POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Computer Engineering

Master Degree Thesis

Design of a RISC-V Based Microarchitecture for Secure Embedded Systems

Author: Simone Peraccini

Advisor: Paolo Ernesto Prinetto

Co-Advisor: Gianluca Roascio

April, 2022

Alla mia famiglia

Abstract

In computer system design, an Instruction Set Architecture (ISA) is an abstract model defining the standard characteristic for specific hardware implementation. The term was introduced by IBM around 1970 and from that moment most of the popular ISAs are proprietary standards. Even if, the protection of their intellectual property is understandable, their centralization counteracts innovation and artificially affects the cost of microprocessors. In contrast to this monopoly, there are also several open-source projects. One of the most popular is RISC-V and has started in 2010, as part of the Parallel Computing Laboratory at UC Berkeley under the direction of Professor David Patterson. RISC-V ISA is completely free and open-source, and builds on and improves the original Reduced Instruction Set Computer (RISC). The result is clean, simple, and modular; features that make it suitable for low-power embedded systems and high-performance computers alike.

RISC-V was born for study and research purposes, but many companies on the market have started to show interest in implementing their microarchitecture. Considering that, recent open computing standards (like TCP/IP and UNIX) have proved successful allowing free-market competition on technical merit; thus, companies and experts have started to think about the possible benefits of a standard ISA for microprocessors.

The CINI RHES Group Torino, in collaboration with the University of Teheran, aims at developing AFTAB, its 32-bit RISC-V-based platform for secure embedded systems. Its core can be used both for research and academic studies, but also allows to make the work on the platform easier. In particular, the development of such a platform allows experimenting both safety and security techniques, adopting the design paradigm known as *security-by-design.*

The aim is to build up a simulation environment combining the RTL design to a software toolchain, and extending it so that it can resist many famous cyber attacks. The set of instructions supported belongs to the 32-bit base integer ISA variant (RV32I), with multiplication and division extension ("M"). Cross-compilation has been performed setting up the RISC-V-toolchain based on GCC compiler. Custom targets have been also added to automate the RTL compilation running Modelsim commands. The list of instructions belonging to the RV32IM ISA has been completed implementing the missing ones. Moreover, since AFTAB is meant for secure embedded systems, control and status register have been introduced to support user and machine privilege levels. In this regard, the "*Zicsr*″ instruction set has been added as an extension, together with the modules needed to handle interrupts and exceptions.

To verify the correctness of architectural changes, a test automation feature has been added to the toolchain together with some custom test applications in Assembly. To

conclude, AFTAB performances have been compared with open-source microcontroller PULPino. The two cores were compared by simulating on both standard benchmarks of the MiBench embedded suite, published by the University of Michigan.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Every core hardware implementation is characterized by precise design choices [14] [7] that are constrained by the Instruction Set Architecture (ISA). In particular, an ISA describes the registers, functioning of machine-level instructions, their encoding and consequently the functionality that a particular hardware device should implement. Therefore ISAs are a crucial aspect for hardware design and innovation, and since their intellectual property is protected, many companies are asked to pay licenses that hence the high cost of the processors. Moreover, proprietary ISAs make it difficult to research in computer architecture because complete RTL descriptions can not be shared. A different approach is offered by open-source projects that distribute licenses for free and incentivize developers' cooperation.

One of the well-known open-source projects has started in 2010 and is named RISC-V. The project was born in the Parallel Computing Laboratory at UC Berkeley and was directed by Professor David Patterson. RISC-V ISA is completely free and open-source, and builds on and improves the original Reduced Instruction Set Computer (RISC). The result is clean, simple, and modular; features that make it suitable for low-power embedded systems and high-performance computers alike. Study and research are the purposes for which RISC-V was born, but interest in implementing micro-architectures has been shown by a large number of companies. Standardization has often proved successful allowing free-market competition on technical merit; thus, companies and experts have started to think about the possible benefits of a standard ISA for microprocessors.

