Documentation

Version 1.0

Table of Contents

1 Overview

1.1 About Steel Core

Steel is a microprocessor core that implements the RV32I and Zicsr instruction sets of the RISC-V specifications. It is designed to be easy to use and targeted for embedded systems projects.

1.2 Licensing

Steel is distributed under the MIT License. The license text is reproduced below. Read it carefully and make sure you understand its terms before using Steel in your projects.

1.3 Specifications

Steel implements the base instruction set RV32I, the Zicsr extension and the M-mode privileged architecture of the RISC-V specifications.

Steel aims to be compliant with the following versions of the RISC-V specifications:

- Base ISA RV32I version 2.1
- Zicsr extension version 2.0
- Machine ISA version 1.11

1.4 Online repository

Steel files and documentation are available at GitHub (`github.com/rafaelcalcada/steel-core`).

1.5 Getting started

To start using Steel, follow these steps:

1. Import all files inside the **rtl** directory into your project;
2. Instantiate the core into a Verilog/SystemVerilog module (an instantiation template is provided below);
3. Connect Steel to a clock source, a reset signal and memory. There is an interface to fetch instructions and another to read/write data, so we recommend a dual-port memory.

There is also interfaces to request for interrupts and to update the time register. The signals of these interfaces must be hardwired to zero if unused.

```
steel_top #(

  // You must provide a 32-bit value. If omitted the boot address is set to 0x00000000
  // -----------------------------------------------------------------------------------
  .BOOT_ADDRESS()

  ) core (

  // Clock source and reset
  // -----------------------------------------------------------------------------------
  .CLK(),         // System clock (input, required, 1-bit)
  .RESET(),       // System reset (input, required, 1-bit, synchronous, active high)

  // Instruction fetch interface
  // -----------------------------------------------------------------------------------
  .I_ADDR(),      // Instruction address (output, 32-bit)
  .INSTR(),       // Instruction data (input, required, 32-bit)

  // Data read/write interface
  // -----------------------------------------------------------------------------------
  .D_ADDR(),      // Data address (output, 32-bit)
  .DATA_IN(),     // Data read from memory (input, required, 32-bit)
  .DATA_OUT(),    // Data to write into memory (output, 32-bit)
  .WR_REQ(),      // Write enable (output, 1-bit)
  .WR_MASK(),     // Write byte mask (output, 4-bit)

  // Interrupt request interface (hardwire to zero if unused)
  // -----------------------------------------------------------------------------------
  .E_IRQ(),       // External interrupt request (optional, active high, 1-bit)
  .T_IRQ(),       // Timer interrupt request (optional, active high, 1-bit)
  .S_IRQ()        // Software interrupt request (optional, active high, 1-bit)

  // Time register update interface (hardwire to zero if unused)
  // -----------------------------------------------------------------------------------
  .REAL_TIME(),   // Value read from a real-time counter (optional, 64-bit)

);
```

1.6 Configuration

Steel has only one configuration parameter, the boot address, which is the address of the first instruction the core will fetch after reset. It is defined when instantiating Steel. If you omit this parameter at instantiation, the boot address will be set to 0x00000000.

1.7 Microarchitecture

Steel has 3 pipeline stages, a single execution thread and issues one instruction per clock cycle. Therefore, all instructions are executed in program order. Its pipeline is plain simple, divided into fetch, decode, and execution stages. The reduced number of pipeline stages eliminates the need for branch predictors and other advanced microarchitectural units, like data hazard detectors and forwarding units. Fig. 1 shows the Steel microarchitecture in register-transfer level (RTL). More implementation details can be found in section 6.

Fig. 2 (next page) shows the tasks performed by each pipeline stage. In the first stage, the core generates the program counter and fetches the instruction from memory. In the second, the instruction is decoded and the control signals for all units are generated. Branches, jumps and stores are executed in advance in this stage, which also generates the immediates and fetches the data from memory for load instructions. The last stage executes all other instructions and writes back the results in the register file.

