CNL_RISC-V AFTAB Microprocessor

User Manual

Release: April 2022

Proprietary Notice

The present document offers information subject to the terms and conditions described hereinafter.

The authors reserve the possibility to change the content and information described in this document and to update such information at any time, without notice. Despite the attention that has been taken in preparing this document, typographical errors, error or omissions may have occurred.

Authors

Maryam RAJABALIPANAH *(PhD candidate, University of Tehran)* m.rajabalipanah@ut.ac.ir
Zahra JAHANPEIMA *(PhD candidate, University of Tehran)* jahanpeima@ut.ac.ir
Simone PERACCINI *(Student, Politecnico di Torino)* simone.peraccini@studenti.polito.it
Mahboobe SADEGHIPOUR *(PhD candidate, Politecnico di Torino)* mahboobe.sadeghipourrudsari@polito.it
Gianluca ROASCIO *(PhD candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Zainalabedin NAVABI *(Full Professor, University of Tehran)* navabi@ut.ac.ir
Paolo PRINETTO *(Director, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it

Disclaimer

Acknowledgments

This work is the result of a fruitful collaboration and friendship between the CINI Cybersecurity National Laboratory and the University of Tehran, whose inspirers are certainly Prof. Paolo Prinetto and Prof. Zainalabedin Navabi.
The organization of the repository, as well as most of the code pertaining to the software toolchain (e.g., makefiles, linker scripts, boot code, etc.), is freely inspired by or taken from the PULPino project[1] by ETH Zurich and the University of Bologna, which we wish to thank for the valuable knowledge offered.

[1] https://github.com/pulp-platform/pulpino

Revision History

Release	Date	Changes
001	April 2022	First public agreed release version

Contents

1. Introduction

This document presents and describes AFTAB, an in-order RISC-V processor core implementing the RV32IM instruction set architecture (ISA). RISC-V project starts in 2010 at UC Berkeley under the direction of Prof. David Patterson[2]. The ISA is provided under open-source licenses that do not require fees to be used. The CINI Cybersecurity National Laboratory, in cooperation with the University of Tehran, aims at implementing a RISC-V core for academic purposes.

AFTAB is a 32-bit microprocessor with multi-cycle architecture. It has been designed in Register-Transfer Level (RTL) including datapath and controller. AFTAB datapath includes a number of combinational and sequential units and its controller consists of several states responsible for handling instruction execution and interrupts. During instruction execution, the states are performed serially, therefore it features no pipelining. Given that pipelining is not implemented, no branch prediction mechanism or unit has been included. Moreover, no caching has been implemented. AFTAB is the abbreviation form for “A Fine Turin/Tehran Architectural Being”.

This implementation supports the following extensions:

- **RV32I**: Base Integer Instruction Set;
- **RV32M**: Base Integer Multiplication and Division Instruction Set Extension;
- **Zicsr**: Control and Status Register Instructions.
- **N**: Standard Extension for User-Level Interrupts

Figure 1 shows AFTAB pinout:

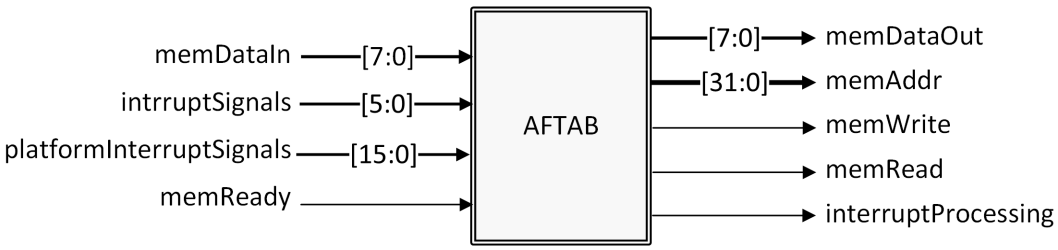

Figure 1: Pinout of the AFTAB microprocessor.

The description of the interfacing signals in Figure 1 is in Table 1.

Signals	Description
`clk, rst`	Fundamental clock and active high asynchronous reset
Set 1	Memory Interface
`memReady`	Ready signal for memory
`memRead`	Read signal for memory
`memWrite`	Write signal for memory
`memAddr`	32-bit address bus for addressing the memory
`memDataIn, memDataOut`	Two 8-bit ports for data input and data output from/to the memory. During load and store, they carry variable number of bytes (B,H,W) in multiple clock cycles.
Set 2	Interrupt Interface
`interruptSignals`	A set of 6 single-bit interrupt sources
`platformInterruptSignals`	16 interrupt lines for platform use
`interruptProcessing`	Indicating that the processor is in the interrupt processing states

Table 1: AFTAB interfacing port.

[2] Patty M. Sailer, Philip M. Sailer and David R. Kaeli. *The DLX instruction set architecture handbook*. Morgan Kaufmann Publishers Inc., 1996.

Figure 2 shows the high-level abstraction of the datapath and controller.

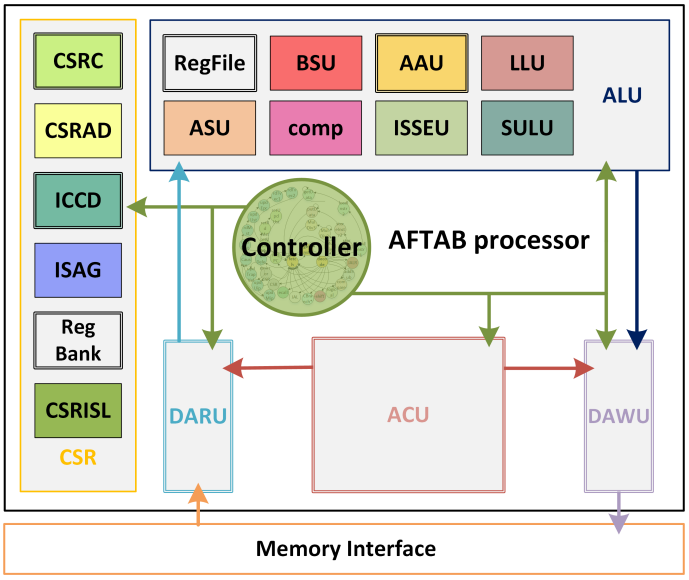

Figure 2: High-level abstraction of the datapath and controller.

AFTAB is thought to work in a Von Neumann architecture. Therefore, the basic environment used includes the core and one memory only, following a little-endian convention.

2. Overview

RISC-V has been originally designed for academic purposes, but in the recent years, also industries has announced hardware based on this standard. In the context of CNL_RISC-V, the project born within the Turin node of the Cybersecurity National Laboratory, the standard allowed us to make design and experiment with a proven and freely-available Instruction Set Architecture (ISA). Among the RISC-V advantages listed in the Specification Manual[3], the present project focuses on:

- Completely open ISA that is freely available to academia and industry;
- Separation into a small base integer ISA (RV32I) and optional standard extensions, to support general-purpose software development;
- The ISA avoids "over-architecting" for a particular microarchitecture style or implementation technology (e.g., full-custom, ASIC, FPGA), but allows efficient implementation in any of these;
- Simplification of experiments with privileged architecture design. Instructions divided into those are generally usable in all privilege modes (Unprivileged) and the ones requiring certain level of privilege (Privileged).

A first description of any RISC-V is implementation is given by the software Execution Environment Interface (EEI), that defines the initial state of the program as well as the number and type of *harts*. From the perspective of software running in a given environment, a hart is a resource that autonomously fetches and executes RISC-V instructions. The also includes the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions on each hart, and the handling of any interrupts or exceptions raised during execution including environment calls.

In our case, the execution environment is defined by the core hardware platform and it starts at power-on reset. Since the core has to be extended to run an operating system and the instruction have full access to the physical address space, this implementation can be considered as a *bare-metal* EEI.

2.1. RISC-V Instruction Set Architecture (ISA)

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISAs are very similar to that of the early RISC processors, except with no branch delay slots and with support for optional variable-length instruction encodings. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and so provides a convenient ISA and software tool-chain "skeleton" around which more customized processor ISAs can be built.

Although it is convenient to speak of the RISC-V ISA, RISC-V actually groups 4 base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers:

- RV32I and RV64I are the integer variants which provide 32-bit or 64-bit address spaces respectively;
- RV32E subset variant of the RV32I base instruction set, added to support small microcontrollers, and which has half the number of integer registers;

[3] https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

- RV128I variant of the base integer instruction set supports a flat 128-bit address space;

All the base integer instruction sets use a two's-complement representation for signed integer values.

Since our academic purpose does not require 64-bit address, only RV32I has been implemented. Integer ISA extension can be implemented in case it is thought to use the core in larger systems. To support more general software development, a set of standard extensions are defined to provide integer multiply/divide, atomic operations, and single/double-precision floating-point arithmetic:

- **The base integer ISA** is named "I" (prefixed by RV32 or RV64 depending on integer register width), and contains integer computational instructions, integer loads, integer stores, and control-flow instructions;
- **Integer multiplication and division extension** is named "M", and adds instructions to multiply and divide values held in the integer registers;
- **Standard atomic instruction extension** is denoted by "A", adds instructions that atomically read, modify, and write memory for inter-processor synchronization;
- **Standard single-precision floating-point extension** is denoted by "F", adds floating-point registers, single-precision computational instructions, and single-precision loads and stores;
- **Standard double-precision floating-point extension** is denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores;
- **Standard compressed instruction extension** is denoted by "C", provides narrower 16-bit forms of common instructions;

2.2. Supported Instruction Set Architecture (ISA)

This section describes the RISC-V Base Integer Instruction Set (RV32I), the Integer Multiplication and Division Extension (RV32M) and "$Zicsr$" Control and Status Register Instructions. As explained in the previous part, it has been thought as a good starting base for our academic intent.

RV32I has been designed as an essential base for a compiler target and for the execution of modern operating systems. The internal state of base integer ISA is given by 32 unprivileged general-purpose registers and the privileged Program Counter (PC). The first holds values stored to be given as input to the execution unit, while the second stores the address of the current instruction.

AFTAB supports 52 instructions encoded in 32 bits, where the 6 least significant bits are reserved as Operation Code ($opcode$). The instructions are grouped in 4 different formats (R, I, U and S), as shown in Figure 3.

In RISC-V ISA, the source ($rs1$ and $rs2$) and destination (rd) registers are at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions, immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 3.

The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits ($imm[10:1]$) and the sign bit stay in fixed positions, while the lowest bit in S format ($inst[7]$) encodes a high-order bit in B format.

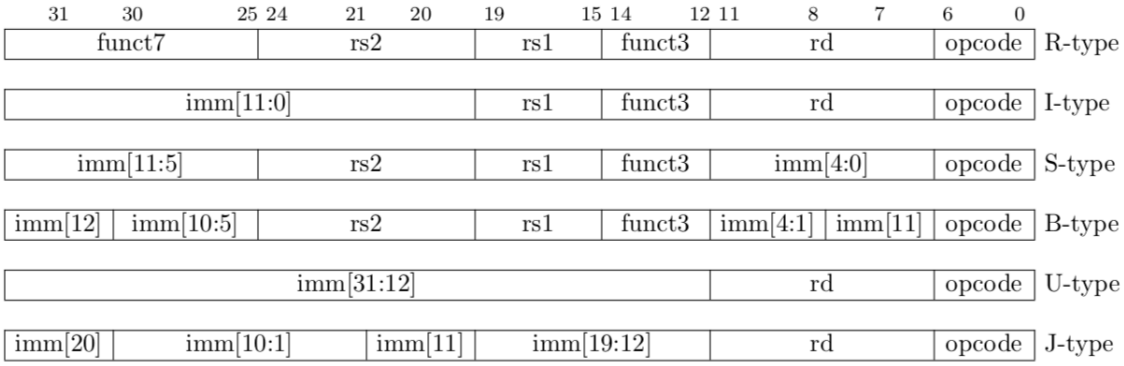

Figure 3: RISC-V base instruction formats showing immediate variants.

An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not 4-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

RV32M is the standard integer multiplication and division instruction extension, which is named "M" and contains instructions that multiply or divide values held in two integer registers.

`mul` performs a 32-bit×32-bit multiplication of $rs1$ by $rs2$ and places the lower 32 bits in the destination register. **`mulh`**, **`mulhu`**, and **`mulhsu`** perform the same multiplication but return the upper 32 bits of the full 64-bit product, for signed×signed, unsigned×unsigned, and signed $rs1$×unsigned $rs2$ multiplication, respectively.

`div` and **`divu`** perform a 32-bit signed and unsigned integer division of $rs1$ by $rs2$, rounding towards zero. **`rem`** and **`remu`** provide the remainder of the corresponding division operation. For **`rem`**, the sign of the result equals the sign of the dividend. For both signed and unsigned division, it holds that *dividend = divisor × quotient + remainder*. If both the quotient and remainder are required from the same division, the recommended code sequence is:

```
div[u] rdq, rs1, rs2
rem[u] rdr, rs1, rs2 # (rdq cannot be the same as rs1 or rs2).
```

"$Zicsr$" is the extension defining Control and Status registers (CSRs) Instructions. RISC-V defines a separate addressing for 4096 CSRs on which it can operate.

CSR instructions use I-Type instruction format and read,modify and write a single CSR. The CSR specifier is encoded in the 12-bit CSR field of the instruction($imm[31:20]$). The immediate forms use a 5-bit zero-extended immediate ($uimm[4:0]$) encoded in the $rs1$ field. A complete description of the implemented instructions and registers for this kind of instructions will be provided in Section 8.