In the academic domain, several are the fields in which RISC-V can be studied and can enrich student knowledge. Working on an architecture, which is based on an instruction set standard, allows to better understand the general functioning of a system and forces the designer to meet standard constraints that would not be encountered in a non-standard implementation. RISC-V also allows software developers to range from bare-metal implementation up to application level, seeing closely how software is influenced by the hardware. Besides hardware design and software implementation which is the core field in which students can research and experiment, the possibility to work with an open ISA as RISC-V also opens up horizons towards which it is possible to create projects linked to different computer engineering fields. In such a context, it is possible to propose to students the deepening of systems both at the architectural level and also levels concerning greater detail such as the design of specific units or the implementation of specific computation or security algorithms. Once the design of a system has been realized, it will also be possible to dedicate oneself to the study of even more detailed topics, such as synthesis and prototyping on FPGA. From the software point of view, however, one of the interesting projects to which a secure platform based on RISC-V could be joined is an Operating System. In this case, too, students can find fertile ground to study and experiment with concepts such as Real-Time and OS security.

The CINI RHES Group Torino, in collaboration with the University of Teheran, aims at developing AFTAB, its 32-bit RISC-V-based processor core for secure embedded systems. Among the RISC-V advantages listed in the official RISC-V specifications [3], [4], we focus on:

- Completely open ISA that is freely available to research field and industry;
- Modular extendibility starting from small base integer ISA (RV32I) to optional standard extensions. The aim is to offer the possibility to implement systems for both bare-metal programming and 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 are divided into those generally used in all privilege modes (Unprivileged) and the ones requiring a certain level of privilege (Privileged).

The final target is to design a SoC that can be used both for research and academic studies, but also allows to make the work on the platform easier. In particular, the development of such a platform allows experimenting with both safety and security techniques, adopting the design paradigm known as *security-by-design*.

The final target is to design an SoC that can be used both for research and academic studies but also allows to make the work on the platform easier. In particular, the development of such a platform allows experimenting with both safety and security techniques, adopting the design paradigm known as *security-by-design*.

The simulation environment combines the RTL design with a software toolchain and extends it so that it can resist many famous cyber-attacks. Examples of security modules that may be integrated are Physically Unclonable Functions (PUF) and Cryptographically-Secure Random Number Generators (CSRNG), ciphers, key managers, and others.

The set of instructions supported belongs to the 32-bit base integer ISA variant (RV32I), with multiplication and division extension ("M"). The base is limited to an essential arrangement of instructions adequate to give a sensible objective to a compiler, constructing agents, linker, and operating frameworks (with additional advantaged operations), thus giving a helpful ISA and software apparatus chain "skeleton" around which specialized processor ISAs can be designed

To perform the source cross-compilation of C, C++, and Assembly inspiration has been taken from the approach adopted by the PULPino project in which the process is assigned to the RISC-V-toolchain [12]. It is an open-source project maintained by the non-profit organization RISC-V international. Being the toolchain based on CMake it has also been possible to define a custom target allowing to compile the RTL design running Modelsim commands. The initial status of the design implemented a good part of the instructions appertaining to the RV32IM ISA but to complete the necessary ones some of them have been implemented whereas some of those already featured have required corrections. Given that AFTAB is intended for Secure Embedded Systems, it has also been necessary to design and implement a Control and Status Register (CSR) such that the core could operate both in machine and user privilege mode. In particular, this unit has allowed the implementation of both System instructions and all the CSR instructions appertaining to the "*Zicsr*" extension. The integration of this unit has also allowed adding an Interrupt Controller able to handle Exceptions and Interrupts.

Architectural changes verification has been performed through a test environment consisting of a Testbench, containing a core and a memory working in a Harvard Architecture, and a feature allowing to log the memory content to file after simulating an application. For all the implemented instructions Assembly test application and expected result generator have been developed. To conclude it has been chosen to compare AFTAB and PULPino [2] [1] [10] performances to understand how similar our simulation environment was to one of the most complete RISC-V-based microcontroller systems. For this intent, it has been chosen to adapt and simulate Rijndael, Dijkstra, and QuickSort algorithms of MiBench [9] a standard benchmark suite for embedded systems, distributed by the University of Michigan.