Figure 1 – Steel Core microarchitecture in detail

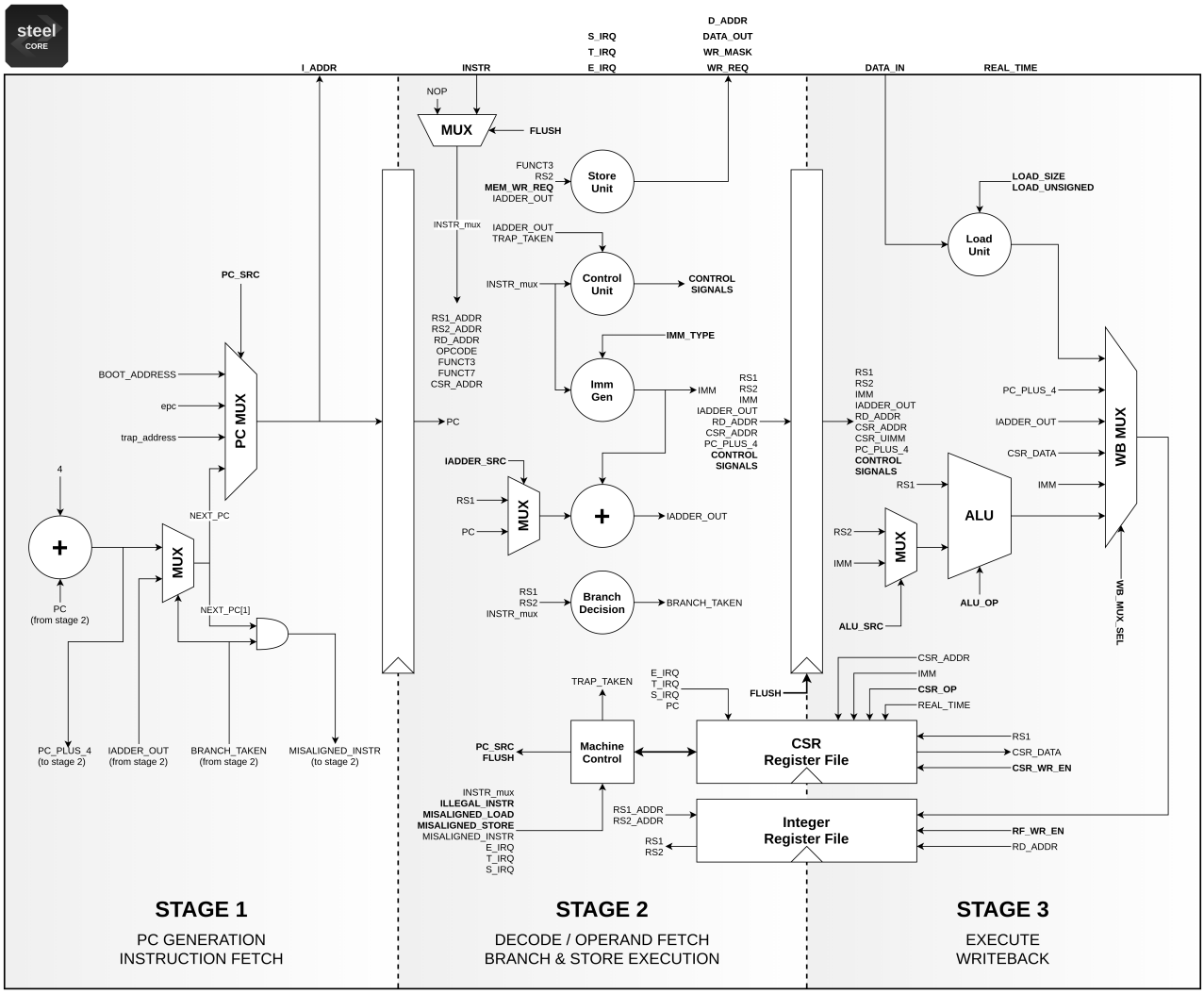
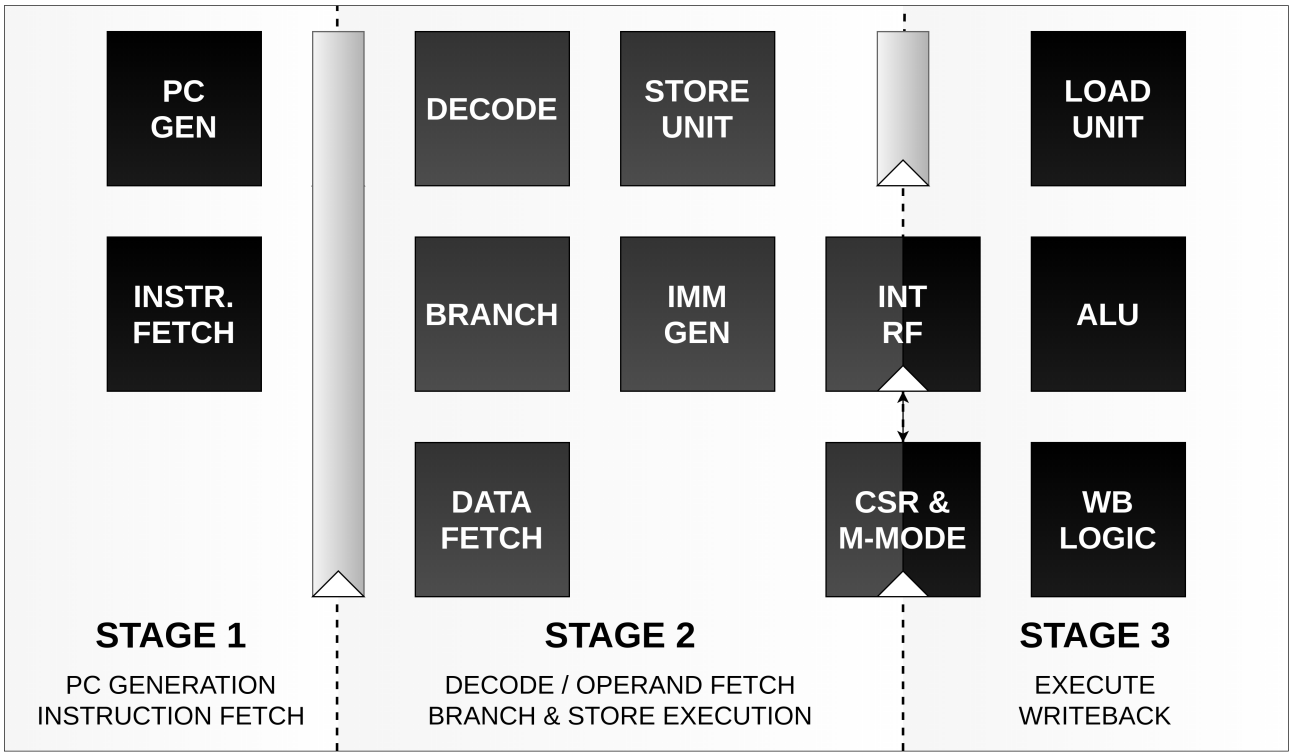