In Table 2, the complete list of supported instruction is reported, together with their encoding parameters. The following legenda is to be applied:

- `rd`: destination register
- `rs1`: source register 1
- `rs2`: source register 2
- `csr`: generic Control and Status Register

- `[a:b]`: bit range notation (i.e., from bit a downto bit b)
- `uimm5`: unsigned 5-bit immediate
- `imm12`: signed 12-bit immediate
- `uimm12`: unsigned 12-bit immediate
- `imm20`: signed 20-bit immediate
- `uimm20`: unsigned 20-bit immediate
- `(s)`: the instruction is performed over signed numbers
- `(u)`: the instruction is performed over unsigned numbers
- `(su)`: the instruction is performed over source 1 as signed and source 2 as unsigned
- `(l)`: the destination is filled with the lowest part of the result
- `(h)`: the destination is filled with the highest part of the result
- &: bitwise logic AND
- |: bitwise logic OR
- ^: bitwise logic XOR
- !: bitwise logic NOT
- DMEM: data memory
- `sign_8_32()`: extend byte to 32-bit word according to sign
- `zero_8_32()`: extend byte to 32-bit word with zeros
- `sign_16_32()`: extend 16-bit word to 32-bit word according to sign
- `zero_16_32()`: extend 16-bit word to 32-bit word with zeros
- §: concatenation operator
- `<<l`: shift logical left
- `>>l`: shift logical right
- `>>a`: shift arithmetic right

Category	*opcode*	*funct3*	*funct7*	Type	Instruction Example	Meaning
Arithmetic	0x33	0x0	0x00	*R*	**add** rd, rs1, rs2	rd= rs1 + rs2
	0x33	0x0	0x20	*R*	**sub** rd, rs1, rs2	rd = rs1 − rs2
	0x13	0x0	/	*I*	**addi** rd, rs1, imm12	rd = rs1 + imm12
	0x33	0x2	0x00	*R*	**slt** rd, rs1, rs2	rd = 1 if rs1 < rs2 else 0 (s)
	0x33	0x3	0x00	*R*	**sltu** rd, rs1, rs2	rd = 1 if rs1 < rs2 else 0 (u)
	0x13	0x2	/	*I*	**slti** rd, rs1, imm12	rd = 1 if rs1 < imm12 else 0 (s)
	0x13	0x3	/	*I*	**sltiu** rd, rs1, uimm12	rd = 1 if rs1 < uimm12 else 0 (u)
Mul and Div	0x33	0x0	0x01	*R*	**mul** rd, rs1, rs2	rd = rs1 ∗ rs2 (s) (l)
	0x33	0x1	0x01	*R*	**mulh** rd, rs1, rs2	rd = rs1 ∗ rs2 (s) (h)
	0x33	0x2	0x01	*R*	**mulhsu** rd, rs1, rs2	rd = rs1 ∗ rs2 (su) (h)
	0x33	0x3	0x01	*R*	**mulhu** rd, rs1, rs2	rd = rs1 ∗ rs2 (u) (h)
	0x33	0x4	0x01	*R*	**div** rd, rs1, rs2	rd = rs1 / rs2 (s)
	0x33	0x5	0x01	*R*	**divu** rd, rs1, rs2	rd = rs1 / rs2 (u)
	0x33	0x6	0x01	*R*	**rem** rd, rs1, rs2	rd = rs1 / rs2 (s)
	0x33	0x7	0x01	*R*	**remu** rd, rs1, rs2	rd = rs1 % rs2 (u)
Data transfer	0x03	0x2	/	*I*	**lw** rd, imm12(rs1)	rd = DMEM[rs1 + imm12]
	0x23	0x2	/	*S*	**sw** rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2
	0x03	0x1	/	*I*	**lh** rd, imm12(rs1)	rd = sign_16_32(DMEM[rs1 + imm12])
	0x03	0x1	/	*I*	**lhu** rd, imm12(rs1)	rd = zero_16_32(DMEM[rs1 + imm12])
	0x23	0x1	/	*S*	**sh** rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2[15:0]
	0x03	0x0	/	*I*	**lb** rd, imm12(rs1)	rd = sign_8_32(DMEM[rs1 + imm12])
	0x03	0x0	/	*I*	**lbu** rd, imm12(rs1)	rd = zero_8_32(DMEM[rs1 + imm12])
	0x23	0x0	/	*S*	**sb** rs2, imm12(rs1)	DMEM[rs1 + imm12] = rs2[7:0]
Logical	0x33	0x7	0x00	*R*	**and** rd, rs1, rs2	rd = rs1 & rs2
	0x33	0x6	0x00	*R*	**or** rd, rs1, rs2	rd = rs1 | rs2
	0x33	0x4	0x00	*R*	**xor** rd, rs1, rs2	rd = rs1 ^ rs2
	0x13	0x7	/	*I*	**andi** rd, rs1, 20	rd = rs1 & 20
	0x13	0x6	/	*I*	**ori** rd, rs1, 20	rd = rs1 ! 20
	0x13	0x4	/	*I*	**xori** rd, rs1, 20	rd = rs1 ^ 20
Shift	0x33	0x1	0x00	*R*	**sll** rd, rs1, rs2	rd = rs1 <<l rs2[4:0]
	0x33	0x5	0x00	*R*	**srl** rd, rs1, rs2	rd = rs1 >>l rs2[4:0]
	0x33	0x5	0x20	*R*	**sra** rd, rs1, rs2	rd = rs1 >>a rs2[4:0]
	0x13	0x1	0x00	*I*	**slli** rd, rs1, uimm5	rd = rs1 <<l uimm5
	0x13	0x5	0x00	*I*	**srli** rd, rs1, uimm5	rd = rs1 >>l uimm5
	0x13	0x5	0x20	*I*	**srai** rd, rs1, uimm5	rd = rs1 >>a uimm5
Conditional branch	0x63	0x0	/	*B*	**beq** rd, rs1, imm12	if (rd = rs1)goto pc + imm12 (s)
	0x63	0x1	/	*B*	**bne** rd, rs1, imm12	if (rd != rs1)goto pc + imm12 (s)
	0x63	0x4	/	*B*	**blt** rd, rs1, imm12	if (rd < rs1)goto pc + imm12 (s)
	0x63	0x5	/	*B*	**bge** rd, rs1, imm12	if (rd >= rs1)goto pc + imm12 (s)
	0x63	0x6	/	*B*	**bltu** rd, rs1, imm12	if (rd < rs1)goto pc + imm12 (u)
	0x63	0x7	/	*B*	**bgeu** rd, rs1, imm12	if (rd >= rs1)goto pc + imm12 (u)
Unconditional branch	0x6F	/	/	*J*	**jal** rd, imm12	rd = pc + 4 goto pc + imm12
	0x67	/	/	*J*	**jalr** rd, imm12(rs1)	rd = pc + 4 goto rs1 + imm12
lui and **auipc**	0x37	/	/	*U*	**lui** rd, uimm20	rd[31:12] = uimm20 rd[11:0] = 0
	0x17	/	/	*U*	**auipc** rd, uimm20	rd = pc + (uimm20 <<l 12)
System	0x73	0x0	0x00	*I*	**ecall**	pc = mtvec + (4 ∗ mcause) mepc = pc
	0x73	0x0	0x18	*S*	**mret**	pc = mepc
	0x73	0x0	0x00	*S*	**uret**	pc = uepc
	0x73	0x1	/	*I*	**csrrw** rd, csr, rs1	rd = csr csr = rs1
	0x73	0x2	/	*I*	**csrrc** rd, csr, rs1	rd = csr csr = csr & !rs1
	0x73	0x3	/	*I*	**csrrs** rd, csr, rs1	rd = csr csr = csr | rs1
	0x73	0x5	/	*I*	**csrrwi** rd, csr, uimm5	rd = csr csr = uimm5
	0x73	0x6	/	*I*	**csrrci** rd, csr, uimm5	rd = csr csr = csr & !uimm5
	0x73	0x7	/	*I*	**csrrsi** rd, csr, uimm5	rd = csr csr = csr | uimm5

Table 2: RV32IM and $Zicsr$ Extension Instructions supported by AFTAB.

2.3. Functional Description

This section aims at providing details on the basic behavior of the overall AFTAB architecture, including its datapath and controller. AFTAB datapath provides a set of combinational and sequential units, customized to perform the supported instructions. AFTAB controller first fetches the instruction, then decodes the instruction, and finally selects the appropriate execution states. In each state of the controller, a set of proper control signals become issued to activate the appropriate units and paths in the datapath. After the reset, the execution states is repeated until exceptions and interrupts. The following Sections provide a detailed description of every single unit inside the datapath and every state in the controller. The overall behavior of AFTAB is summarized in the following set of states:

1. **Fetch states**: aiming at loading the new Program Counter (PC), selecting the address to load into Instruction Register (IR) and adjusting the instruction read from memory. This tasks are performed in two different states, named `Fetch` and `GetInstr`. `Fetch` state loads the Program Counter, while in `GetInstr` state the Data Adjustment Read Unit dedicated (DARU) handles the instruction adjusting;

2. **Decode states**: aiming at extracting the instruction fields to be used as operands to execute instructions. This are addresses for reading from register file and immediate values. The remaining fields are extracted by the Controller Unit, that checking $opcode$ and $funct$ fields configures the datapath in order to perform a specific instruction;

3. **Execution states**: aiming at performing the actual computation needed to perform a specific instruction. After being selected, the input operands are used by one of the units in the execute states to perform an arithmetic computation or updating the core state. Also based on the type of instruction, reading and writing data from and to memory is needed. Since memory is byte-addressable, this operation has to be performed in multiple clock cycles, and requires data adjustment that is performed by Data Adjustment Read Unit (DARU) and Data Adjustment Write Unit (DAWU); Also, writing to the Register File at the end of some execution states is required.

4. **Interrupt/Exception processing states**: aiming at following interrupt entry and exit procedures according to RISC-V specifications. Providing proper values for CSR registers and filling the PC with the proper value are the most important tasks in these states.

2.4. Privilege Levels

Since AFTAB wants to operate in the *secure embedded system* application field, the implementation of privileged levels and instruction is required. As basic concept, it should be considered that every RISC-V hardware thread runs at some privilege level, encoded in the Control and Status Registers (CSR). Figure 4 shows RISC-V privilege modes.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Figure 4: RISC-V privilege levels.

Privilege levels are used to provide protection between different components of the software stack, and attempt to perform operations not permitted by the current privilege mode cause an exception to be raised. These exceptions normally cause traps into an underlying execution environment.

The machine level has the highest privilege and is the only mandatory privilege level for a RISC-V hardware platform. Code run in *machine mode* (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. *User mode* (U-mode) and *supervisor mode* (S-mode) are intended for conventional application and operating system usage respectively.

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine.

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

Figure 5: Supported combinations of privilege modes.

As shown in Figure 5, RISC-V supports three different combinations of privilege modes: machine (M), machine-user (M,U) and machine-supervisor-user (M,S,U). A hart normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, that in our implementation may run in M-mode. The hart then executes the trap handler, which eventually resumes execution at (or after) the original trapped instruction in U-mode. Traps increasing privilege level are termed *vertical traps*, while traps causing no privilege elevation are termed *horizontal traps*. The RISC-V privileged architecture provides flexible routing of traps to different privilege layers.

At the current version, AFTAB is intended to be used as a Secure embedded system, as it implements M-mode and U-mode. In order to run an OS, S-mode should be added to the design. Debug mode is not provided.

2.5. Exceptions

According to the RISC-V specifications, AFTAB rises exceptions in the following cases:

- Illegal instruction for illegal opcode, illegal CSR instruction or division by zero;
- Instruction address misalignment;

In all above cases the `exceptionRaise` signal is raised immediately and the controller enters the exception handling states which are `checkDelegation`, `updateTrapeValue`, `updateCause`, `updateEPC`, `readTvec1` and `readTvec2`, `readMstatus`, `updateMstatus`, `updateUstatus` in sequence.

The datapath units also prioritize the exceptions, provide the cause code and trap value information for the granted exception, determine delegation mode, generate the start address of the exception handler through the Interrupt Vector Table (IVT).

2.6. Interrupts

AFTAB supports External, Software and Timer interrupts for both Machine and User mode. Beside these 6 lines, a set of 16 interrupt lines for the platform uses are provided. In such cases `interruptRaise` signal is raised and the controller enters the interrupt procssing states which

are `checkDelegation`, `updateMip`, `updateUip`, `updateCause`, `updateEPC`, `readTvec1` and `readTvec2`, `readMstatus`, `updateMstatus`, `updateUstatus` in sequence.

The datapath units again prioritize the interrupts, provide the cause code for the granted interrupt, determine delegation mode, generate the start address of the interrupt handler through the Interrupt Vector Table (IVT).

2.7. Programmer's Model

This Section is intended to show the AFTAB software model available to programmer for programs execution. As previously anticipated, it implements the RISC-V Base Integer Instruction Set (RV32I), the Integer Multiplication and Division Extension (RV32M) and "$Zicsr$" Control and Status Register Instructions. AFTAB also supports privilege execution (U-mode and M-Mode) and has an Interrupt Vector Table (IVT). As starting point, for running both machine-only and machine-user executions, Appendix A explains how to setup privilege execution.

When writing a program for AFTAB, a programmer may choose to provide both an Assembly source file (.s) or a C/C++ source file. Since the address bus is on 32 bit, the **`.text`** section for instructions and the **`.data`** section for variables can reach up to 4 GB. Actually, for allowing fast simulation even without resorting to professional licenses for simulator or server equipment, the addressing space has been truncated to 8 KB.

General-purpose registers are labeled from `x0` to `x31`. The first register, `x0`, is hardwired to 0, i.e., always returns 0 when read and always ignores write. Registers from `x1` to `x31` are all equally general-use registers as far as the processor is concerned, but the RISC-V programming convention assigns specific roles to most of them. In the mnemonic RISC-V Assembly language, they are given standardized names as part of the RISC-V application binary interface (ABI).

In the RISC-V standard register convention, there are the **saved registers** `s0` to `s11`, that are used to store data that must be preserved across function calls. Then, the **argument registers** `a0` to `a7` are used to pass arguments, and the **temporary registers** `t0` to `t6` are used inside a function for computation. The convention also defines specialized registers such as the Stack Pointer `sp`, the Global Pointer `gp`, the Thread Pointer `tp` and Link Register `ra`. Register naming convention is listed in Table 3.

RISC-V also provides standard *pseudo-instructions*, that do not have a direct machine equivalent but are translated by the mnemonic Assembly in machine language instructions. An example can be the register "copy" pseudo-instruction, coded as:

```
mv rd, rs
```

but actually translated into the machine language instruction as:

```
addi rd, rs, 0
```

A list of the RISC-V pseudo-instruction is in Table 4.

RISC-V does not support `pop` and `push` instructions, and they are performed through simple load and store to stack memory region. Whenever the processor is intended to use the stack, the programmer is strongly recommended to:

- Decide the stack size that should be used;
- Decrement `x2` (conventionally used as `sp`) by the decided size;
- Incrementally store and load form the top of the stack area;
- Restore the stack value adding the used size.

The memory can be used by the programmer according to its own convention: RISC-V standard does not specify fixed map or convention. At the current state, AFTAB does not support any memory protection or management mechanism.

Register	ABI	Use by convention	Preserved?
x0	zero	hardwired to 0	ignores writes
x1	ra	return address/link register	no
x2	sp	stack pointer	yes
x3	gp	global pointer	_n/a_
x4	tp	thread pointer	_n/a_
x5	t0	temporary register 0	no
x6	t1	temporary register 1	no
x7	t2	temporary register 2	no
x8	s0 or fp	saved register 0 or frame pointer	yes
x9	s1	saved register 1	yes
x10	a0	return value or function argument 0	no
x11	a1	return value or function argument 1	no
x12	a2	function argument 2	no
x13	a3	function argument 3	no
x14	a4	function argument 4	no
x15	a5	function argument 5	no
x16	a6	function argument 6	no
x17	a7	function argument 7	no
x18	s2	saved register 2	yes
x19	s3	saved register 3	yes
x20	s4	saved register 4	yes
x21	s5	saved register 5	yes
x22	s6	saved register 6	yes
x23	s7	saved register 7	yes
x24	s8	saved register 8	yes
x25	s9	saved register 9	yes
x26	s10	saved register 10	yes
x27	s11	saved register 11	yes
x28	t3	temporary register 3	no
x29	t4	temporary register 4	no
x30	t5	temporary register 5	no
x31	t6	temporary register 6	no
pc	(none)	program counter	_n/a_

Table 3: RV32I general register map.