The remainder of the document is organized as follows:

- **Chapter 2: Background on RISC-V ISA**: a general description of the RISC-V instruction set architecture focusing on the implemented extensions.
- **Chapter 3: The RISC-V-based AFTAB processor**: a general description of the AFTAB core providing also some information about the different sub-modules.
- **Chapter 4: Extension and Test implementation**: description of the activities performed to build up the current status of the simulation environment.
- **Chapter 5: Performance Evaluation**: describes and evaluates some performance data collected during the work.
- **Chapter 6: Conclusions and Future Work**: describes ideas for future development.

Chapter 2

Background on RISC-V ISA

The original RISC-V aim is for study and research purposes, but in recent years, also industries have announced hardware solutions based on this standard. The aim of this chapter is to describe all the RISC-V's technical elements needed to have a complete understanding of the activities performed during this thesis elaborate. Since it would be too verbose to describe all the extensions and features covered by the official RISC-V ISA documentation this part focuses only on those studied and experimented with by the RHES Group Torino and the University of Teheran. The first description of any RISC-V is implementation is given by the software Execution Environment Interface (EEI), which defines the initial state of the program as well as the number and type of *harts*. An Hart is an described as an entity able to fetch and execute RISC-V Operations.

The EEI also comprises the privilege modes supported by the harts, memory and I/O area accessibility and attributes, the behavior of all lawful instructions on each hart, and the handling of any interrupts or exceptions triggered 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 instructions 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)

RISC-V consists of a base integer ISA and optional extensions allowing the modular extensibility of any implementation. The base integer ISA must be present in any implementation and has been inspired to that of the early RISC processors, except with no branch delay slots but with support for optional variable-length instruction encodings. The base is reduced to an essential set of instructions sufficient to provide a minimal target for compilers, assemblers, linkers, and operating systems (requiring privileged operations), and so it results to be simple starting point ISA to combine to a toolchain structure around which incrementally complex ISAs can be added.

Even if we usually talk about RISC-V ISA, RISC-V groups 4 base ISAs. Each of them is characterized by the width of the integer registers, their number, and the corresponding size of the address space:

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

All extensions use a two's-complement representation for signed integer values. Since our academic purpose does not require a 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. Standard extensions are defined to group correlated instructions such as integer multiply/divide, atomic operations, and single/double-precision floating-point arithmetic. This is particularly convenient for developing general-purpose solutions and avoiding "over-architecting". The extensions defined by RISC-V ISA are the following:

- **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", introduces new floating-point registers, and explands 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 RISC-V ISA Extensions

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.

RV32I has been designed as an essential base for a compiler target and for executing modern operating systems. The internal state of base integer ISA is stored into 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 2.1.

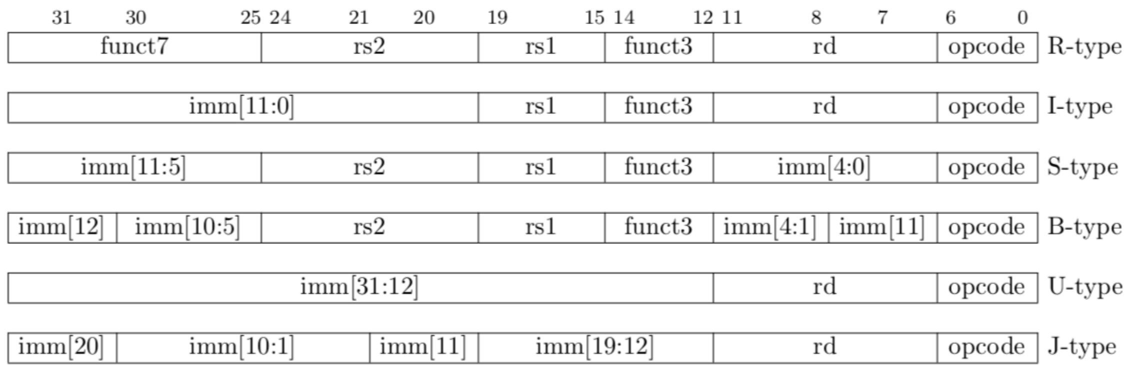
31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 2.1. RISC-V base instruction formats showing immediate variants.