Figure 2 – Steel Core pipeline overview



2 Input and output signals

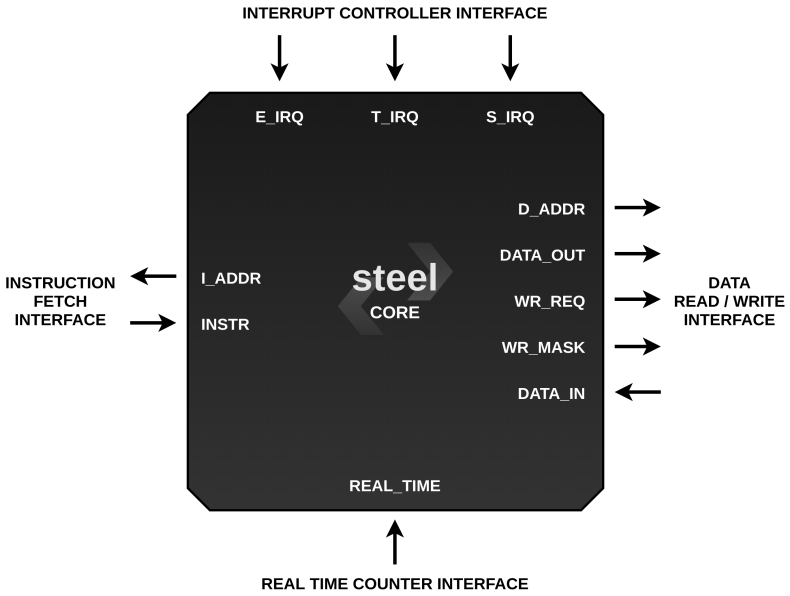
Steel has 4 communication interfaces, shown in the figure below.

The core was designed to be connected to a memory with one clock cycle read/write latency, which means that the memory should take one clock cycle to complete both read and write operations.

The interrupt request interface has signals to request for external, timer and software interrupts, respectively. They can be connected to a single device or to an interrupt controller managing interrupt requests from several devices. If your system does not need interrupts you should hardwire these signals to zero.

The real-time counter interface provides a 64-bit bus to read the value from a real-time counter and update the time register. If your system does not need hardware timers, you should hardwire this signal to zero.

Figure 3 – Steel Core input and output signals



2.1 Instruction fetch interface

The instruction fetch interface has two signals used in the instruction fetch process, shown in Table 1. The process of fetching instructions is explained in section 3.1.

Table 1 – Instruction fetch interface signals

Signal	Width	Direction	Description
INSTR	32 bits	Input	Contains the instruction fetched from memory.
I_ADDR	32 bits	Output	Contains the address of the instruction the core wants to fetch from memory.

2.2 Data read/write interface

The data read/write interface has five signals used in the process of reading/writing data from/to memory. The signals are shown in Table 2. The process of fetching data from memory is explained in section 3.2. The process of writing data is explained in section 3.3.

2.3 Interrupt controller interface

The interrupt controller interface has three signals used to request external, timer and software interrupts, shown in Table 3. The interrupt request process is explained in sections 3.4 and 4.

Table 2 – Data read/write interface signals

Signal	Width	Direction	Description
DATA_IN	32 bits	Input	Contains the data fetched from memory.
D_ADDR	32 bits	Output	In a write operation, contains the address of the memory position where the data will be stored. In a read operation, contains the address of the memory position where the data to be fetched is. The address is always aligned on a four byte boundary (the last two bits are always zero).
DATA_OUT	32 bits	Output	Contains the data to be stored in memory. Used only with write operations.
WR_REQ	1 bit	Output	When high, indicates a request to write data. Used only with write operations.
WR_MASK	4 bits	Output	Contains a mask of four *byte-write enable* bits. A bit high indicates that the corresponding byte must be written. See section 3.3 for details. Used only with write operations.

Table 3 – Interrupt controller interface signals

Signal	Width	Direction	Description
E_IRQ	1 bit	Input	When high indicates an external interrupt request.
T_IRQ	1 bit	Input	When high indicates a timer interrupt request.
S_IRQ	1 bit	Input	When high indicates a software interrupt request.

2.4 Real time counter interface

The real time counter interface has just one signal used to update the `time` CSR, shown in Table 4. The process of updating the `time` register is explained in section 3.5.

Table 4 – Real time counter interface

Signal	Width	Direction	Description
REAL_TIME	64 bits	Input	Contains the current value read from a real time counter.

2.5 CLK and RESET signals

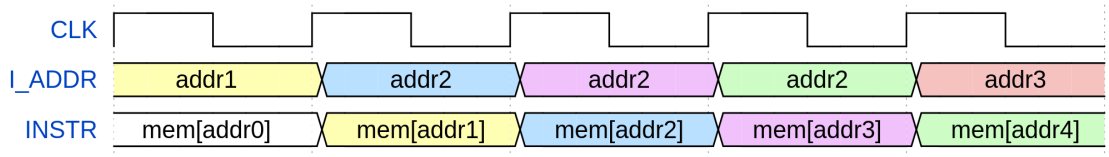
The core has CLK and RESET input signals, which were not shown in figure 3 (above). The CLK signal must be connected to a clock source. The RESET signal is active high and resets the core synchronously.