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc x6, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, −1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.b rd, rs`	`slli rd, rs, XLEN − 8` `srai rd, rd, XLEN − 8`	Sign extend byte
`sext.h rd, rs`	`slli rd, rs, XLEN − 16` `srai rd, rd, XLEN − 16`	Sign extend byte
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`zext.b rd, rs`	`andi rd, rs, 255`	Zero extend byte
`zext.h rd, rs`	`slli rd, rs, XLEN − 16` `srli rd, rd, XLEN − 16`	Zero extend half word
`zext.w rd, rs`	`slli rd, rs, XLEN − 32` `srli rd, rd, XLEN − 32`	Zero extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if != zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if <zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if >zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if != zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if <zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if >zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine

Table 4: Listing of standard RISC-V pseudo-instructions.

3. AFTAB Simulation Environment Memory Map

As 32-bit RISC-V implementation, AFTAB has a virtual byte-addressable address space of 4 GB for all memory accesses. A word of memory is defined as 32 bits (4 bytes). Correspondingly, a halfword is 16 bits (2 bytes), a doubleword is 64 bits (8 bytes), and a quadword is 128 bits (16 bytes).

As already introduced, to ease simulation by users with no professional licenses for simulators or with no powerful hardware resources, the physical address space has been limited to 8 KB. The memory unit provided by this project automatically manages the conversion from virtual to physical.

AFTAB simulation environment currently uses the memory map in Figure 6 from which one can observe:

- **Instruction Memory**: 4 KB RAM with base address `0x00000000`.
- **Data Memory**: 1.5 KB RAM with base address `0x00100000`.
- **Stack**: 2.5 KB RAM with base address `0x00100600`.
- **Peripherals**: AFTAB simulation environment currently do not include any peripheral, and memory addresses are set by convention and for future use.

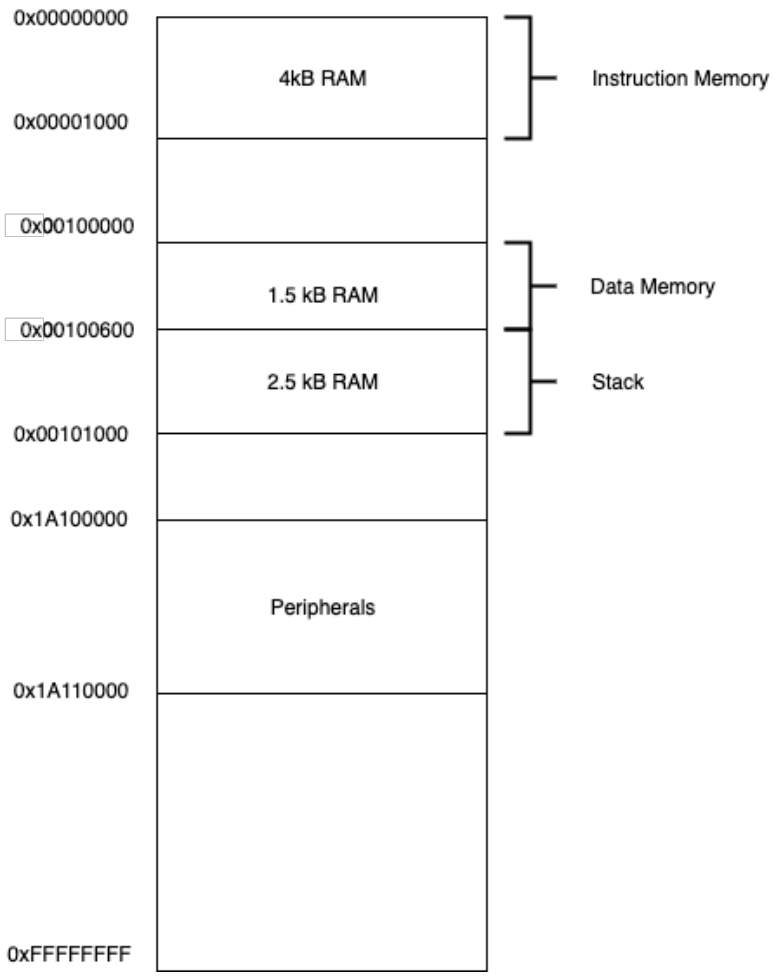

Figure 6: AFTAB simulation environment memory map.

4. Memory Access Protocols

This section describes the AFTAB memory access protocols while fetching instruction, loading and storing data. The memory interface ports are described in Table 5.

Signals	Description
Set 1	Memory Interface
`memReady`	Ready signal for memory
`memRead`	Read signal for memory
`memWrite`	Write signal for memory
`memAddr`	32-bit address bus for addressing the memory
`memDataIn, memDataOut`	Two 8-bit ports for data input and data output from/to the memory. During load and store, they carry variable number of bytes (B,H,W) in multiple clock cycles.

Table 5: AFTAB memory interface.

4.1. Instruction Fetch

The instruction fetcher is able to supply the Decode state with one instruction in 5 clock cycles. After one clock cycle, in which the Program Counter is updated, the instruction is read by the Data Adjustment Read Unit.

4.1.1. Protocol

In the following, the instruction read protocol is explained:

1. If memory is not performing operations, `memReady` signal is set to 1 and the read process can start.
2. If `memReady` is set, the DARU sets the output signal `memRead` and provides the instruction base address on (`memAddr`). At this point, the combinational read of the first byte is performed through the signal `memDataIn`;
3. If more than one byte is read, `memRead` signal is kept high and the address incremented by one. In case of an instruction, this operation takes 4 clock cycles.

The instruction bytes are adjusted inside the core by the DARU.

4.2. Load and Store

Load and store operations are also performed through the memory interface. Given that data bus are on 8 bits, memory operations are necessarily multi-cycle. In load and store operations, the clock cycle count changes according to the performed instruction: 4 clock cycles (CC) for reading/writing words, 2 CC for halfwords, and 1 CC for byte.

Read and write adjustment are performed by dedicated Data Adjustment Read Unit (DARU) and Data Adjustment Write Unit (DAWU).

4.2.1. Protocol

Here a detailed description of data read and write protocol is provided. The protocol used for reading is the same as the protocol explained in Section 4.1.1. The one used for writing is the following:

1. If memory is not performing operations, `memReady` signal is set to 1 and the write process can start.
2. If `memReady` is set, the DAWU sets the output signal `memWrite`, sets the write base address `memAddr` and sets the byte to be written on the data bus (`memDataOut`). The write of the first byte is then performed on memory;
3. If more than one byte has to be written, `memWrite` signal is kept high, the address incremented by one and the next byte set on the data bus.

5. Instruction Execution Flow

This Section aims at providing details on a basic behavior of the AFTAB microprocessor, going through each unit of the datapath (shown in the RTL diagram provided in the documentation files). When the reset signal is asserted, the following operations are performed:

- Reset of internal registers and flip-flops;
- Reset value of the PC;
- The first instruction is fetched.

After the initial state reset, the execution cycle is repeated infinitely unless it is blocked by exceptions or interrupts. In the following, the states the core may enter during the machine cycle are detailed:

- **Fetch states**: aiming at loading the program counter with the address of the new instruction, selecting the value to forward to the Instruction Register (IR) and adjusting the memory read for the memory. This tasks are performed in two different states named `Fetch` and and `GetInstr`:
  - `Fetch`: the fetch state is always performed in one clock cycle and aims at choosing the value to load in the PC, loading it and setting the control signals for the following state `GetInstr`. Starting from PC choice, the diagram shows that it can be selected through the 5 control signals `selAdd`, `selI4`,`selMepcCSR`, `selInterAddrDir`, `selInterAddrVec` between five different inputs:
    - branch or jump result (obtained adding current PC and immediate encoded in the instruction);
    - current PC incremented by 4;
    - Machine/User Exception Program Counter (MEPC/UEPC);
    - Machine/User Trap Vector Base Address (MTVEC/UTVEC).

    The Fetch also sets `startDARU` for starting the Data Adjustment Read Unit (DARU).
  - `GetInstr`: since only 32-bit instructions are implemented, this state always require 4 clock cycles to complete, each one of them reading an instruction byte. Starting from the PC address, the DARU unit reads the instruction increasing the address by 1 at each clock cycle, and adjusting the read value into a 32-bit output, forwarded to the the IR.
- **Decode state**: the decode state always lasts one clock cycle and aims at extracting the instruction fields from the Instruction Register and leading the controller to the proper execution states. The extracted fields are the following, and change according to the instruction type:
  - $opcode$ fixed at $IR[6:0]$;
  - Destination register (rd) fixed at $IR[11:7]$;
  - Register addres 1 ($rs1$) fixed at $IR[19:15]$;
  - Register addres 2 ($rs2$) fixed at $IR[24:20]$;
  - Immediate at different position according to instruction type provided by Controller Unit. For I-Type, it is taken from $IR[31:20]$, for S-Type from $IR[31:25]$ and for U-Type from $IR[31:12]$;

  - Extended unsigned immediate for CSR instructions;

  The remaining field, such as $funct3$ and $funct7$ are extracted by the Controller Unit. At the end of the clock cycle, the Register File (RF) syncronously updates the two read ports p1 and p2, while the immediate is computed by the Immediate Selection and Sign Extension Unit (ISSEU).

- **Execution states**: during the execute, the operands (p1 and p2) and immediate are ready to be used by all the units inside the execute state. The actual values provided to the computational units are selected by two multiplexers that selects the two inputs between the computed ones. A first multiplexer chooses the operand between:
  - output of the RF read port 1 (p1)
  - current value of the Program Counter

  A second multiplexer chooses between:
  - output of the read port 2 (p2)
  - the immediate computed by the ISSEU

  The execution state lasts a variable number of clock cycles that depends on the operation to perform. In particular,
  - **`jalr`** and **`jal`** require 1 clock cycle;
  - Conditional branches require 1 clock cycle;
  - Shift instructions require 1 clock cycle;
  - Bitwise logical instructions require 1 clock cycle;
  - Compare instructions require 1 clock cycle;
  - Addition/subtraction require 1 clock cycle;
  - CSR instructions require maximum 2 clock cycles;
  - The load instructions require maximum 6 clock cycles. The states `LoadInstr1` and `LoadInstr2` are required to compute the memory address and start reading from memory (`startDARU` to 1); Then the CU starts DARU, and enters the states `getData` , and keeps each state for a number of clock cycle that depends on the number of bytes exchanged from the memory. In particular, the clock cycles required for instruction **`lb`** are one, while two clock cycles are taken for **`sh`** and **`lh`** and four for **`lw`**;
  - The store instructions require maximus 6 clock cycles, The states `StoreInstr1` and `StoreInstr2` are required to compute the memory address, read the data register and start writing to memory (`startDAWU` to 1); Then the CU starts DAWU, and enters the states `putData`, and keeps each state for a number of clock cycle that depends on the number of bytes exchanged to the memory. In particular, the clock cycles required for instructions **`sb`** are one, while two clock cycles are taken for **`sh`** and four for **`sw`**;
  - The multiplication is implemented through Booth's algorithm that takes 33 clock cycles to perform 32-bit multiplication;
  - The division is performed through the restoring algorithm that takes 64 clock cycles to perform 32-bit division.

  Since there is no ALU output register: the operations result and data read from memory are directly written to the register file. The CU sets the signal `writeRegFile` and selects the data to be written through the select signals of `Mux10`.

6. Datapath

The aim of this section is to present the main components inside the AFTAB datapath, i.e.,:

- Generic datapath register;
- Register File;
- Immediate Selection and Sign Extension Unit (ISSEU);
- Comparator;
- Adder/Subtractor Unit (ASU);
- Logical Logic Unit (LLU);
- Barrel Shifter Unit (BSU);
- Attached Arithmetic Unit (AAU);
- Booth Multiplier;
- Signed/Unsigned Divider;
- Data Adjustment Read Unit (DARU);
- Data Adjustment Write Unit (DAWU);
- Signed/Unsigned Load Unit (SULU).
- Control and Status Registers Units.

The RTL diagram of the datapath is not reported here for readability reasons, and it can be found in file `aftab_datapath.pdf` in this folder.

6.1. Datapath Registers

Figure 7 shows the ports of the generic entity `aftab_register` used inside the datapath.

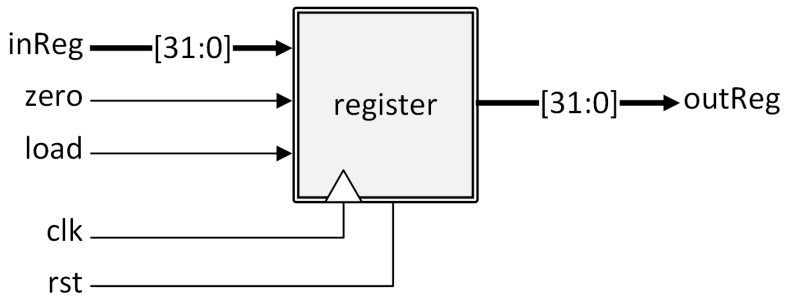

Figure 7: AFTAB Register.

Each register has asynchronous reset and a load signal that is set whenever the register content has to be updated. An additional signal allows to set it to 0 synchronously.

Excluding the 32 general-purpose registers of the Register File, AFTAB has 4 additional registers inside the datapath:

- Program Counter (PC), of 32 bit;
- Instruction Register (IR), of 32 bit;
- Memory Address (ADR), of 32 bit;
- Memory Data Register (DR), of 32 bit.

6.2. Register File

As previously anticipated, the AFTAB Register File (RF) is a synchronous component that features 32 general-purpose registers. Figure 8 shows its block diagram with its ports.

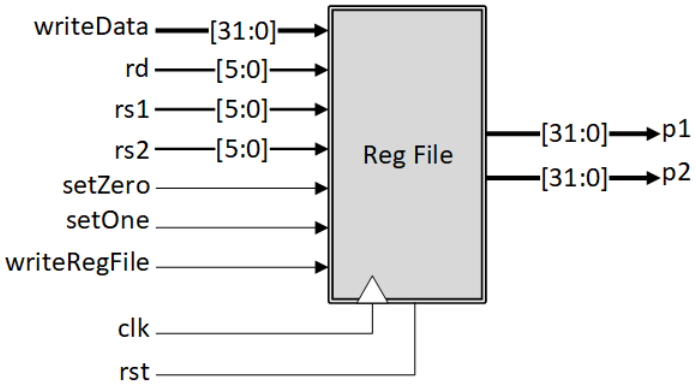

Figure 8: AFTAB Register File.

Both read and write operations are triggered by the rising edge of the clock. Read operations are always enabled, therefore no read enable signal are used. Port 1 (p1) and port 2 (p2) output the content of the registers at internal address `rs1` and `rs2`, respectively. Write operations can be performed setting to 1 three possible input signals: `writeRegFile`, `setZero` and `setOne`. In case `setZero` or `setOne` are set, the destination register is respectively written with all 0s or value 1. If the `writeRegFile` is set, the content of `rd` is written with the input value `writeData`.

Reset is performed synchronously setting the active high signal `rst`. When this operation is performed, all the registers are set to zero.

6.3. Immediate Selection and Sign Extension Unit (ISSEU)

AFTAB Immediate Selection and Sign Extension Unit (ISSEU) allows to prepare the immediate value encoded in the instruction register (IR) according to the instruction type given as input through the signals grouped in `sel` on 12 bits. Figure 9 shows the ISSEU ports, in which the immediate is the 32 bit output `imm`.