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

There are a further two variants of the instruction formats (B/J) based on the handling of immediate, as shown in Figure 2.1.

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*. No operation allows computing both quotient and reminder.

"*Zicsr*" is the extension defining Control and Status Registers (CSRs) Instructions used to operate on CSRs addresses. CSR instructions use I-Type format and can read and write a single register. The CSR specifier is encoded in the 12-bit 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. Since RISC-V allows to implement subsets of CSRs, AFTAB implements only those needed for handling exceptions/interrupts and for supporting user (U) and machine (M) modes. Table 2.1 lists the Control and Status registers implemented in AFTAB.

The RISC-V ISA counts 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.

CSR Address				Hex	Acc.	Name
11:10	9:8	7:6	5:0			
`00`	`11`	`00`	`000000`	`0x300`	*RW*	Machine Status (MSTATUS)
`00`	`11`	`00`	`000101`	`0x305`	*RW*	Machine Trap-Vector Base Address (MTVEC)
`00`	`11`	`00`	`000001`	`0x341`	*RW*	Machine Exception Program Counter (MEPC)
`00`	`11`	`00`	`000010`	`0x342`	*RW*	Machine Trap Cause (MCAUSE)
`11`	`00`	`00`	`010000`	`0xC10`	*R*	Privilege Level (PRIVLV)

Table 2.1. Implemented Control and Status Registers .

2.3 Control and Status Registers

Machine Status (MSTATUS)

The MSTATUS register encodes the internal machine status that is given by value associated to the following fields:

- **Machine Previous Privilege mode (MPP)**: encodes in `MSTATUS[12:11]` previous privilege mode.
- **Previous Machine Interrupt Enable (MPIE)**: encodes in `MSTATUS[7]` the interrupt enable of the previous privilege.
- **Machine Interrupt Enable (MIE)**: encodes in `MSTATUS[3]` the interrupt enable of the current priviledge mode.

Machine Trap-Vector Base Address (MTVEC)

The Machine Trap-Vector Base Address contains the base address of the Interrupt Vector Table (IVT) in memory. The lowest 2 bits are hardwired to `0x1`, as they are irrelevant in a 4-byte aligned access. This code indicates that the core answers to interrupts in *vectored* mode, i.e., uses register MCAUSE as offset to be added to the base (PC is set to BASE + 4 $\times$ cause). Bits from 31 down to 2 contains the base address.

Machine Exception PC (MEPC)

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

Machine Cause (MCAUSE)

The Machine Cause Register (MCAUSE) is used to encode the exception/interrupt currently handled. 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 that can assume one of the values defined by RISC-V ISA.
- **Interrupt**: encodes in `MCAUSE[31]` the exception source. In case it was triggered by an external interrupt, it is set to 1, otherwise to 0.

Privilege Level (PRIVLV)

The Privilege Level register (PRIVLV) stores the current privilege level the core is executing. The register is read-only and the reset value is `0x000000003`, indicating Machine mode.

A list of all the implemented instructions is shown in Tables 2.2 and 2.3.