3 Timing diagrams

3.1 Instruction fetch

To fetch an instruction, the core places the instruction address on the I_ADDR bus. The memory must place the instruction on the INSTR bus at the next clock rising edge. Fig. 4 shows the timing diagram of this process. In the figure, `mem[addrX]` denotes the instruction stored at the memory position `addrX`.

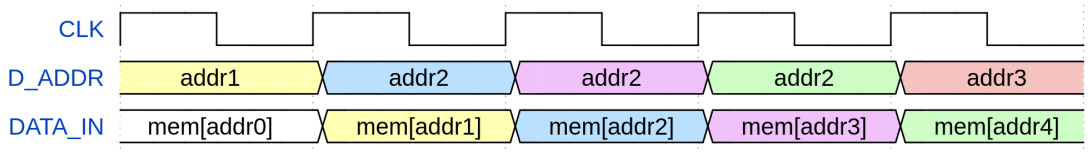
Figure 4 – Instruction fetch timing diagram



3.2 Data fetch

To fetch data from memory, the core puts the data address on the D_ADDR bus. The memory must place the data on the DATA_IN bus at the next clock rising edge. Fig. 5 shows the timing diagram of this process. In the figure, `mem[addrX]` denotes the data stored at the memory position `addrX`.

Figure 5 – Data fetch timing diagram



3.3 Data writing

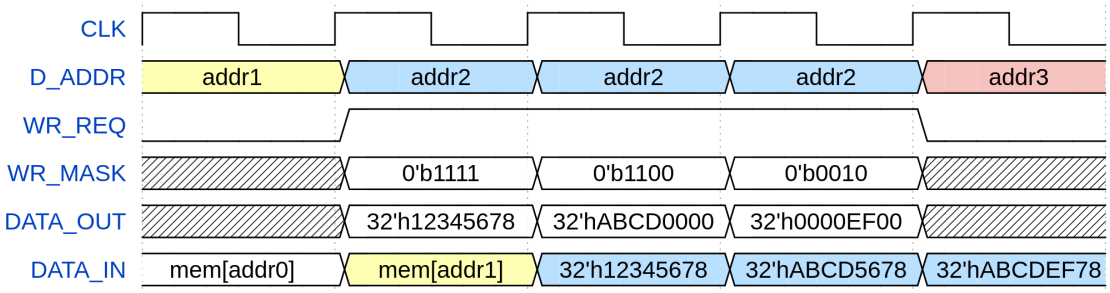
To write data to memory, the core drives D_ADDR, DATA_OUT, WR_REQ and WR_MASK signals as follows:

- D_ADDR receives the address of the memory position where the data must be written;
- DATA_OUT receives the data to be written;
- WR_REQ is set high;
- WR_MASK receives a byte-write enable mask that indicates which bytes of DATA_OUT must be written.

The memory must perform the write operation at the next clock rising edge. The core can request to write bytes, halfwords and words.

Fig. 6 (next page) shows the process of writing data to memory. DATA_IN is not used in the process and appears only to show the memory contents after writing. The figure shows five clock cycles, in which the core requests to write in the second, third and fourth cycles. In the second clock cycle, the core requests to write the word `0x12345678` at the address `addr2`. In the third, requests to write the halfword `0xABCD` at the upper half of `addr2`, and in the fourth requests to write the byte `0xEF` at the second least significant byte of `addr2`. The content of `addr2` after each of these operations appears on the DATA_IN bus and are highlighted in blue.

Figure 6 – Data writing timing diagram

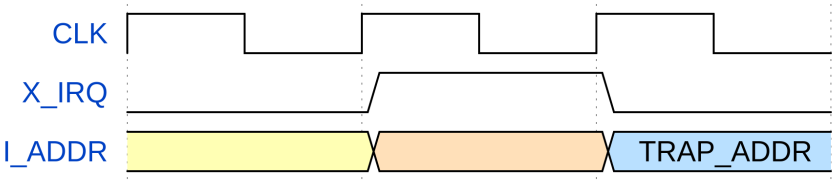


3.4 Interrupt request

An external device (or an interrupt controller managing several devices) can request interrupts by setting high the appropriate IRQ signal, which is E_IRQ for external interrupts, T_IRQ for timer interrupts and S_IRQ for software interrupts. The IRQ signal of the requested interrupt must be set high for one clock cycle and set low for the next.

Fig. 7 shows the timing diagram of the interrupt request process. Since the process is the same for all types of interrupt, X_IRQ is used to denote E_IRQ, T_IRQ or S_IRQ. TRAP_ADDR denotes the address of the trap handler first instruction.

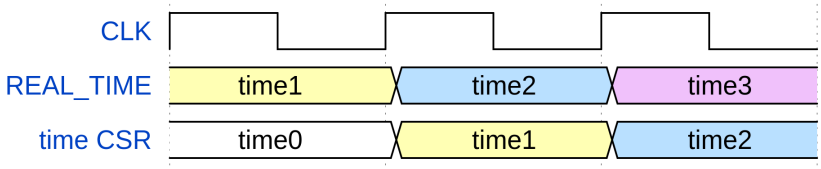
Figure 7 – Interrupt request timing diagram



3.5 Time CSR update

When connected to a real-time counter, the core updates the `time` CSR with the value placed on REAL_TIME at each clock rising edge, as shown in Fig. 8 . In the figure, `timeX` denotes arbitrary time values.