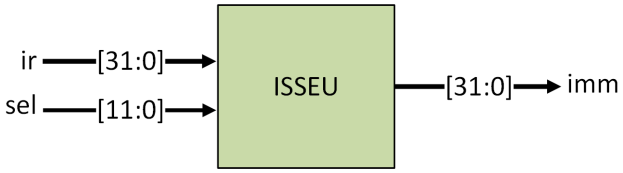

Figure 9: AFTAB Immediate Selection and Sign Extension Unit.

The immediate output is set according to the following configuration of the signal `sel`:

- For I type, `010011001001`;
- For S type, `010011010010`;
- For U type, `100000100100`;
- For J type, `101001001100`;
- For B type, `010101010100`.

6.4. Comparator

The Comparator is implemented as a behavioral entity that takes two input operands on 32 bit and outputs the result of three comparisons: *greater-than* (`gt`), *lower-than* (`lt`) and *equal-to* (`eq`). The signal `ComparedSignedUnsignedBar` allows to choose if the comparison is performed on signed or unsigned numbers. Figure 10 shows the interfacing ports.

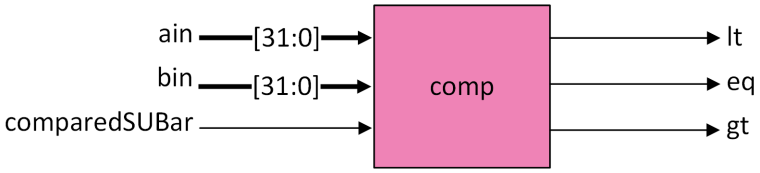

Figure 10: AFTAB Comparator.

6.5. Adder/Subtractor Unit (ASU)

The Adder and Subtractor Unit is able to perform addition or subtraction depending on what selected through the signal `addSubBar`. The operation uses as sources the two 32-bit inputs (`a` and `b`), and gives the output on 32 bit (`outRes`). The output signal `cout` is set in case of output carry. The component also allows to mask the operand `a` considering it as zero and operand `b` places on the output. This operation can be performed setting the `pass` signal to 1.

Figure 11 shows the interfacing ports.

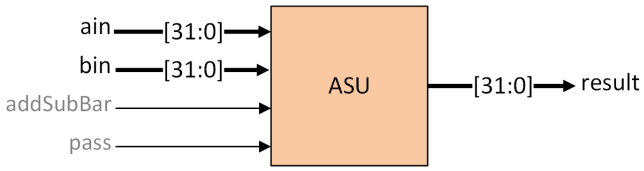

Figure 11: AFTAB Adder Subtractor.

6.6. Logical Logic Unit (LLU)

The Logical Logic Unit is able to perform three possible bitwise logical operations: XOR, OR and AND. Each one of them can be selected through the signal `selLogic`. The result of the bitwise operation is on 32 bit. Figure 12 shows the interfacing ports.

6.7. Barrel Shifter Unit (BSU)

The Barrel Shifter Unit is able to perform three different operations: Shift Left Logical (SLL), Shift Right Logical (SRL) and Shift Right Arithmetic (SRA). Each one of them can be selected through the signal `selShift`. Figure 13 shows the interfacing ports.

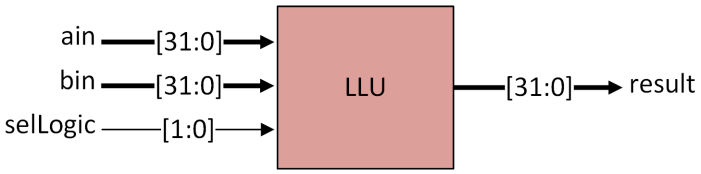

Figure 12: AFTAB Logical Logic Unit.

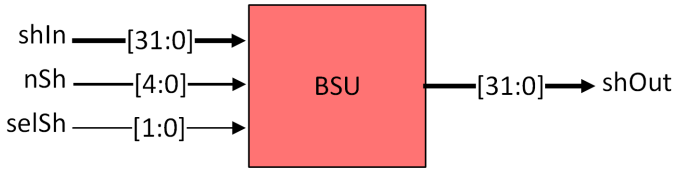

Figure 13: AFTAB Barrel Shifter Unit.

6.8. Attached Arithmetic Unit (AAU)

The Attached Arithmetic Unit (AAU) allows to perform both multiplication and division on 32 bit. The two operations are started setting the signals `startMultAAU` and `startDivideAAU` to 1, and are completed when the `completeAAU` signal is set. Both inputs of AAU are 32-bit operands. In order to specify if the operands has to be treated as signed or unsigned, the signals `signedSigned`, `unsignedUnsigned` and `signedUnsigned` can be set by the CU. In case the AAU is asked to perform a division by zero, the start signal is ignored and the signal `dividedByZeroFlag` is set to 1 for one clock cycle. Figure 14 shows the interfacing ports.

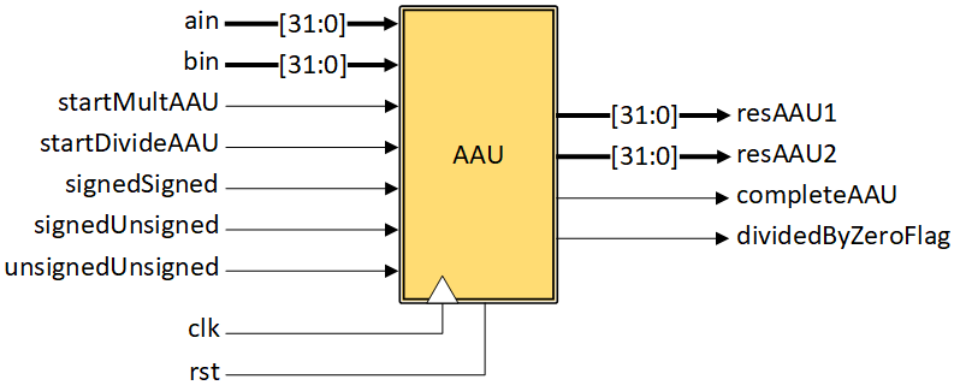

Figure 14: AFTAB Attached Arithmetic Unit.

6.8.1. Booth Multiplier

The multiplication is implemented exploiting the Booth Algorithm. It can perform booth signed and unsigned multiplications.Both multiplicand and multiplier are 32-bit inputs. According to control signals one bit for sign specification is added to each operand on the left side. So, multiplication is performed with 33-bit inputs. For 33-bit multiplication, it requires 33 clock cycles to complete. The general behavior of the algorithm is the following:

1. Assign multiplicand and multiplier to two registers (`M` and `Mr`);
2. Instantiate a register for the product (`pReg`) and concatenate a zero bit on the right position of `Mr`. `pReg` is set to zero;

3. Instantiate a *counter* to count from zero to n, that is the bit length of Mr;

4. Check the bit sequence Mr[1,0] and according to its value perform :

   - If 11 or 00, only perform an arithmetic shift right on pReg and Mr. The bit shifted out from pReg is the shift in for Mr;

   - If 01, M is added to pReg. After this, an arithmetic shift right (by 1) is performed on both pReg and Mr. The bit shifted out from pReg is the shift in for Mr;

   - If 10, M is subtracted to pReg. After this, an arithmetic shift right (by 1) is performed on both pReg and Mr. The bit shifted out from pReg is the shift in for Mr;

5. Repeat the previous steps n-1 times;

6. Final Result is obtained concatenating the final content of registers pReg and Mr[length ,1]. pReg is assigned to Res[63:32] and Mr to Res[31:0].

For the multiplier, an FSM and a separate datapath design has been chosen. Figures 15 and 16 show the multiplier's controller and datapath implementations.

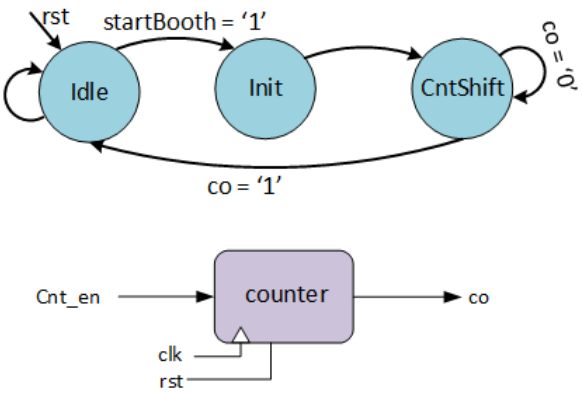

Figure 15: AFTAB Booth Multiplier controller.

As the finite state machine in Figure 15 shows, the multiplier controller operates in three different states: **Idle**, **Init**, and **Count_Shift**. **Idle** state is waiting for startBooth signal. **Init** is the initialization state in which M and Mr values are loaded. pReg is set to zero and the counter is reset. When startBooth is set to 1, the state is set to **Count_Shift**. In this state, the counter is incremented and control signals for the shift are set, the arithmetic shift right is performed until the counter is not zero. According to the value of the datapath, output control signal op, is set in order to define if and how pReg has to be updated.

6.8.2. Signed/Unsigned Divider

The AFTAB Division Unit internally uses the Restoring Algorithm for unsigned division and extends it to signed numbers thanks to a wrapper. When dividing, the unit is able to provide both quotient and remainder. The general behavior of the algorithm is the following:

1. Registers initialization to their corresponding values. The dividend is assigned to Q, the divisor to M, and result to 0. An additional counter set to the number of bits (n) is used to count the algorithm iteration;

2. The content of register R and Q is shifted left as if they are a single unit;

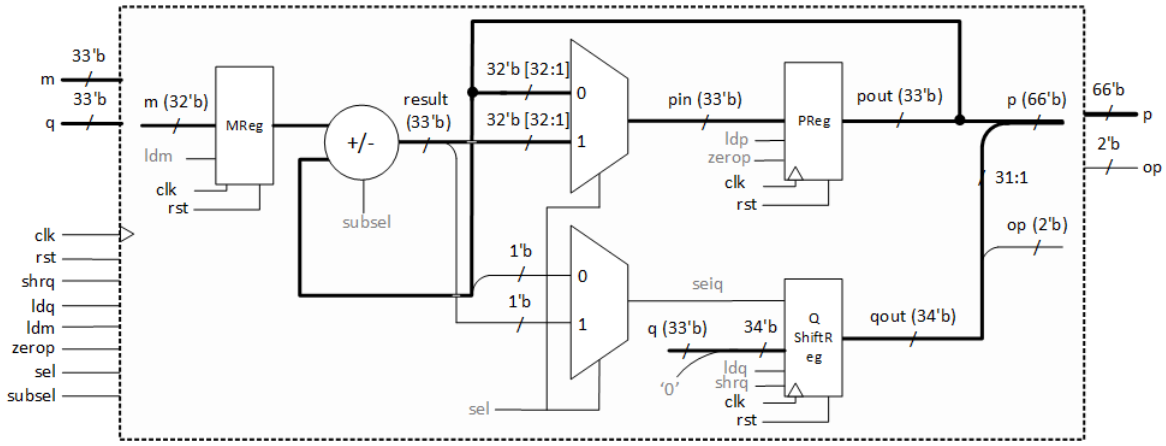

Figure 16: AFTAB Booth Multiplier datapath.

3. The content of register M is subtracted from R and result is stored in **sub**;
4. The most significant bit of the **sub** is checked: if it is 0, the least significant bit of Q is set to 1; otherwise, if it is 1, the least significant bit of Q is set to 0 and value of register R is restored (i.e., the value of **sub** before the subtraction with M);
5. The value of counter n is decremented;
6. If the value of n becomes 0, the loop ends, otherwise it repeats from step 2;
7. Finally, the register Q contain the quotient and R contain remainder.

For the divider, an FSM and a separate datapath design has been chosen as well. Figures 18, 19 and 17 show the divisor Datapath and Controller implementations. As it is outlined in the FSM in Figure 17, the divisor operates in three different states: **IDLE**, **STEP1** and **STEP2**. At reset time, the state is set to **IDLE**. Here, the registers for dividend (RegQ), reminder (RegR) and divisor (RegM) are initialized as well as the counter. This state is kept until `startDiv` signal is set to 1 and the state changes to **STEP1**. Here, the shift is performed. If the counter (i.e., `coCnt`) is not zero, the next state is **STEP2**, otherwise the division is concluded and the state is set back to **IDLE**. In **STEP2**, RegR is decremented by RegM and RegR content is updated. In addition, the counter is decremented and Q is updated according to the check performed on the most significant bit of RegR. From **STEP2**, the current state unconditionally changes to **STEP1**.

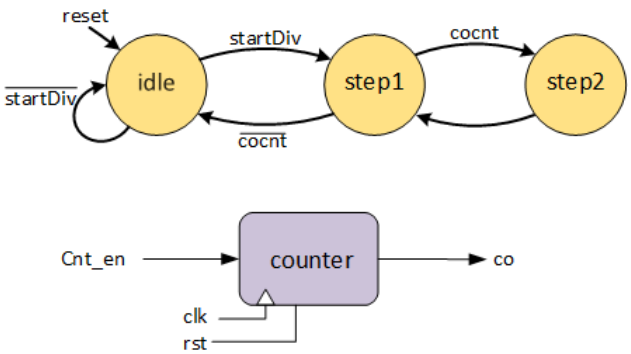

Figure 17: AFTAB Divider controller.

Figure 18 shows the implementation of unsigned divider with using of the Restoring Algorithm, while Figure 19 shows the wrapper that allows to select between signed and unsigned handling.

The wrapper simply determines the sign of remainder and quotient, and the unsigned divider performs the division on unsigned numbers.

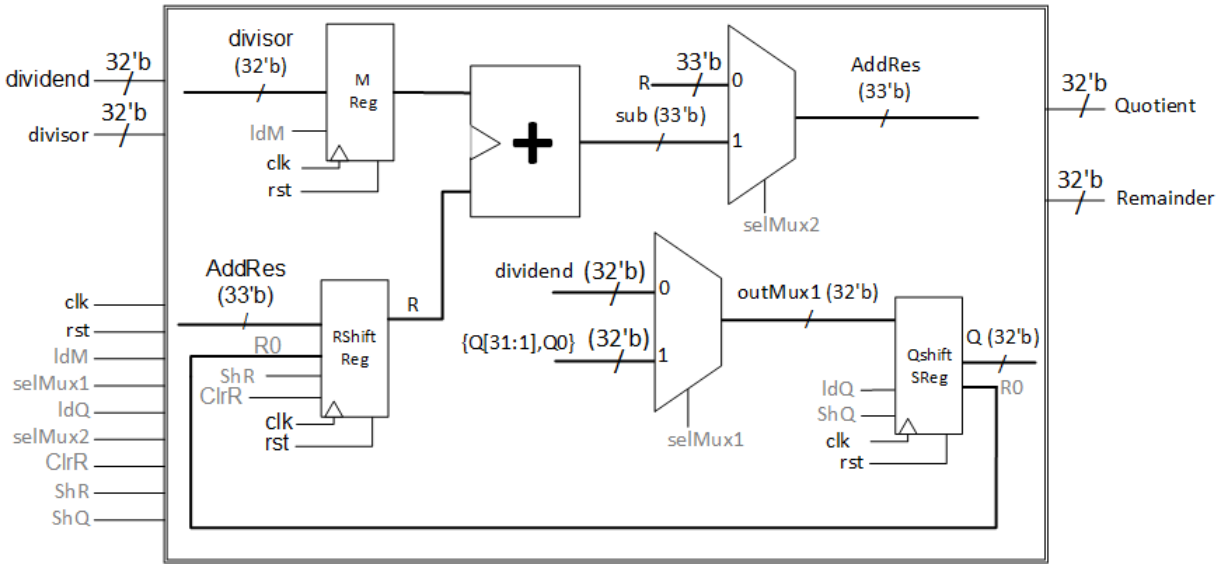

Figure 18: AFTAB Unsigned Divider datapath.