Category	Type	Instruction Example	Meaning
Arithmetic	*R*	`add rd, rs1, rs2`	`rd= rs1 + rs2`
	R	`sub rd, rs1, rs2`	`rd = rs1 - rs2`
	I	`addi rd, rs1, imm12`	`rd = rs1 + imm12`
	R	`slt rd, rs1, rs2`	`rd = 1 if rs1 < rs2 else 0` (s)
	R	`sltu rd, rs1, rs2`	`rd = 1 if rs1 < rs2 else 0` (u)
	I	`slti rd, rs1, imm12`	`rd = 1 if rs1 < imm12 else 0` (s)
	I	`sltiu rd, rs1, uimm12`	`rd = 1 if rs1 < uimm12 else 0` (u)
Mul and Div	*R*	`mul rd, rs1, rs2`	`rd = rs1 * rs2` (s) (l)
	R	`mulh rd, rs1, rs2`	`rd = rs1 * rs2` (s) (h)
	R	`mulhu rd, rs1, rs2`	`rd = rs1 * rs2` (u) (h)
	R	`mulhsu rd, rs1, rs2`	`rd = rs1 * rs2` (su) (h)
	R	`div rd, rs1, rs2`	`rd = rs1 / rs2` (s)
	R	`divu rd, rs1, rs2`	`rd = rs1 / rs2` (u)
	R	`rem rd, rs1, rs2`	`rd = rs1 / rs2` (s)
	R	`remu rd, rs1, rs2`	`rd = rs1 % rs2` (u)
Data transfer	*I*	`lw rd, imm12(rs1)`	`rd = DMEM[rs1 + imm12]`
	S	`sw rd, imm12(rs1)`	`DMEM[rs1 + imm12] = rd`
	I	`lh rd, imm12(rs1)`	`rd = sign_16_32(DMEM[rs1 + imm12])`
	I	`lhu rd, imm12(rs1)`	`rd = zero_16_32(DMEM[rs1 + imm12])`
	S	`sh rd, imm12(rs1)`	`DMEM[rs1 + imm12] = rd[15:0]`
	I	`lb rd, imm12(rs1)`	`rd = sign_8_32(DMEM[rs1 + imm12])`
	I	`lbu rd, imm12(rs1)`	`rd = zero_8_32(DMEM[rs1 + imm12])`
	S	`sb rd, imm12(rs1)`	`DMEM[rs1 + imm12] = rd[7:0]`
Logical	*R*	`and rd, rs1, rs2`	`rd = rs1 & rs2`
	R	`or rd, rs1, rs2`	`rd = rs1 | rs2`
	R	`xor rd, rs1, rs2`	`rd = rs1 ^ rs2`
	I	`andi rd, rs1, 20`	`rd = rs1 & 20`
	I	`ori rd, rs1, 20`	`rd = rs1 ! 20`
	I	`xori rd, rs1, 20`	`rd = rs1 ^ 20`
Shift	*R*	`sll rd, rs1, rs2`	`rd = rs1 <<l rs2[4:0]`
	R	`srl rd, rs1, rs2`	`rd = rs1 >>l rs2[4:0]`
	R	`sra rd, rs1, rs2`	`rd = rs1 >>a rs2[4:0]`
	I	`slli rd, rs1, uimm5`	`rd = rs1 <<l uimm5`
	I	`srli rd, rs1, uimm5`	`rd = rs1 >>l uimm5`
	I	`srai rd, rs1, uimm5`	`rd = rs1 >>a uimm5`
Conditional branch	*B*	`beq rd, rs1, imm12`	`if (rd = rs1) goto pc + imm12` (s)
	B	`bne rd, rs1, imm12`	`if (rd != rs1) goto pc + imm12` (s)
	B	`blt rd, rs1, imm12`	`if (rd < rs1) goto pc + imm12` (s)
	B	`bge rd, rs1, imm12`	`if (rd >= rs1) goto pc + imm12` (s)
	B	`bltu rd, rs1, imm12`	`if (rd < rs1) goto pc + imm12` (u)
	B	`bgeu rd, rs1, imm12`	`if (rd >= rs1) goto pc + imm12` (u)

Unconditional branch	*J*	`jal rd, imm12`	`rd = pc + 4`
			`goto pc + imm12`

	J	`jalr rd, imm12(rs1)`	`rd = pc + 4`
			`goto rs1 + imm12`

`lui` and `auipc`	*U*	`lui rd, uimm20`	`rd[31:12] = uimm20`
			`rd[11:0] = 0`

	U	`auipc rd, uimm20`	`rd = pc + (uimm20 <<l 12)`

Table 2.2. RV32IM Instructions supported by AFTAB.

Category	Type	Instruction Example	Meaning

System	*I*	`ecall`	`pc = mtvec + (4 * mcause)`
			`mepc = pc`

	I	`mret`	`pc = mepc`

	I	`csrrw rd, csr, rs1`	`rd = csr`
			`csr = rs1`

	I	`csrrc rd, csr, rs1`	`rd = csr`
			`csr = csr & !rs1`

	I	`csrrs rd, csr, rs1`	`rd = csr`
			`csr = csr | rs1`

	I	`csrrwi rd, csr, uimm5`	`rd = csr`
			`csr = uimm5`

	I	`csrrci rd, csr, uimm5`	`rd = csr`
			`csr = csr & !uimm5`

	I	`csrrsi rd, csr, uimm5`	`rd = csr`
			`csr = csr | uimm5`

Table 2.3. RV32IM and *Zicsr* Extension Instructions supported by AFTAB.