Figure 8 – `time` CSR update timing diagram



4 Exceptions and Interrupts

4.1 Supported exceptions and interrupts

Steel supports the exceptions and interrupts shown in Table 5. They are listed in descending priority order (the highest priority is at the top of the table). If two or more exceptions/interrupts occur at the same time, the one with the highest priority is taken.

Exceptions always cause a trap to be taken. An interrupt will cause a trap only if enabled. Each type of interrupt has an interrupt-enable bit in the `mie` register. Interrupts are globally enable/disabled by setting the MIE bit of `mstatus` register.

Table 5 – Steel supported exceptions and interrupts

	`mcause` **value**	
Exception / Interrupt	**Interrupt**	**Exception code**
Machine external interrupt	1	11
Machine software interrupt	1	3
Machine timer interrupt	1	7
Illegal instruction exception	0	2
Instruction address-misaligned exception	0	0
Environment call from M-mode exception	0	11
Environment break exception	0	3
Store address-misaligned exception	0	6
Load address-misaligned exception	0	4

4.2 Trap handling in Steel

Exceptions and interrupts are handled by a trap handler routine stored in memory. The address of the trap handler first instruction is configured using the `mtvec` register. Steel supports both direct and vectorized interrupt modes.

When a trap is taken, the core proceeds as follows:

- the address of the interrupted instruction (or the instruction that encountered the exception) is saved in the `mepc` register;
- the value of the `mtval` register is set to zero;
- the value of the `mstatus` MIE bit is saved in the MPIE field and then set to zero;
- the program counter is set to the trap handler first instruction.

The `mret` instruction is used to return from traps. When executed, the core proceeds as follows:

- the value of the `mstatus` MPIE bit is saved in the MIE field and then set to one;
- the program counter is set to the value of `mepc` register.

4.3 Nested interrupts capability

The core globally disables interrupts when takes into a trap. The trap handler can re-enable interrupts by setting the `mstatus` MIE bit to one, enabling nested interrupts. To return from nested traps, the trap handler must stack and manage the values of the `mepc` register in memory.

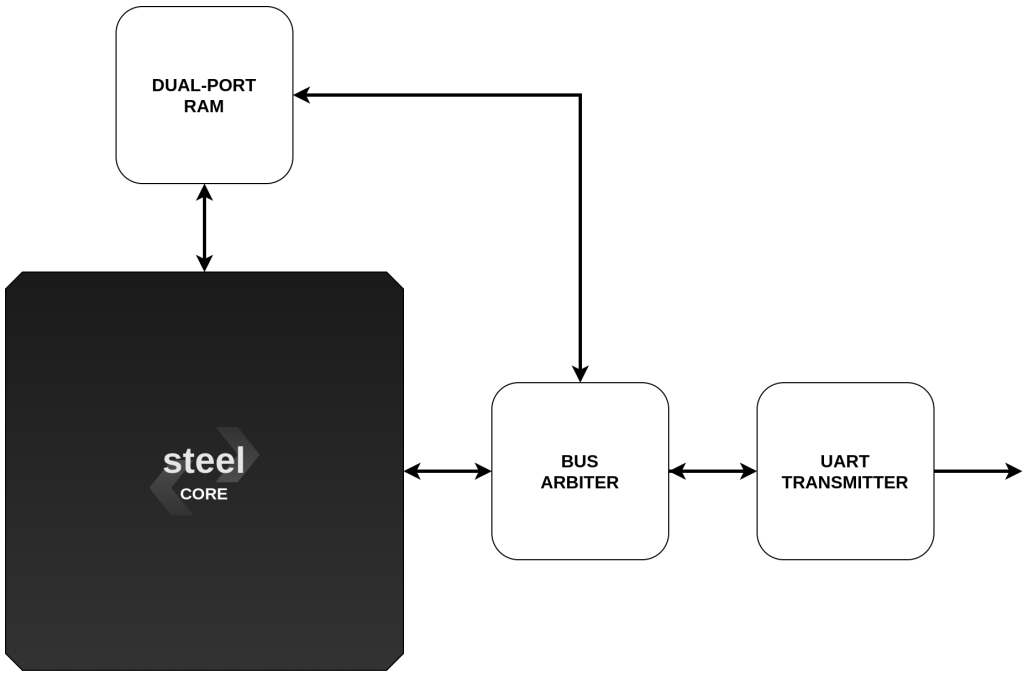
5 Example system built with Steel

The figure below shows an example system built with Steel, composed of an RAM memory array, a memory mapped UART transmitter, a bus arbiter and, of course, the Steel Core. The timer and interrupt request signals are hardwired to zero because neither timers nor interrupts are needed in this system. The implementation files of this system are inside the **soc** directory in the project repository. The **util** directory has an example program (**hello.c**) for this system. The program sends the string ”Hello World, Steel!” through the UART transmitter.

Note that the RAM memory and the UART transmitter share the interface to read/write data. The bus arbiter is used to multiplex this interface signals according to the address the core wants to access. In this example, the address 0x00010000 is used to access the UART transmitter. RAM addresses start at 0x00000000 and end at 0x00001fff. All other addresses are invalid.

The RISC-V GNU Toolchain provides the Newlib cross-compiler, which can be used to compile software for Steel. Instructions to install it can be found in the toolchain repository (available at `https://github.com/riscv/riscv-gnu-toolchain`).