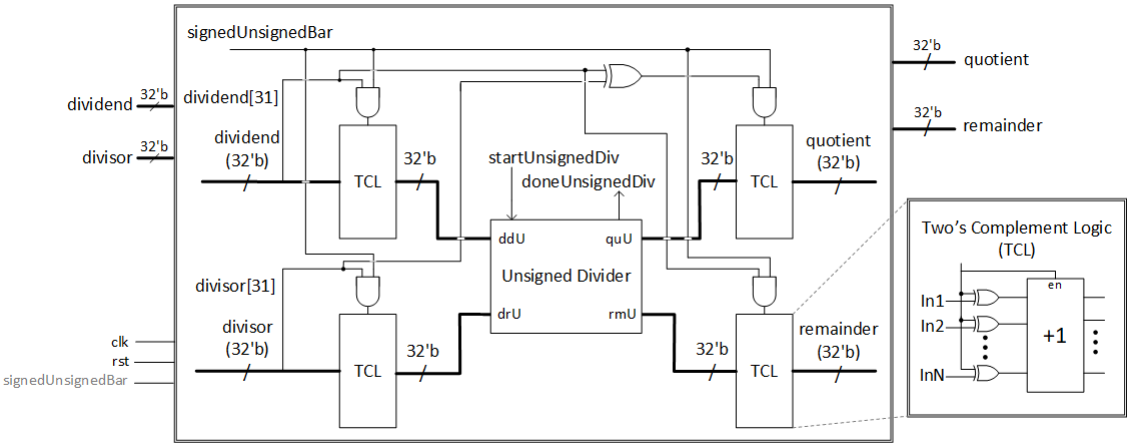

Figure 19: AFTAB Signed/Unsigned divider with wrapper.

6.9. Data Adjustment Read Unit (DARU)

The Data Adjustment Read Unit (DARU) aims at handling memory read operations, reading bytes from adjacent memory addresses and grouping them into a 32-bit word. Figure 20 shows the DARU interfacing ports.

- `clk` and `rst`: the internal state of the unit is updated at the rising edge of the clock and the reset is asynchronous;
- `addrIn`: 32-bit input for read operation start address. This input is taken from datapath Address register;
- `memData`: 8-bit data bus for data read from memory at every clock cycle. These will be internally adjusted by the DARU;

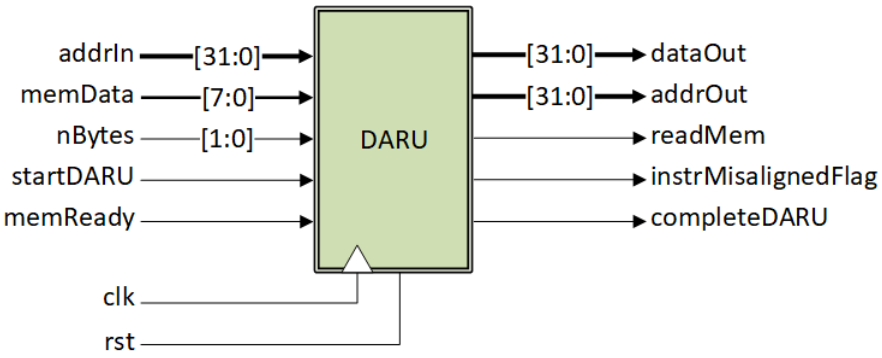

Figure 20: AFTAB Data Adjustment Read Unit.

- `nBytes`: 2-bit signal indicating number of bytes to be read from memory;
- `startDARU`: this signal starts the read process. During the clock cycle in which it is set, the DARU records the number of bytes of the operation and the start address;
- `dataOut`: output for the read value adjusted on 32 bits;
- `memReady`: if this input signal is set to 1, memory is ready to start an operation. The core has to wait in case it is set to zero;
- `completeDARU`: if it is set to 0, memory is performing an operation. When the operation is concluded, it is set to 1;
- `readMem`: output signal enabling memory read, performed on clock rising edge. It has to be set for the whole read process;
- `instrMisalignedFlag`: this signal notifies a misaligned address to the exception handling system[4].

6.10. Data Adjustment Write Unit (DAWU)

The Data Adjustment Write Unit aims at handling the memory write operation, splitting the data into bytes and writing each one of them into the proper memory address. Figure 21 shows the DAWU interfacing ports.

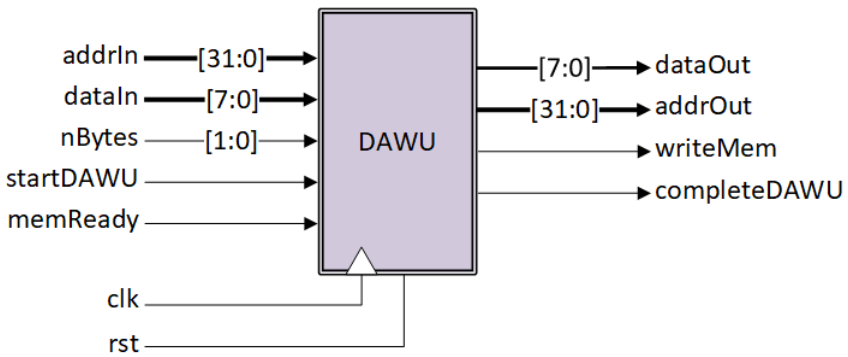

Figure 21: AFTAB Data Adjustment Write Unit.

[4]Only instruction misaligned address faults are considered. The corresponding signal in the DARU instance for data load is left open, as data alignment in memory might be up to the program.

- `clk` and `rst`: the internal state of the unit is updated at the rising edge of the clock and the reset is asynchronous;
- `addrIn`: 32-bit input for read operation start address. This input is taken from datapath Address register;
- `addrOut`: 32-bit for the actual address provided to memory. When write operations are performed in more than one clock cycle, the address is incremented by one at every rising edge;
- `dataIn`: 32-bit data bus for data to be written to memory. This is taken from data register output;
- `dataOut`: 8-bit for the actual byte data provided to memory;
- `nBytes`: 2-bit signal indicating number of bytes to be written to memory;
- `startDAWU`: this signal starts the write process. During the clock cycle in which it is set, the DAWU records the number of bytes of the operation, the start address and the content to be written;
- `memReady`: if this input signal is set to 1, memory is ready to start an operation. The core has to wait in case it is set to 0;
- `completeDAWU`: if it is set to 0, memory is performing an operation. When the operation is concluded it is set to 1;
- `writeMem`: output signal enabling memory write, performed on clock falling edge. It has to be set for the whole write process.
- `storeMisalignedFlag`: this signal notifies a misaligned address to the exception handling system[5].

6.11. Signed/Unsigned Load Unit (SULU)

The Signed/Unsigned Load Unit aims at selecting the output requested by a load instruction. The DARU always outputs 4 bytes form which the SULU chooses and adjusts the required ones. Figure 22 shows the SULU interfacing ports.

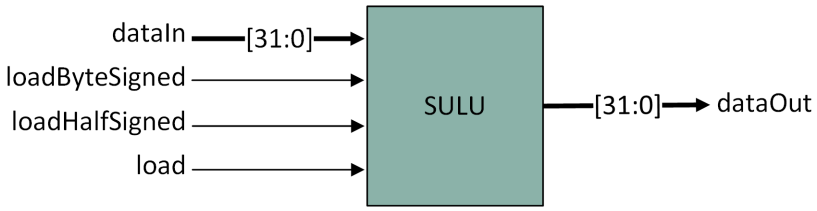

Figure 22: AFTAB Signed Unsigned Load Unit.

- `loadHalfSigned`: set if signed half word is loaded;
- `loadByteSigned`: set if signed byte is loaded;
- `load`: set if word is loaded;
- `dataIn`: 32-bit input taken from DARU output;
- `dataOut`: 32- bit output for adjusted data.

[5] Unused at the moment, as data alignment in memory might be up to the program.

6.12. Control and Status Register Units

The Control and Status Register Units are a set of components hosting the CSRs (see Section 8) implemented by the core. These units work as a simple memory bank allowing to read and write every single CSR according to their own read and write permissions. This operation can be performed through CSR instructions, in which each register can be read and updated within three clock cycles. CSR units also allow to handle exceptions and interrupts, requiring an entry phase that are started through the signal `interruptRaise` and an exit phase with using xRET instruction.

6.12.1. Register Bank

Register bank contains multiple units that are described below:

- CSR `registers`: at the current state of this implementation, AFTAB supports 16 registers from total CSRs. These registers implemented in CSR `registers unit` as a small memory bank. Input signals are `writeRegBank`, `addressRegBank` and `inputRegBank` .`OutRegBank` is output signal. Each of read and write operations takes one clock cycle.
- `Carbon Copy registers`: AFTAB implementation has three carbon copy registers from MIE, MIP, and mie field of MSTATUS register. MIP carbon copy acts as a synchronization register for interrupt source signals. Copy of MIE register and mie field are used for generation signal `interruptRaise` and cause detection. Carbon copy registers update when original registers in CSR bank change.
- `CSR Address Logic`: CSR address logic unit generates signals for load data in carbon copy mie field and MIE register, when specific addresses are detected.This unit also generates control signals for mirror CSRs.

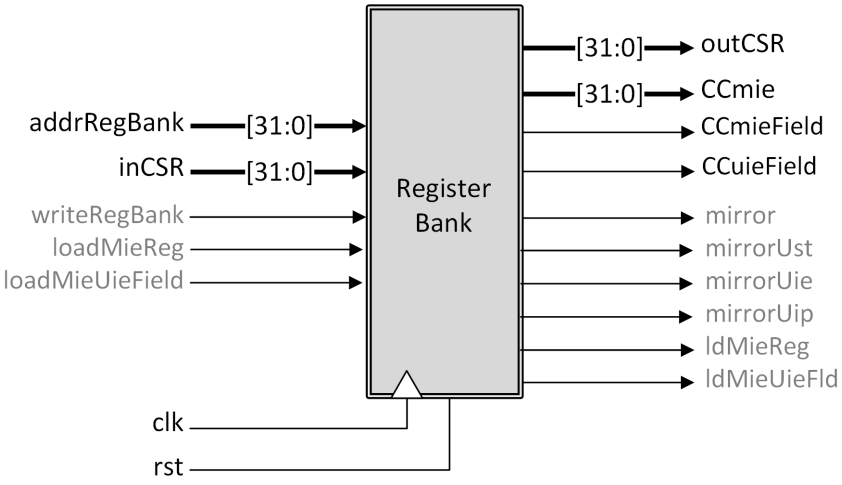

Figure 23: AFTAB Register Bank Unit.

6.12.2. CSR Input Selection Logic (CSRISL)

CSRISL unit prepares input data for CSR registers. This input must be selected between multiple choices that are listed bellow:

- Data from port 1 of register file;

- Immediate value for CSR instructions;
- Data for set or clear a specific CSR according to instruction;
- Appropriate data for CSRs in interrupt/exception process;

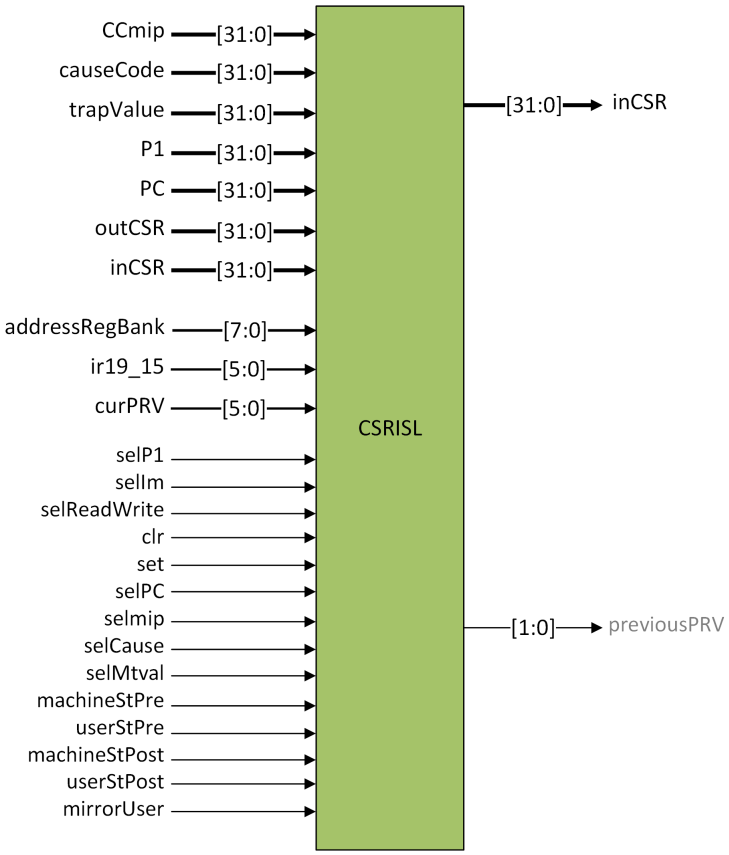

Figure 24: AFTAB CSRISL Unit.

6.12.3. CSR Addressing Decoder and CSR counter

CSR counter is a 3-bit up/down counter that addresses decoder unit for generation a specific CSR address. Counter and decoder units only are used in interrupt/exception process. The output of decoder is input address of register bank.

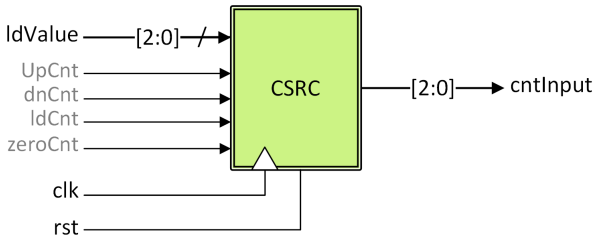

Figure 25: AFTAB CSR counter and Addressing Decoder Unit.

6.12.4. Interrupt Start Address Generation Unit (ISAGU)

Interrupt address generation unit generates PC address for interrupt/exception process. This unit supports interrupt in both direct and vectored mode. Output of this unit is PC value for trap handling.

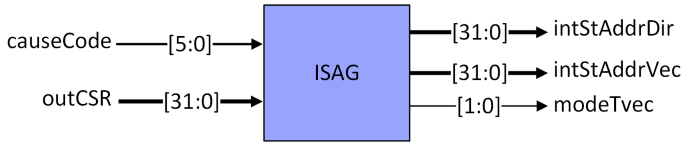

Figure 26: AFTAB ISAGU Unit.

