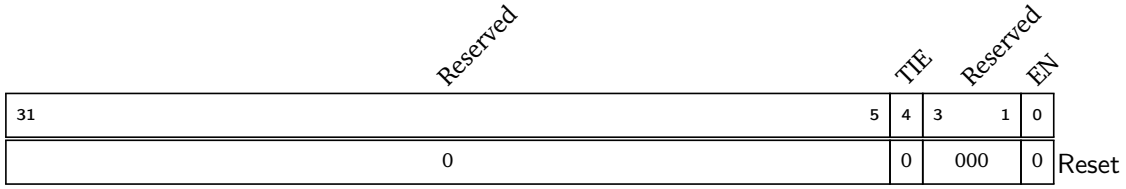
**PE** Parity error. Set to 1 if parity is enabled and there is a parity error while receiving. Automatically cleared when data register is read or when writing 0 in the PE bit in USART_STAT.

**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 the (first) stop bit. Automatically cleared when data register is read or writing a 0 in the FE bit in USART_STAT.

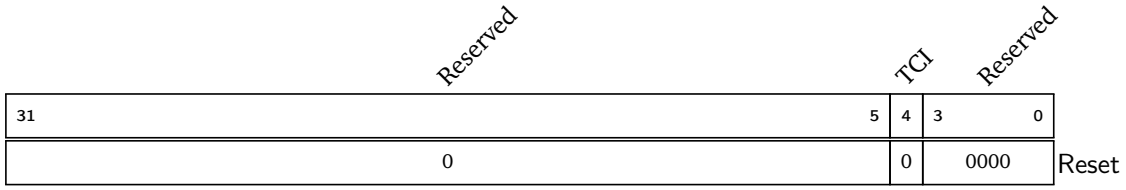
## E.3 TIMER1




**Register E.7:** TIMER1 CONTROL REGISTER TIMER1_CTRL *(0x80)*

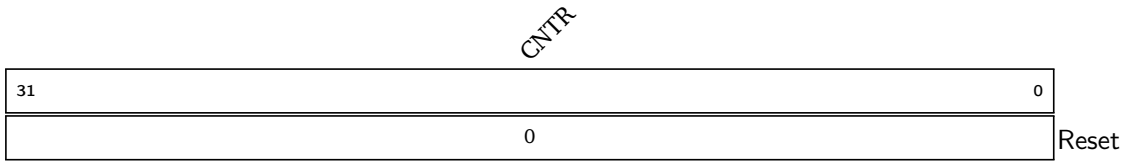
**EN** Enable the timer

**TIE** Timer compare match interrupt enable



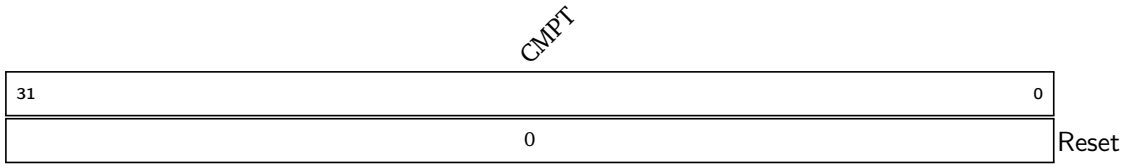

**Register E.8:** TIMER1 STATUS REGISTER TIMER1_STAT *(0x84)*

**TCI** Timer compare match interrupt. Set to 1 on compare match between the timer Count register and the Compare Match register. Must be cleared by software by writing a 0.



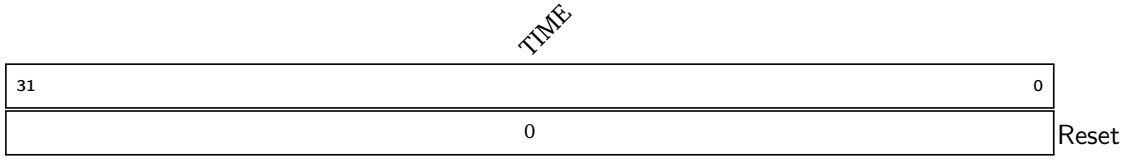

**Register E.9:** TIMER1 COUNT REGISTER TIMER1_CNTR *(0x88)*

**CNTR** This register holds the counted clock pulses on the timer. This register may be written by software. Rolls over every $2^{32}/f_{clk}$ seconds.



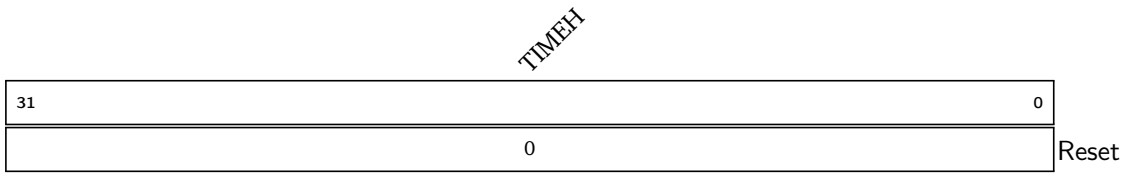

**Register E.10:** TIMER1 COMPARE TIMER REGISTER TIMER1_CMPT *(0x8c)*

**CMPT** This register holds the value at which the counter register is compared. On CNTR compares to greater than or equal to CMPT, the counter register will be cleared and the TCI flag will be set (both in the next clock cycle).



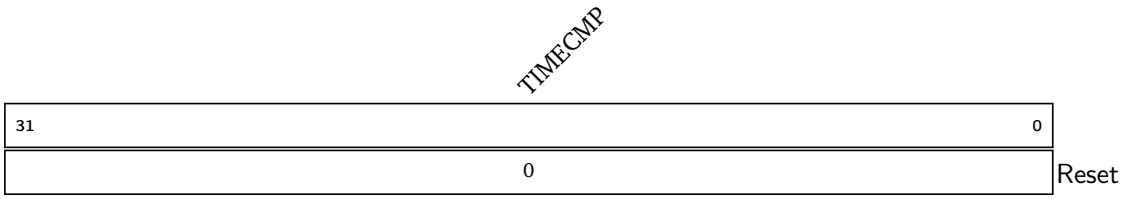

**Register E.11:** TIME EXTERNAL TIMER REGISTER TIME *(0xf0)*

This register holds the low 32 bits of the external timer. Currently read-only.



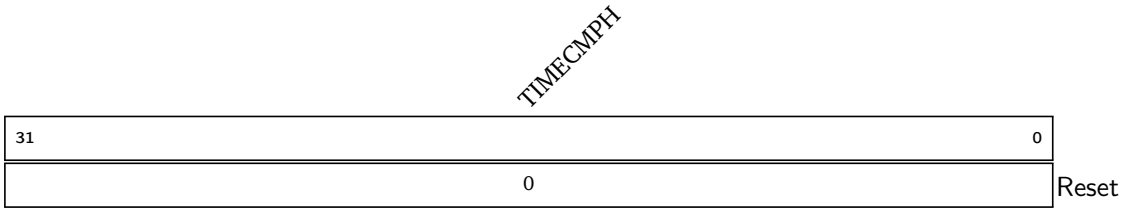

**Register E.12:** TIMEH EXTERNAL TIMER REGISTER TIME *(0xf4)*

This register holds the upper 32 bits of the external timer. Currently read-only.




**Register E.13:** TIMECMP EXTERNAL TIMER COMPARE REGISTER TIMECMP *(0xf8)*

This register holds the low 32 bits of the external timer compare register.




**Register E.14:** TIMECMPH EXTERNAL TIMER COMPARE REGISTER TIMECMP *(0xfc)*

This register holds the upper 32 bits of the external timer compare register.