Figure 9 – Steel-based system example



6 Implementation details

This section contains information on implementation details. It is intended for those who want to know more about how Steel works.

6.1 Implemented control and status registers

The control and status registers implemented in Steel are shown in Table 6. Other M-mode registers not shown in the table return the hardwired value defined by the RISC-V specifications when read.

Table 6 – Steel Core implemented CSRs

CSR	Name	Address
cycle	*Cycle Counter*	`0xC00`
time	*System Timer*	`0xC01`
instret	*Instructions Retired*	`0xC02`
mstatus	*Machine Status*	`0x300`
misa	*Machine ISA*	`0x301`
mie	*Machine Interrupt Enable*	`0x304`
mtvec	*Machine Trap Vector*	`0x305`
mscratch	*Machine Scratch*	`0x340`
mepc	*Machine Exception Program Counter*	`0x341`
mcause	*Machine Cause*	`0x342`
mtval	*Machine Trap Value*	`0x343`
mip	*Machine Interrupt Pending*	`0x344`
mcycle	*Machine Cycle Counter*	`0xB00`
minstret	*Machine Instructions Retired*	`0xB01`
mcountinhibit	*Machine Counter Inhibit*	`0x320`

6.2 Modules

6.2.1 Decoder

The Decoder (`decoder.v`) decodes the instruction and generates the signals that control the memory, the Load Unit, the Store Unit, the ALU, the two register files (Integer and CSR), the Immediate Generator and the Writeback Multiplexer. The description of its input and output signals are shown in Table 7.

Table 7 – Decoder input/output signals

Signal name	Width	Direction	Description
OPCODE_6_TO_2	5 bits	Input	Connected to the instruction *opcode* field.
FUNCT7_5	1 bit	Input	Connected to the instruction *funct7* field.
FUNCT3	3 bits	Input	Connected to the instruction *funct3* field.
IADDER_OUT_1_TO_0	2 bits	Input	Used to verify the alignment of loads and stores.
TRAP_TAKEN	1 bit	Input	When high indicates that a trap will be taken in the next clock cycle. Connected to the Machine Control module.
ALU_OPCODE	4 bits	Output	Selects the operation to be performed by the ALU. See Table 9 for possible values.
MEM_WR_REQ	1 bit	Output	When high indicates a request to write to memory.
LOAD_SIZE	2 bits	Output	Indicates the word size of load instruction. See Table 14.
LOAD_UNSIGNED	1 bit	Output	Indicates the type of load instruction (signed or unsigned). See Table 14.
ALU_SRC	1 bit	Output	Selects the ALU 2nd operand.
IADDER_SRC	1 bit	Output	Selects the Immediate Adder 2nd operand.
CSR_WR_EN	1 bit	Output	Controls the WR_EN input of CSR Register File.
RF_WR_EN	1	Output	Controls the WR_EN input of Integer Register File. See Table 11.
WB_MUX_SEL	3	Output	Selects the data to be written in the Integer Register File.
IMM_TYPE	3	Output	Selects the immediate based on the type of the instruction.
CSR_OP	3	Output	Selects the operation to be performed by the CSR Register File (read/write, set or clear).
ILLEGAL_INSTR	1 bit	Output	When high indicates that an invalid or not implemented instruction was fetched from memory.
MISALIGNED_LOAD	1 bit	Output	When high indicates an attempt to read data in disagreement with the memory alignment rules.
MISALIGNED_STORE	1 bit	Output	When high indicates an attempt to write data to memory in disagreement with the memory alignment rules.

6.2.2 ALU

The ALU (`alu.v`) applies ten distinct logical and arithmetic operations in parallel to two 32-bit operands, outputting the result selected by `OPCODE`. ALU input and output signals are shown in Table 8, and opcodes are shown in Table 9.

The opcode values were assigned to facilitate instruction translation. The most significant bit of `OPCODE` matches with the second most significant bit in the instruction *funct7* field. The remaining three bits match with the instruction *funct3* field.

Table 8 – ALU signals

Signal name	Width	Direction	Description
OP_1	32 bits	Input	Operation first operand.
OP_2	32 bits	Input	Operation second operand.
OPCODE	4 bits	Input	Operation code. This signal is driven by *funct7* and *funct3* instruction fields.
RESULT	32 bits	Output	Result of the requested operation.

Table 9 – ALU opcodes

Opcode	Operation	Binary value
ALU_ADD	Addition	4'b0000
ALU_SUB	Subtraction	4'b1000
ALU_SLT	Set on less than	4'b0010
ALU_SLTU	Set on less than unsigned	4'b0011
ALU_AND	Bitwise logical AND	4'b0111
ALU_OR	Bitwise logical OR	4'b0110
ALU_XOR	Bitwise logical XOR	4'b0100
ALU_SLL	Logical left shift	4'b0001
ALU_SRL	Logical right shift	4'b0101
ALU_SRA	Arithmetic right shift	4'b1101

6.2.3 Integer Register File

The Integer Register File (`integer_file.v`) has 32 general-purpose registers and supports read and write operations. Reads are requested in the pipeline stage 2 and provide data from one or two registers. Writes are requested in the pipeline stage 3 and put the data coming from the Writeback Multiplexer into the selected register. If stage 3 requests to write to a register being read by stage 2, the data to be written is immediately forwarded to stage 2. Each operation is driven by a distinct set of signals, shown in tables 10 and 11.