6.12.5. Interrupt Check and Cause Detection Unit (ICCD)

ICCD unit receives interrupt source signals, MIE, and mie field of MSTATUS register then checks `interruptRaise` signal. This unit also generates delegation mode and cause code for interrupt. According to the delegation mode, ICCD unit determines privilege mode for trap handling.

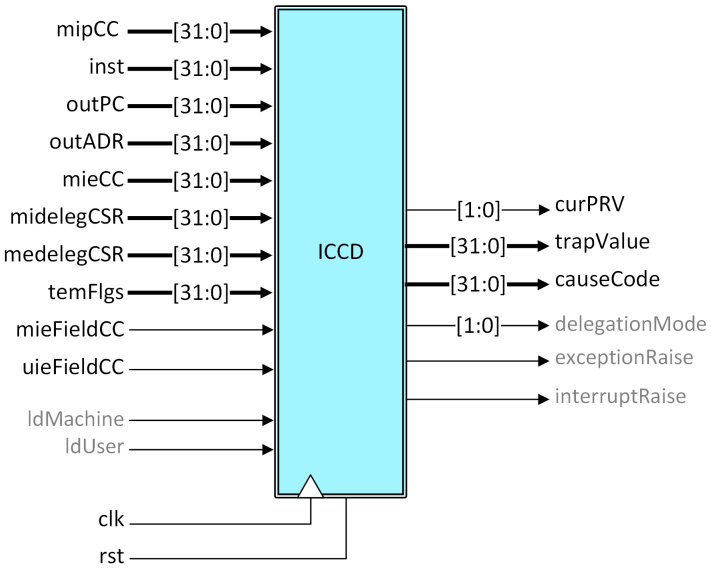

Figure 27: AFTAB ICCD Unit.

7. Controller Unit

The AFTAB Controller Unit is the coordinator all the instructions executed inside the core. Every datapath configuration is performed through the control signals. First the CU fetches the instruction, then the CU identifies the instruction through the fields $opcode$, $funct3$ and $funct7$, and generates the correct control word for the datapath. The control word is defined as the set of all signals required by the datapath to commit the instruction: signals for the registers, multiplexer selectors, and any other type of command for the datapath.

The control word of the AFTAB has 83 control signals in total, with 78 single-bit signals and 5 multi-bit signals.

When IR loads a new instruction, the $opcode$ ($IR[6:0]$) , the $funct3$ ($IR[14:12]$) and the $funct7$ ($IR[31:25]$) fields are read by the CU to compute the control word. The CU is internally defined as an Finite-State Machine (FSM) in which every single state sets the entire control word according to the configurations to be performed each of the datapath units. The Control Unit is a completely synchronous block, so the internal state is updated every clock cycle. Figure 37 shows a schematic of the Controller Unit, with control signals divided per machine state.

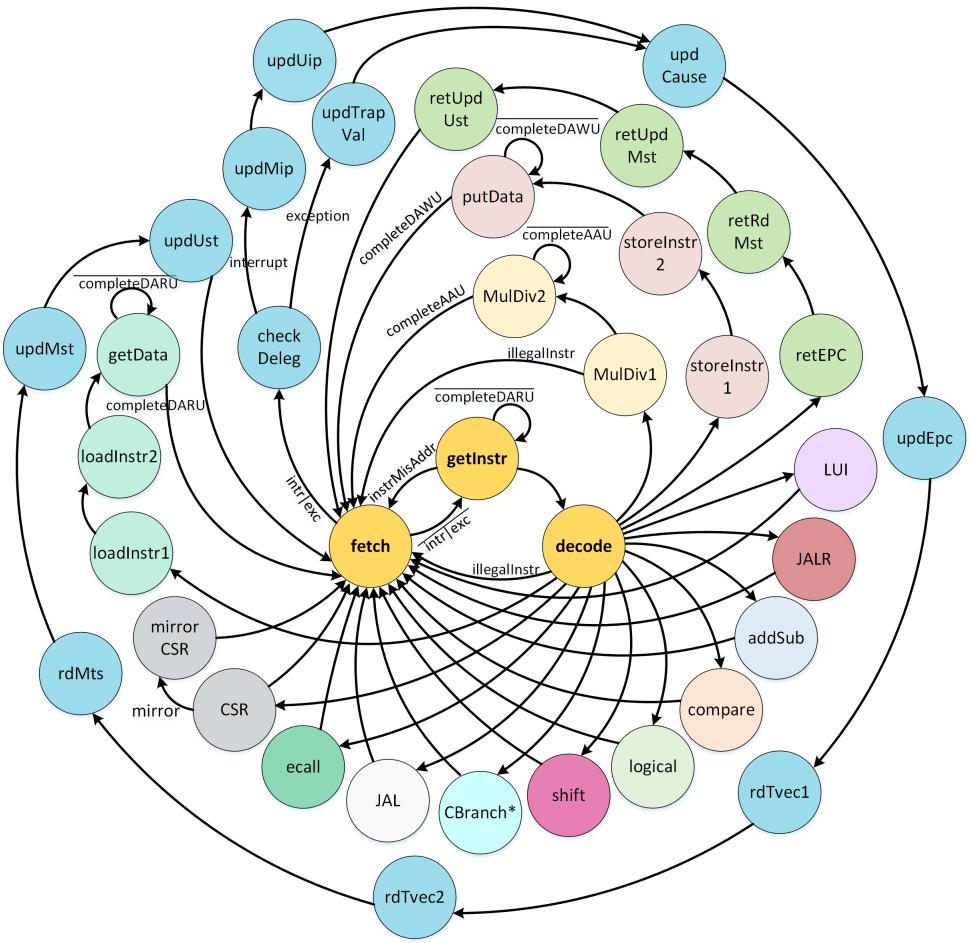

Figure 28: AFTAB Controller Unit.

7.1. States

At each clock cycle, the internal Control Unit FSM can be in one of the states shown in Table 6. All the control signals are set to be asserted in the state in which they operate.

States	Description
Fetch	This is the initial state at reset time. During this state the next value of the program counter is chosen and also the current value to be read form memory. It also starts the reading from IRAM.
GetInstr	In getInstr the instruction is loaded over four clock cycles in byte blocks. When all the blocks are loaded the instruction is loaded into IR
Decode	In decode state the control unit identifies which instruction has to be performed looking at the $OPCODE$, func3 and func7 fields of the IR. According to their vale the next state is chosen.
loadInstr1-2	LoadInstr1 computes the load address and loads it into address register. LoadInstr2 specifies the number of bytes to be read and starts the read process.
getData	In getData state data are read in byte blocks and the total amount of clock cycles depends on the specified amount of bytes. Word requires 4 CCs, halfword requires 2 CCs and byte require 1 CC.
storeInstr1-2	storeInstr1 computes the store address, loads it into address register and the data to be store in the data register. storeInstr1 specifies the number of bytes to be stored and starts the store process.
putData	In putData state data are stored in byte blocks and the total amount of clock cycles depends on the specified amount of bytes. Word requires 4 CCs, halfword requires 2 CCs and byte require 1 CC.
addSub	Datapath is configured to perform addition or subtraction.
compare	Datapath is configured to perform a comparison. Signed-unsigned combinations specified by the control word.
logical	Datapath is configured to perform a logic instruction.
shift	Datapath is configured to execute a shift instruction.
multiplyDivide1-2-3	Datapath is configured to perform Division or Multiplication, respectively require 64 and 33 CCs.
conditionalBranch	Datapath is configured to perform a Branch instruction and select the new PC according the comparison result.
JAL	Datapath is configured to perform Jump and Link Instruction
JALR	Datapath is configured to perform Jump and Link register Instruction
LUI	Datapath is configured to perform Load Upper Immediate.
ecall	The environment exception signal is issued.
CSR	Datapath is configured to read/modify/write the CSRs.
mirrorCSR	Datapath is configured to update the mirror CSR. The mirror CSRs are USTATUS, UIP and UIE.
retEpc	Datapath is configured to read the XEPC CSR.
retRdMst	Datapath is configured to read the MSTATUS CSR.
retUpdMst	Datapath is configured to return the previous status and update the MSTATUS CSR.
retUpdUst	Datapath is configured to update the USTATUS CSR.
checkDeleg	Once an interrupt or exception is raised the controller enters to this state to determine the delegation mode for trap handling. This state reads the delegation CSRs MIDELEG for interrupts and MEDELEG for exceptions.
updTrapVal	If exception is raised, the datapath is configured to update the XTVAL CSR.
updMip	If interrupt is raised, the datapath is configured to update MIP CSR with correct value.
updUip	If interrupt is raised, the datapath is configured to update UIP CSR with correct value.
updCause	Datapath is configured to update XCAUSE CSR with the value of cause code.
updEpc	Datapath is configured to write XEPC CSR with the value of PC.
rdTvec1	Datapath is configured to read XTVEC CSR.
rdTvec2	Datapath is configured to write PC with the value of XTVEC.
rdMst	Datapath is configured to read the content of MSTATUS CSR.
updMst	Datapath is configured to update MSTATUS CSR with the proper value.
updUst	Datapath is configured to update USTATUS CSR with the proper value.

Table 6: Controller Unit states.

8. Control and Status Registers (CSR)

At the current state, AFTAB implements only the Control and Status Registers needed for handling exceptions/interrupts and for supporting user (U) and machine (M) modes. This means that only a subset of the registers listed by the RISC-V privileged specifications are used.

8.1. CSR Address Mapping Convention

The RISC-V ISA encodes CSR addresses on 12-bits, for a total amount of 4096 CSRs. Conventionally, the upper 4 bits are used to express read and write accessibility according to privilege levels. If the two most significant bits (`csr[11:10]`) are set to `00`, `01` or `10`, the register is read-write, while if they are set to `11`, it is read-only. The other two bits (`csr[9:8]`) indicate the lowest privilege level that can access the CSR.

In order to operate on CSRs, instructions in $Zicsr$ extension have been implemented. These allow to perform three different operations: *write* (**`csrw`**), *clear* (**`csrc`**) and *set* (**`csrs`**). Each operation can be performed as register-register or register-immediate instruction.

The **`csrrw`** (Atomic Read/Write CSR) instruction atomically swaps values between CSRs and integer registers. **`csrrw`** reads the old value of the CSR, zero-extends it to 32 bits and then writes it to `rd`. The **`csrrs`** (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to 32 bits, and writes it to integer register `rd`. The initial value in integer register `rs1` is treated as a bit mask specifying bit positions to be set (to 1) in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected. The **`csrrc`** (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends it to 32 bits, and writes it to integer register `rd`. The initial value in integer register `rs1` is treated as a bit mask that specifies bit positions to be cleared (to 0) in the CSR. Other bits in the CSR are unaffected.

The **`csrrwi`**, **`csrrsi`**, and **`csrrci`** variants are similar to **`csrrw`**, **`csrrs`**, and **`csrrc`** respectively, except that they update the CSR using an 32-bit value obtained by zero-extending a 5-bit unsigned immediate (`uimm[4:0]`) field encoded in the rs1 field.

Depending on destination register and `rs1` value, write and read may be performed or not. Figure 29 illustrates in which cases read and write are performed.

Register operand				
Instruction	rd	rs1	read CSR?	write CSR?
CSRRW	x0	-	no	yes
CSRRW	!x0	-	yes	yes
CSRRS/C	-	x0	yes	no
CSRRS/C	-	!x0	yes	yes
Immediate operand				
Instruction	rd	uimm	read CSR?	write CSR?
CSRRWI	x0	-	no	yes
CSRRWI	!x0	-	yes	yes
CSRRS/CI	-	0	yes	no
CSRRS/CI	-	!0	yes	yes

Figure 29: Table showing whether a CSR instruction reads or writes a given CSR.

An illegal instruction exception is raised while accessing CSR without permission. Since the read-only CSRs are not included in this version of implementation, the exception for writing read-only CSRs is ignored.

Table 7 lists the Control and Status registers implemented in AFTAB in machine mode.All these registers also implemented in user mode with corresponding names (USTATUS, UIP, . . .).

CSR Address				Hex	Acc.	Name
11:10	9:8	7:6	5:0			
00	11	00	000000	0x300	MRW	Machine Status (MSTATUS)
00	11	01	000100	0x344	MRW	Machine Interrupt Pending (MIP)
00	11	00	000100	0x304	MRW	Machine Interrupt Enable (MIE)
00	11	00	000101	0x305	MRW	Machine Trap-Vector Base Address (MTVEC)
00	11	01	000001	0x341	MRW	Machine Exception Program Counter (MEPC)
00	11	01	000010	0x342	MRW	Machine Trap Cause (MCAUSE)
00	11	01	000011	0x343	MRW	Machine Trap Value (MTVAL)
00	11	00	000011	0x303	MRW	Machine Interrupt Delegation Register (MIDELEG)
00	11	00	000010	0x302	MRW	Machine Exception Delegation Register (MEDELEG)
00	00	00	000000	0x000	URW	User Status (USTATUS)
00	00	01	000100	0x044	URW	User Interrupt Pending (UIP)
00	00	00	000100	0x004	URW	User Interrupt Enable (UIE)
00	00	00	000101	0x005	URW	User Trap-Vector Base Address (UTVEC)
00	00	01	000001	0x041	URW	User Exception Program Counter (UEPC)
00	00	01	000010	0x042	URW	User Trap Cause (UCAUSE)
00	00	01	000011	0x043	URW	User Trap Value. (UTVAL)

Table 7: Implemented Control and Status Registers.

8.2. Status Registers (MSTATUS and USTATUS)

The MSTATUS register encodes the internal machine status that is given by value associated to different fields. Every field has a fixed position within the register, and is used for a specific purpose. Restricted view of the MSTATUS register appear as the USTATUS register in the user level ISA. MSTATUS encloses the following fields:

- **Machine Previous Privilege mode (MPP)**: encodes in `MSTATUS[12:11]` previous privilege mode. For machine and user mode, it is set to `11` or `00` respectively;
- **Previous Machine Interrupt Enable (MPIE)**: encodes in `MSTATUS[7]` the interrupt enable of the previous privilege. If an exception/interrupt is encountered, MPIE is set to MIE (see below). While exiting exception handling (executing **`mret`**), MIE is set back to MPIE;
- **Machine Interrupt Enable (MIE)**: encodes in `MSTATUS[3]` the interrupt enable of the current priviledge mode. Interrupts are enabled if it is set to 1;

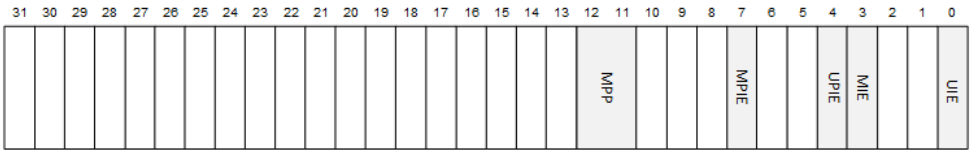

Figure 30: MSTATUS register.