2.4 Privilege Levels

Since AFTAB is moving towards a design able to run an operating system, 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 2.2 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 2.2. RISC-V privilege levels.

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

The highest privilege is held by machine-level and it must necessarily be implemented by any RISC-V hardware microarchitecture. Code run in *machine mode* (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation, therefore it is used to manage secure execution environments on RISC-V. M-mode has unfettered access to the whole machine and can access any register and memory region. *User mode* (U-mode) and *supervisor mode* (S-mode) are intended for conventional application and operating system usage respectively.

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 2.3. Supported combinations of privilege modes.

As shown in Figure 2.3, RISC-V supports three different combinations of privilege modes: machine (M), machine-user (M, U), and machine-supervisor-user (M, S, U). A normal execution flow is performed U-mode but the hart is forced to switch to a different privilege level because of a trap (in AFTAB it may run in M-mode). The first step executed by the hart is to jump and execute a trap handler, at the end of which execution may resume from the original trapped instruction from U-mode. Traps increasing privilege level are termed *vertical traps*, while traps causing no privilege elevation are termed *horizontal traps*.

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.

2.5 Exceptions,Traps and Interrupts

AFTAB supports interrupts, exceptions and traps that can be handled thanks to the presence in the datapath of two units: the CSR Unit and the Interrupt Controller. 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.

2.5.1 Exceptions

Exceptions are used to handle internal faults regarding instruction execution. Illegal instruction exceptions will raise an interrupt to the normal execution flow and cannot be masked. From RISC-V ISA the following cases have been considered:

- illegal instruction opcode;
- division by zero;
- instruction address misaligned (i.e., not multiple of 4);
- CSR illegal instructions (i.e., bad CSR address or permission denied).

2.5.2 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 tho the `ecall` handler base address. In order to go back to the previous privilege level, the `mret` instruction is executed.

2.5.3 Interrupts

Interrupts are events for which normal instruction flow is interrupted by an external request through a physical pin. This 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 Interrupt Controller.

Chapter 3

The RISC-V-based AFTAB processor

The aim of this section is to provide a general description of AFTAB, a custom implementation of the RISC-V architecture by the group of the RHES Group Torino and the University of Teheran.

3.1 AFTAB core

AFTAB is a 32-bit microprocessor with 4 basic macro-stages and a load/store ISA. The architecture is divided adopting a structure that includes *Fetch*, *Decode*, *Execute*, and *Memory.* During instruction execution, the stages 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. All arithmetic operations are performed once both operands are loaded in the register file from the main memory. Intermediate processed data are stored into 32-bit general-purpose registers form `r0` to `r31`. `r0` always assume value zero and can not be written. At the current state, AFTAB is thought to be simulated in Von Neumann architecture. Therefore, the basic environment used includes the core and one memory only, following a little-endian convention.

As per good practice in hardware design, the core consists of two main sub-components, Datapath and Control Unit. These sub-modules can generally be described as follows:

- **AFTAB Control Unit**: the AFTAB Control Unit is the coordinator of all the instructions executed inside the core, which means that its control signals are used to drive any possible datapath configuration. The set of all the control signals is named control world and it is set by the control unit during the decode stage.
- **AFTAB Datapath**: the AFTAB Datapath contains the components of *Fetch*, *Decode*, *Execute*, and *Memory.* The Fetch aim is to select the new instruction that can be chosen between value read from instruction memory, trap vector, exception program counter, and jump result. Once the instruction is selected, data are read from memory thanks to a *Data Adjustment Read Unit* (DARU) and written into the

instruction register. In the decode stage, the instruction fields are decoded and given used as reading addresses for the register file or immediate. Register file outputs and immediate are used in the execute stage to fed the computational units that will be briefly described in the following sections. Finally, the memory stage allows both to read and write data thanks to a DARU and a *Data Adjustment Write Unit* (DAWU).