Table 10 – Integer Register File signals for read operation

Signal name	Width	Direction	Description
RS_1_ADDR	5 bits	Input	*Register source 1 address*. The data is placed at RS_1 immediately after an address change.
RS_2_ADDR	5 bits	Input	*Register source 2 address*. The data is placed at RS_2 immediately after an address change.
RS_1	32 bits	Output	Data read (source 1).
RS_2	32 bits	Output	Data read (source 2).

Table 11 – Integer Register File signals for write operation

Signal name	Width	Direction	Description
RD_ADDR	5 bits	Input	*Destination register address*.
RD	32 bits	Input	Data to be written in the destination register.
WR_EN	1 bit	Input	*Write enable*. When high, the data placed on RD is written in the destination register at the next clock rising edge.

6.2.4 Branch Unit

The Branch Unit (`branch_unit.v`) decides if a branch instruction must be taken or not. It receives two operands from the Integer Register File and, based on the value of *opcode* and *funct3* instruction fields, decides the branch. Jump instructions are interpreted as branches that must always be taken. Internally, the unit realizes just two comparisions, deriving other four from them. Table 12 shows the module input and output signals.

Table 12 – Branch Unit signals

Signal name	Width	Direction	Description
OPCODE_6_TO_2	5 bits	Input	Connected to the *opcode* instruction field.
FUNCT3	3 bits	Input	Connected to the *funct3* instruction field.
RS1	32 bits	Input	Connected to the register file 1st operand source.
RS2	32 bits	Input	Connected to the register file 2nd operand source.
BRANCH_TAKEN	1 bit	Output	High if the branch must be taken, low otherwise.

6.2.5 Load Unit

The Load Unit (`load_unit.v`) reads DATA_IN input signal and forms a 32-bit value based on the load instruction type (encoded in the *funct3* field). The formed value (placed on OUTPUT) can then be written in the Integer Register File. The module input and output signals are shown in Table 13. The value of OUTPUT is formed as shown in Table 14.

Table 13 – Load Unit signals

Signal name	Width	Direction	Description
LOAD_SIZE	2 bits	Input	Connected to the two least significant bits of the *funct3* instruction field.
LOAD_UNSIGNED	1 bit	Input	Connected to the most significant bit of the *funct3* instruction field.
DATA_IN	32 bits	Input	32-bit word read from memory.
IADDER_OUT_1_TO_0	2 bits	Input	Indicates the byte/halfword position in DATA_IN. Used only with load byte/halfword instructions.
OUTPUT	32 bits	Output	32-bit value to be written in the Integer Register File.

Table 14 – Load Unit output generation

LOAD_SIZE	**Effect on OUTPUT**
2'b00	The byte in the position indicated by IADDER_OUT_1_TO_0 is placed on the least significant byte of OUTPUT. The upper 24 bits are filled according to the LOAD_UNSIGNED signal.
2'b01	The halfword in the position indicated by IADDER_OUT_1_TO_0 is placed on the least significant halfword of OUTPUT. The upper 16 bits are filled according to the LOAD_UNSIGNED signal.
2'b10	All bits of DATA_IN are placed on OUTPUT.
2'b11	All bits of DATA_IN are placed on OUTPUT.
LOAD_UNSIGNED	**Effect on OUTPUT**
1'b0	The remaining bits of OUTPUT are filled with the sign bit.
1'b1	The remaining bits of OUTPUT are filled with zeros.

6.2.6 Store Unit

The Store Unit (`store_unit.v`) drives the signals that interface with memory. It places the data to be written (which can be a byte, halfword or word) in the right position in DATA_OUT and sets the value of WR_MASK in an appropriate way. Table 15 shows the unit input and output signals.

Table 15 – Store Unit signals

Signal name	Width	Direction	Description
FUNCT3	3 bits	Input	Connected to the *funct3* instruction field. Indicates the data size (byte, halfword or word).
IADDER_OUT	32 bits	Input	Contains the address (possibly unaligned) where the data must be written.
RS2	32 bits	Input	Connected to Integer Register File source 2. Contains the data to be written (possibly in the wrong position).
MEM_WR_REQ	1 bit	Input	Control signal generated by the Control Unit. When high indicates a request to write to memory.
DATA_OUT	32 bits	Output	Contains the data to be written in the right position.
D_ADDR	32 bits	Output	Contains the address (aligned) where the data must be written.
WR_MASK	4 bits	Output	A bitmask that indicates which bytes of DATA_OUT must be written. For more information, see section 3.3.
WR_REQ	1 bit	Output	When high indicates a request to write to memory.

6.2.7 Immediate Generator

The Immediate Generator (`imm_generator.v`) rearranges the immediate bits contained in the instruction and, if necessary, sign-extends it to form a 32-bit value. The unit is controlled by the IMM_TYPE signal, generated by the Control Unit. Table 16 shows the unit input and output signals.

Table 16 – Immediate Generator signals

Signal name	Width	Direction	Description
INSTR	25 bits	Input	Connected to the instruction bits (32 to 7).
IMM_TYPE	2 bits	Input	Control signal generated by the Control Unit that indicated the type of immediate that must be generated.
IMM	32 bits	Output	32-bit generated immediate.

6.2.8 CSR Register File

The CSR Register File (`csr_file.v`) has the control and status registers required for the implementation of M-mode. Read/write, set and clear operations can be applied to the registers. Table 17 shows the unit input and output signals, except those used for communication with the Machine Control, which are shown in Table 18.