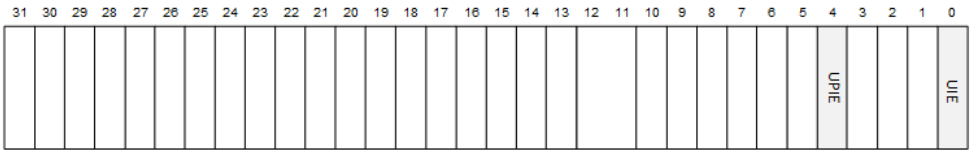

Figure 31: USTATUS register.

The address of MSTATUS is `0x300` and its reset value is `0x00001800`. The address of USTATUS is `0x000`.

8.3. Machine Trap-Vector Base Address (MTVEC)

The MTVEC register is an 32-bit read/write register that holds the base address of the Interrupt Vector Table (IVT) in memory, consisting of a vector base address (BASE) and a vector mode (MODE). The lowest 2 bits indicate MODE setting. When MODE=Direct, all traps into machine mode cause the PC to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the PC to be set to the address in the BASE field, whereas interrupts cause the PC to be set to the address in the BASE field plus 4 times the interrupt cause number (Figure 33). Bits from 31 down to 2 contains the base address.

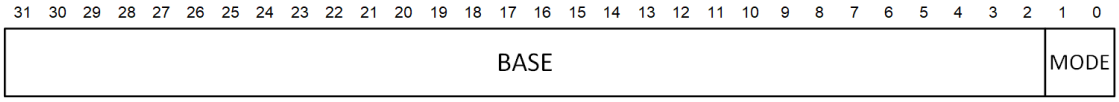

Figure 32: Machine trap-vector base-address register (MTVEC).

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Figure 33: MTVEC register modes.

The MTVEC address is `0x305` and the reset value is `0x00000011`. The UTVEC address is `0x005`.

8.4. Machine Exception PC (MEPC)

The Machine Exception Program Counter (MEPC) is used to store the current program counter whenever an exception or interrupt is encountered. Once the exception handling is concluded and the **`mret`** instruction is executed, MEPC is loaded in the program counter.

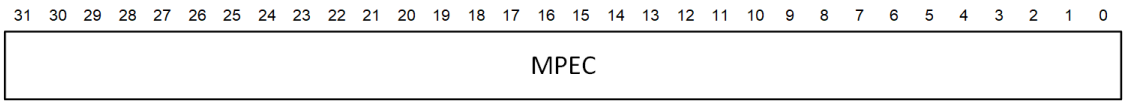

Figure 34: MEPC register.

The register address is `0x341` and the reset value is `0x00000000`. The address of UEPC is `0x041`.

8.5. Machine Cause (MCAUSE)

The Machine Cause Register (MCAUSE) is used to encode the exception/interrupt currently handled into M-mode. Whenever an exception is encountered, the register is set to its *Exception Code*. The register contains two fields:

- **Exception Code**: encodes in `MCAUSE[4:0]` the exception code. This AFTAB implementation supports the following causes (taken from RISC-V specifications):
  - Instruction Address Misaligned code (`0x000`);

  - Illegal Instruction code (`0x002`);
  - Environment call from User and machine mode;
  - External interrupt (from `0x010` to `0x01F`, depending on which of the 16 interrupt lines is asserted);

- **Interrupt**: encodes in `MCAUSE[31]` the exception source. In case it was triggered by an interrupt, it is set to 1, if the trap was caused by an exception this field is set to 0.

Figure 37 reports the complete encoding of interrupt and exception causes taken from RISC-V manual.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Figure 35: Interrupt and exceptions cause encodings.

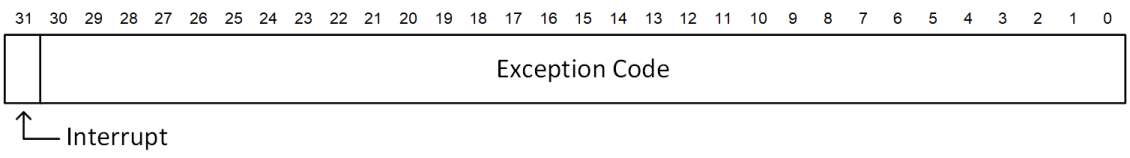

Figure 36: MCAUSE register.

The MCAUSE address is `0x342` and the reset value is `0x00000000`. The UCAUSE address is `0x042`.

8.6. Machine Trap Delegation Registers (MEDELEG and MIDELEG)

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction. To increase performance, implementations can provide individual read/write bits within MEDELEG and MIDELEG to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (MEDELEG) and machine interrupt delegation register (MIDELEG) are 32-bit read/write registers. In AFTAB implementation with two modes, setting a bit in MEDELEG or MIDELEG will delegate the trap in U-mode to the U-mode. MIDELEG holds trap delegation bits for individual interrupts, with the layout of bits matching those in the MIP register. MEDELEG has a bit position allocated for every synchronous exception with the index of the bit position equal to the value returned in the MCAUSE register. The MIDELEG and MEDELEG register addresses are `0x303` and `0x302` respectively.

8.7. Machine Interrupt Registers (MIP and MIE)

The MIP register is an 32-bit read/write register containing information on pending interrupts, while MIE is the corresponding 32-bit read/write register containing interrupt enable bits. There are individual bits for software interrupt, timer interrupt, and external interrupt in MIP are writable through this CSR address. Restricted views of the MIP and MIE registers appear as the UIP/UIE registers in U-mode. If an interrupt is delegated to privilege mode user by setting a bit in the MIDELEG register, it becomes visible in the UIP register and is maskable using the UIE register. Otherwise, the corresponding bits in UIP and UIE appear to be hardwired to zero. The MIP and the MIE addresses are `0x344` and `0x304`.

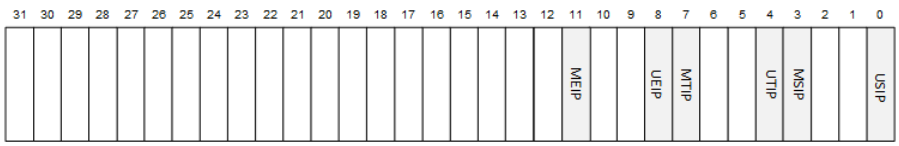

Figure 37: Machine interrupt-pending register.

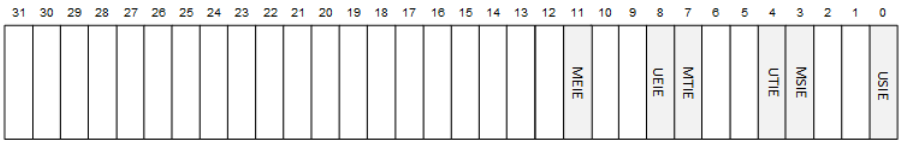

Figure 38: Machine interrupt-enable register.

9. Exceptions, Traps, and Interrupts

AFTAB supports interrupts, exceptions and traps that can be handled in either Machine or User mode based on the delegation CSRs contents. By default, all traps at any privilege level are handled in Machine mode. The presented CSR Units and the interrupt processing states in previous sections are responsible for handling such events.

9.1. Exceptions

In AFTAB, exceptions are used to handle internal faults regarding instruction execution. Illegal instruction exceptions will raise an interrupt to the normal execution flow. They can be raised by:

- Illegal instruction, including illegal instruction opcode, division by zero and CSR illegal instructions (i.e., permission denied or non-existing register).
- instruction address misaligned (i.e., not multiple of 4);

Illegal instruction opcodes are directly detected by the Controller Unit. Division by zero, instruction address misaligned and CSR illegal instruction are detected by AAU, DARU and CSR units, respectively.

9.2. Traps

Traps are exceptions required from user program, e.g., to invoke system/environment permission to perform an action. In RISC-V ISA, the instruction allowing to execute an environment call is named **`ecall`**. When it is executed, the core switches to a higher privilege mode (for example for User to Machine) and the PC jumps to the **`ecall`** handler base address. In order to go back to the previous privilege level, the **`mret`** instruction is executed. (**`ecall`**) trap are directly detected by the Controller Unit.

9.3. Interrupts

Interrupts are events for which normal instruction flow is interrupted by an external request through a physical pin. This AFTAB implementation allows to disable interrupt in a general basis only. The global interrupt enable is done via setting the corresponding bit in the CSR register MSTATUS. Concurrent interrupts priority is given by an ICCD unit.

9.4. Handling

9.4.1. Exception Handling

The entry procedure of handling exceptions, traps and is performed as follows:

- XCAUSE.interrupt = 0; XCAUSE.exception_code = exception cause code
- MSTATUS.XPIE = 1, save previous interrupt enable
- MSTATUS.MPP = current privilege mode (M or U)
- MSTATUS.XIE = 0
- MTVAL = Exception specific information related to the cause
- XEPC = PC, i.e., instruction that caused trap

- PC = handler address
- current privilege mode = delegation mode

The alphabet X can be filled with M or U when machine or user mode, respectively, is selected to handle the raised interrupt.

The exit procedure is when **`mret`**/**`uret`** becomes executed:

- MSTATUS.XIE = MSTATUS.XPIE, restore interrupt enable
- MSTATUS.MPP = 0
- MSTATUS.XPIE = 0
- PC = XEPC
- Privilege mode = MSTATUS.MPP, restore privilege

9.4.2. Interrupt Handling

The entry procedure of handling of interrupts is performed as follows:

- XCAUSE.interrupt = 1; XCAUSE.exception_code = interrupt cause code
- MSTATUS.XPIE = 1, save previous interrupt enable
- MSTATUS.MPP = current privilege mode (M or U)
- MSTATUS.XIE = 0, interrupts are disabled unless the ISR re-writes this
- XEPC = Interrupted PC, save the return address
- PC = handler address
- Privilege mode = delegation mode

The exit procedure using **`mret`**/**`uret`** is the same as for exceptions.

9.5. Interrupt Processing States

This part describes the interrupt processing states. In entry procedure, the controller traverses the following states.

1. When an interrupt is raised, the controller waits until `fetch` state arrives. When an exception is raised, the controller immediately goes to `fetch` state. In `fetch` state, the controller checks whether interruptRaise or exceptionRaise signal is 1;
2. If so, the controller goes to `checkDeleg` and checks the MIDELEG or MEDELEG, respectively, for interrupt handling or exception handling. The controller, then, determines the privilege mode in which interrupt or exception must be handled. The next state is `updTrapVal` when exception is raised, or `updMip` when interrupt is raised;
3. In `updMip`, the interrupt corresponding bit in MIP becomes updated. Since MIP has a mirror CSR, the content of UIP also becomes updated in `updUip`. The next states are common between interrupts and exceptions;
4. The next states are common between interrupts and exceptions. In `updCause`, the cause code is written to XCAUSE;

5. Afterwards, in `updEpc`, the content of PC is written to XEPC;
6. In the next two states `rdTvec1` and `rdTvec2`, PC is filled with the XTVEC and the current privilege also becomes updated;
7. In the next three states `rdMst`, `updMst` and `updUst`, the status of AFTAB is read and MSTATUS and USTATUS become updated;
8. Finally the controller returns to `fetch`.

After the handler instructions execution, the instruction **`mret`**/**`uret`** will be eventually executed, with the following behavior:

1. Executing **`mret`** instruction makes the Controller Unit enter `retEpc` state in which the value of XEPC becomes restored in PC;
2. In the next three states `retRdMst`, `retUpdMst` and `retUpdUst`, the status of AFTAB is restored and MSTATUS and USTATUS become updated:

AFTAB does not support automatic priority enqueuing: modules sending an interrupt request or an exception signal must keep it active until they are served.

9.6. Interrupt Vector Table (IVT)

Whenever an exception or an interrupt is encountered, the Interrupt Vector Table (IVT) base address is read from the MTVEC register, processed in order to add the offset relative to the encountered exception (encoded in MCAUSE register), and written into the program counter. AFTAB supports the Interrupt Vector Table as shown in Figure 8.

Description	Address
Reset Handler	`0x00000000`
Instruction Address Misaligned Handler	`0x00000004`
Illegal Instruction Handler	`0x0000000C`
Environment Call Handler	`0x00000030`
External Interrupt Signals Handlers	`0x000000C4` – `0x00000100`

Table 8: AFTAB interrupt vector table.

Other than the fundamental reset handler, the table only takes into account the supported exceptions and interrupts (see 8.5). The addresses are those that the Control and Status Register Unit produces as next program counter in the event of an interrupt or an exception, according to the formula:

```
pc = (MTVEC[31:2] & "00")+ (MCAUSE[31] & (MCAUSE[4:0] l<< 2))
```

where symbol `l<<` means shift logical left and symbol & means bit concatenation. The addresses contain a **`jal`** jumping to the address of the corresponding service routine.

A. How to simulate AFTAB

This Appendix aims to provide a step-by-step guide for writing, simulating and testing Assembly and C programs on the AFTAB core.

A.1. What is needed

In order to correct set up the AFTAB system, the following software has to be installed:

- A Linux distribution supporting Intel® ModelSim (e.g., Ubuntu >= 16.04 LTS);
- Recent version of Intel® ModelSim (e.g., >= 20.x)[6];
- CMake (>= 3.0)[7]. Versions after 3.1.0 are recommended for supporting Ninja package;
- The GNU RISC-V Toolchain[8] maintained by Berkeley;
- Python (>= 3).

A.2. Repository organization

AFTAB repository provides the following subfolders:

- `/doc`: contains the technical documentation for AFTAB;
- `/rtl`: contains the VHDL source files of AFTAB. All the files are named as `aftab_component_name.vhd`;
- `/sw`: contains all the source programs involved in the simulation environment. Its subfolders hierarchy is the following:
  - `/apps`: contains custom and test programs source files. The folder already presents different applications for testing the implemented RISC-V ISA assembly instructions;
  - `/build`: contains the compilation outputs for every application contained in folder `/apps`. Every application has his own folder under the constant path `./apps/app_name`. Here are archived applications in Executable and Linkable format (`.elf`), S-Record (SERC or S19), hexadecimal format. More, memory initialization and simulation files are stored, with the initial content of the memory and expected results
  - `/ref`: contains boot scripts and linker scripts;
  - `/utils`: contains Python utilities used for generating different executable formats (.S19 or .slm) and automated tests. Automated tests scripts allow to generate simple Assembly test programs and their corresponding expected results (*golden dump*);
- `/tb`: contains all the RTL components that, together with the core, make up the hardware test environment: testbench and memory entities;
- `/vsim`: this folder contains scripts used for compilation of the AFTAB RTL description and simulation of test or custom programs.

[6] https://www.intel.com/content/www/us/en/software-kit/684215/intel-quartus-prime-lite-edition-design-software-version-21-1-for-linux.html? for Linux users. Open the tab "Individual Files" and download the "Questa*-Intel® FPGA Edition" for your operating systems. Please follow the installation guidelines at the bottom of the pages.

[7] https://cmake.org/install/

[8] https://github.com/riscv-collab/riscv-gnu-toolchain. Configure through running `./configure --prefix=/opt/riscv/ --with-arch=rv32im --with-abi=ilp32`, and then just build with `make`.

A.3. How to add custom application

All the applications and scripts to be run inside the AFTAB simulation environment are placed inside the folder `/sw`. Here (and in subfolders) a file named `CMakeLists.txt` is needed for CMake configuration. This file generally contains a set of directives and instructions that aim at describing project's source files and compilation targets. To add an application to the execution environment, a dedicated folder has to be created in the folder `/sw/apps/`. The `app_name` of this folder is conventionally used also for the main source file. Inside it, the file `app_name.c` or `app_name.s` containing the `main()` function can be created, together with other header or source files and the fundamental `CMakeLists.txt`.

CMake configuration files has to be modified as follows:

- `/apps/sw/CMakeLists.txt`: this file performs general CMake configurations, adds external libraries and specifies apps as a sub-directory of the global compilation target. At the very end, the file should be modified adding `app_name` to the set of the `SUB_DIRS`;
- `/apps/sw/apps/CMakeLists.txt`: this file adds to CMake compilation rules all the applications sub-folders though the command `add_subdirectory(app_name)`.
- `/apps/sw/apps/app_name/CMakeLists.txt`: this file has to be created in every application subfolder. It simply adds the application to compilation rules. This operation is performed through the command `add_application(app_name ...)`, followed by the entire list of source files that must be compiled.

At this point, we the CMake configuration can be concluded moving to folder `/build` and running the configuration script through

```
> ./cmake_configure.aftab.gcc.sh
```

If the operation is completed successfully, the Makefiles are created and the application can be compiled from the `/build` directory running the command

```
> make app_name
```

A.4. How to write a program

The source file created under `/sw/apps/app_name` can be both Assembly or C, as the adopted toolchain allows to compile both. If Assembly is chosen, the only constraint is to define a global text area named `main` to indicate the application entry point, as follows:

```
.text
.globl  main
.type main, @function
main:
# Assembly code here
```

In case C language is chosen, the classic `main()` function has to be defined:

```
int main()
{
  // C code here
  return 0;
}
```

A.5. How to compile a custom application

After creating the application folder and performing CMake configuration, an application can be built by moving to `/build` folder and running the following command:

```
> make app_name
```

This command creates executable and linkable format (`.elf`) for the application, and converts it into memory initialization files using a Python engine present in folder `/utils`. The resulting file is named `spi_stim.txt`, which is saved under the path `/sw/apps/app_name/slm_files`.

A.6. How to compile the AFTAB RTL

Prior to any simulation, the AFTAB RTL design must be compiled for setting up the ModelSim environment. This can be done by running the following command from the `/build` folder:

```
> make vcompile
```

This operation runs the script `vcompile_aftab.sh` located in path `/vsim/vcompile/rtl`.

A.7. AFTAB Simulation and Test Environment

This part concerns the AFTAB test environment and how to use it. As Figure 39 shows, the testbench consists in core and memory instantiation, plus a test process tasked to drive test signals.

The testbench internally make connections between core and memory interface, as well as for the signal driven by the testbench process itself. These signals are:

- `clk`: a unique clock with the same frequency used for both core and memory (default: 33 MHz);
- `rst`: asynchronous reset asserted at the beginning of the simulation. During this operation, the instruction memory region is initialized by a process internal to the memory entity, which reads the external `.slm` file produced by the compilation within the application folder in `/sw/build/apps`. Once this is concluded, the processor starts fetching the first instruction;
- `log_en`: this signal is used to perform a data memory dump into an external file (`dram_dump.txt`).

The memory support a 32-bit address, which correspond to 4 GB virtual addressing space, but it is internally designed for a physical dimension of 8 KB (cfr. Figure 6). In order to simplify the resizing, the hardware description for the memory is generic. Memory initialization and data memory dump files are used as parameters inside the VHDL file. The adopted memory protocol is the one described in Section 4.2.

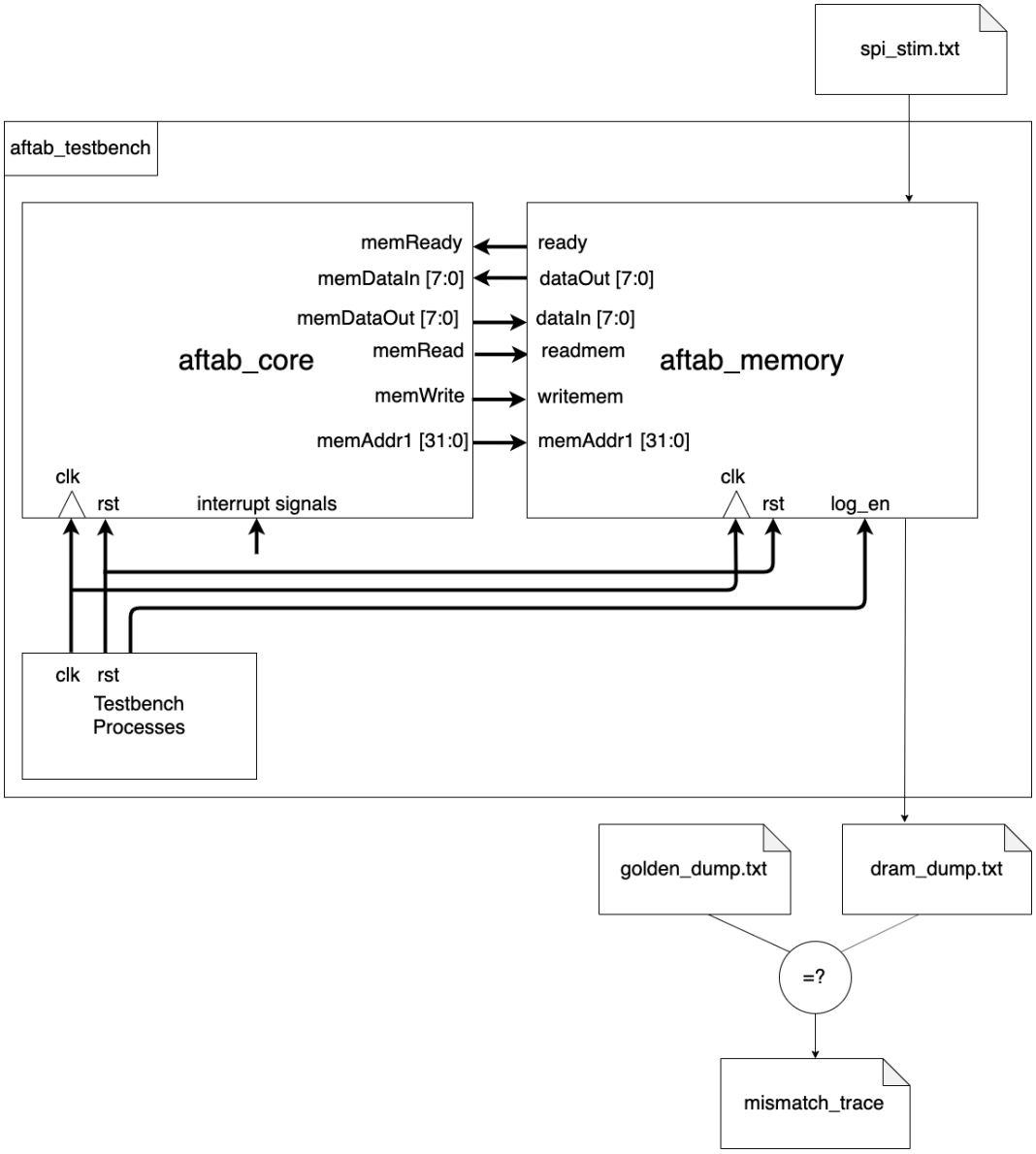

Figure 39: AFTAB simulation environment.

A.8. How to simulate a custom application

The simulation can be perfomed through the following target command:

```
> make app_name.vsim
```

This runs the `run.tcl` script stored `/vsim/tcl_files`. The `.vsim` target runs a graphic instance of ModelSim, including a view with all the AFTAB internal signals, and runs the simulation for a specific simulation time that can be adapted depending on needs.

Please note that this target does not perform the results check at the end. You can perform the automatic check of the application result through running:

```
> make app_name.vsimc
```

that executes ModelSim in command-line mode, and calls the checking script `/vsim/tcl_files/sim_check.py`. Remember that this target only works for application owning the `golden_dump.txt` file inside their `/sw/build` post-compilation folder.

A.9. How to configure simulation time

The default simulation time is assigned to a CMake cache variable that is set inside `/sw/apps/CMakeLists.txt` through the following command:

```
set(SIM_TIME_NS 10000 CACHE STRING "Simulation time.")
```

The variable is named `SIM_TIME_NS` and can be configured from terminal running the following command:

```
> ./cmake_configure.aftab.gcc.sh <sim_time_ns>
```

If no argument is provided while performing the configuration, the default value is set.

A.10. Other available targets

In general, each application supports the following targets:

- `<app_name>`: Compile the application and generate all stimuli for simulation
- `<app_name>.vsim`: Start modelsim in GUI mode
- `<app_name>.vsimc`: Start modelsim in console mode and check results when available
- `<app_name>.elf`: Compile the application and generate the elf file
- `<app_name>.read`: Perform an objdump of the binary and save it as `<app_name>.read` under the corresponding `/sw/build` folder
- `<app_name>.list`: Perform an objdump of the binary with -D option (i.e., with data dumped as well) and save it as `<app_name>.lst`

A.11. Instruction test automation

As shown in Figure 39, the instruction memory has been provided with a signal `log_en` used to dummp the content of the data memory to the file `dram_dump.txt`. After the simulation, this file can be optionally compared to a file containing the expected memory dump, to verify the correct behaviour of the AFTAB design. This has to be named `golden_dump.txt` and has to be placed in the same folder of `dram_dump.txt` (i.e., `/sw/build/app_name/slm_files`).

The check operation is authomatically performed running the command-line simulation command (i.e., `.vsimc` target only). The test result can be analyzed through the terminal that outputs possible mismatching memory addresses, with their expected and actual memory content.

AFTAB simulation environment allows to add test following the same procedure for adding a custom application and placing its expected output in the dedicated folder. The folder `/sw/apps` already contains some applications allowing to test specific RV32IM Assembly instructions. Specific test applications are provided for:

- Arithmetic instructions;
- Branch and jump instructions;
- Control and Status Registers instructions. For this test, a specific boot script has been written (see below);
- Data transfer instructions;
- Logical shift and load immediate instructions.

Inside the folder `/sw/utils`, the scripts for the automatic update of the tests are provided. These are inside the scripts `test_asm_instr_generator.py` and `test_asm_csr_generator.py`, respectively. These are used to generate both test application codes and related golden dumps that are directly updated inside their folders. The scripts can be run through apposite Make targets, namely:

```
> make gen_test_instr
```

and

```
> make gen_test_csr
```

In order to understand their general functioning, here some info are provided:

- `test_asm_instr_generator.py`: this Python script allows to automatically generate application and expected result for the following applications: `test_asm_arithmetic`, `test_asm_branches`, `test_asm_data_transfer` and `test_asm_logical_shift_lui`. Inside the script, some functions are defined and used to provide an essential generalization of the test. Test and golden model files are opened in write mode in order to modify their content. The data used to set registers and generate expected values are contained inside the arrays `REGS` and `IMM` that can be modified if it is needed to perform a specific calculation. Support variables such as `ADDR` and `OFFSET` are used to keep track of the address to be used. Functions used to write the files are:
  - `set_regs()`: writes to the test file instructions that set a certain amount of registers with the values `REGS` array;
  - `test_RR("op")`: writes to test file some register-register instructions using the specified operation;
  - `test_RI("op")`: writes to test file some register-immediate instructions using the specified operation;
  - `test_load_and_store()`: writes to test file some load and store instructions;
  - `test_LUI("op")`: writes to test file **`lui`** or **`auipc`** instructions;
  - `test_J("op")`: writes to test file some jump instructions using the specified operation;

  - `test_B("op")`: writes to test file some branch instructions using the specified operation.

  All the functions write the expected results to the proper `golden_dump.txt`.

- `test_asm_csr_generator.py`: this Python script allows to automatically generate application and expected result for the application `test_asm_csr`. This requires a dedicated script because of the different instruction combinations. As for `test_asm_instr_generator.py`, some functions allow to define a general method to write into files. Similar are also the variables and arrays. Function used to write the files are:

  - `set_regs()`: writes to the test file instructions that set a certain amount of registers with the values REGS array;
  - `test_CSRR("op","r"` or `"i")`: this function allow to write test for a specific CSR instruction on some control and status registers. Since the instructions **`csrrw`**, **`csrrc`** and **`csrrs`** are available both as register-register and register-immediate, `"r"` or `"i"` option has to be specified. The tested registers are: `mstatus`, `mtvec`, `mcause`, `mepc`;
  - `test_CSR("op", "csr", "r", "w"` or `"rw", "r"` or `"i")`: this function allow to write test for a CSR instruction a specific control and status registers. For the instructions **`csrrw`**, **`csrrc`** and **`csrrs`** it has to be specified whether read (`"r"`), write (`"w"`) or read-write (`"rw"`) operation has to be performed, the register on which it is performed and if the operation is `"r"` or `"i"`;
  - `test_illegal()`: this functions writes to the test file some illegal instruction that should rise illegal instruction exception. In particular, these instructions are those accessing CSR registers without having privilege.

  The CSR test is split between its dedicated boot loader (inside `ctr0.boot_MU_test_csr.S`, executed with all privileges) and the test code. The test code file contains only the illegal instructions, while all the remaining ones are written inside the boot loader substituting the pattern `#### TEST ... ####`. Note that removal of this pattern in the original provided file may cause failure of the test generation.

A.12. How to setup privilege modes

AFTAB simulation environment supports two different execution modes: Machine-only (M) and Machine-User (MU). Their main difference concerns the boot process and exception handling.

The first operation performed by both boot processes is to enter the reset handler procedure, in which:

- registers are zeroized;
- stack and BSS start addresses are set;
- BSS region is zeroized;
- `main()` routine is entered.

Regardless of the execution mode chosen (M or MU), this operation is performed in Machine privilege mode, that is defaultly set at reset time by the hardware itself. This privilege level is kept with M boot mode, while it changes to User in case MU is choosen. It is important to highlight that, if the privilege level must switch to User, the Assembly instruction used to perform to enter the `main()` routine must be **`mret`**, while if Machine mode is kept, **`jal`** is sufficient. From now on, if the `main()` runs with Machine privileges, all Control and Status regisets can be accessed,

wherease if we are in User mode, some of them require to increase privilege, for example by using environment call (**`ecall`**).

One more difference between the two boot modes is related to the exception handling. If the core is executing instructions in Machine mode, and an exception is encountered, the privilege mode does not change, while it does if it encountered in User mode. For this reason, to jump back to the normal execution flow, in the first case a simple **`ret`** instruction is used, but in the second case, **`mret`** is required.

Under the folder `/sw/ref`, the following boot scripts are available:

- `ctr0.boot_M.S`: boot script for Machine-only mode;
- `ctr0.boot_MU.S`: boot script for Machine-User mode;
- `ctr0.boot_MU_test_csr.S`: boot script for Machine-User mode, but customized for testing CSR instructions.

To set a specific boot mode, the corrisponding line can be uncommented (and the others commented) in `CMakeLists.txt` in folder `/sw/apps`:

```
set(BOOT_MODE "crt0_boot_M")
#  set(BOOT_MODE "crt0_boot_MU")
#  set(BOOT_MODE "crt0_boot_MU_test_csr")
```