Table 17 – CSR Register File signals

Signal name	Width	Direction	Description
WR_EN	1 bit	Input	*Write enable*. When high, updates the CSR addressed by CSR_ADDR at the next clock rising edge according to the operation selected by CSR_OP.
CSR_ADDR	12 bits	Input	Address of the CSR to read/write/modify.
CSR_OP	3 bits	Input	Control signal generated by the Control Unit. Selects the operation to be performed (read/write, set, clear or no operation).
CSR_UIMM	5 bits	Input	*Unsigned immediate*. Connected to the five least significant bits from the Immediate Generator output.
CSR_DATA_IN	32 bits	Input	In write operations, contains the data to be written. In set or clear operations, contains a bit mask.
PC	32 bits	Input	*Program counter* value. Used to update the `mepc` CSR.
E_IRQ	1 bit	Input	*External interrupt request*. Used to update the `MEIP` bit of `mip` CSR.
T_IRQ	1 bit	Input	*Timer interrupt request*. Used to update the `MTIP` bit of `mip` CSR.
S_IRQ	1 bit	Input	*Software interrupt request*. Used to update the `MSIP` bit of `mip` CSR.
REAL_TIME	64 bits	Input	Current value of the real time counter. Used to update the `time` and `timeh` CSRs.
CSR_DATA_OUT	32 bits	Output	Contains the data read from the CSR addressed by CSR_ADDR.
EPC_OUT	32 bits	Output	Current value of the `mepc` CSR.
TRAP_ADDRESS	32 bits	Output	Address of the trap handler first instruction.

Table 18 – CSR Register File and Machine Control interface signals

Signal name	Width	Direction[1]	Description
I_OR_E	1 bit	Input	*Interrupt or exception*. When high indicates an interrupt, otherwise indicates an exception. Used to update the most significant bit of `mcause` register.
CAUSE_IN	4 bits	Input	Contains the exception code. Used to update the `mcause` register. See Table 5.
SET_CAUSE	1 bit	Input	When high updates the `mcause` register with the values of I_OR_E and CAUSE_IN.
SET_EPC	1 bit	Input	When high, updates the `mepc` register with the value of PC.
INSTRET_INC	1 bit	Input	When high enables the instructions retired counting.
MIE_CLEAR	1 bit	Input	When high sets the MIE bit of `mstatus` to zero (which globally disables interrupts). The old value of MIE is saved in the `mstatus` MPIE field.
MIE_SET	1 bit	Input	When high sets the MPIE bit of `mstatus` to one. The old value of MPIE is saved in the `mstatus` MIE field.
MIE	1 bit	Output	Current value of MIE bit of `mstatus` CSR.
MEIE_OUT	1 bit	Output	Current value of MEIE bit of `mie` CSR.
MTIE_OUT	1 bit	Output	Current value of MTIE bit of `mie` CSR.
MSIE_OUT	1 bit	Output	Current value of MSIE bit of `mie` CSR.
MEIP_OUT	1 bit	Output	Current value of MEIP bit of `mip` CSR.
MTIP_OUT	1 bit	Output	Current value of MTIP bit of `mip` CSR.
MSIP_OUT	1 bit	Output	Current value of MSIP bit of `mip` CSR.

[1] Direction regarding to the CSR Register File. An input of CSR Register File is an output of Machine Control and vice-versa.

6.2.9 Machine Control

The Machine Control module (`machine_control.v`) implements the M-mode, controlling the the program counter generation and updating several CSRs. It has a special communication interface with the CSR Register File, already shown in Table 18 (above). Its input and output signals are shown in Table 19.

Internally, the module implements the finite state machine shown in figure 10 (next page).

Table 19 – Machine Control module signals

Signal name	Width	Direction	Description
ILLEGAL_INSTR	1 bit	Input	*Illegal instruction*. When high indicates that an invalid or not implemented instruction was fetched from memory.
MISALIGNED_INSTR	1 bit	Input	*Misaligned instruction*. When high indicates an attempt to fetch an instruction which address is in disagreement with the memory alignment rules.
MISALIGNED_LOAD	1 bit	Input	*Misaligned load*. When high indicates an attempt to read data in disagreement with the memory alignment rules.
MISALIGNED_STORE	1 bit	Input	*Misaligned store*. When high indicates an attempt to write data to memory in disagreement with the memory alignment rules.
OPCODE_6_TO_2	5 bits	Input	Value of the *opcode* instruction field.
FUNCT3	3 bits	Input	Value of the *funct3* instruction field.
FUNCT7	7 bits	Input	Value of the *funct7* instruction field.
RS1_ADDR	5 bits	Input	Value of the *rs1* instruction field.
RS2_ADDR	5 bits	Input	Value of the *rs2* instruction field.
RD_ADDR	5 bits	Input	Value of the *rd* instruction field.
E_IRQ	1 bit	Input	*External interrupt request*.
T_IRQ	1 bit	Input	*Timer interrupt request*.
S_IRQ	1 bit	Input	*Software interrupt request*.
PC_SRC	2 bit	Output	Selects the program counter source.
FLUSH	1 bit	Output	Flushes the pipeline when set.
TRAP_TAKEN	1 bit	Output	When high indicates that a trap will be taken in the next clock cycle.

Figure 10 – M-mode finite state machine