Figure 3.1 shows the high-level representation of the datapath stages.

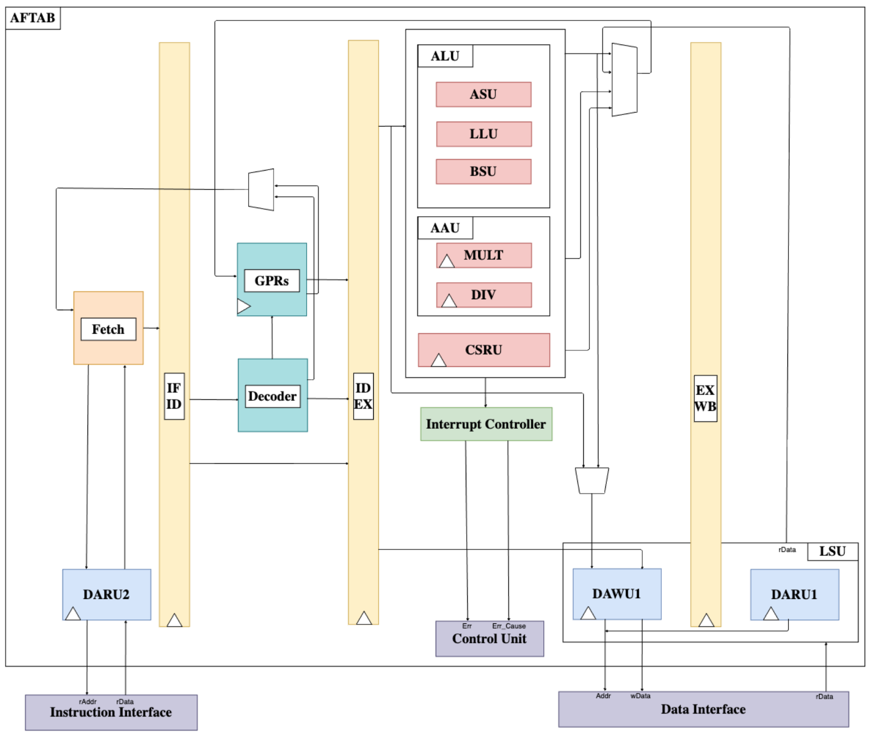

Figure 3.1. High-level representation of the AFTAB datapath.

The Control and Status Unit (CSRU) and Interrupt Controller components will be described in the third Chapter, since they have been implemented during the performed activities.

3.1.1 Control Unit

The AFTAB Control Unit is the coordinator of all the instructions executed inside the core. Every datapath configuration is performed through the control signals. After the instruction is fetched, the CU identifies the instruction through the fields opcode, $func3$, and $func7$, 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: clock enables, signals for the registers, multiplexer selectors, and any other type of command for the datapath. The control word of the AFTAB has 37 control signals in total, with 27 single-bit signals and 10 multi-bit signals. The amount for the multi-bit signals is 66 and the total control bit count is 93. When IR loads a new instruction, the opcode (IR[6 : 0]), the $func3$ (IR[14: 12]), and the $func7$ (IR[31: 25]) fields are read by the CU to compute the control word. The CU is internally defined as a Finite-State Machine (FSM) in which every single state sets the entire control word according to the configurations to be performed in each of the datapath stages. The Control Unit is a completely synchronous block, so the internal state is updated every clock cycle. Figure 31 shows a schematic of the Control Unit, with control signals divided per machine stage and unit. Figure A.2 in the Appendix shows a schematic of the Control Unit, with control signals divided per machine stage and unit.

3.1.2 Datapath and Instruction Execution Flow

This Section aims at providing details on a basic behavior of the AFTAB microprocessor, going through each stage 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 Program Counter(PC), which is forwarded to the instruction memory port;
- 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 stages the core may enter during the machine cycle are detailed.

Fetch

It aims 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 instruction read for instruction memory. This tasks are performed in two different states named `Fetch` and and `GetInstr`:

- `Fetch`: the fetch stage 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`. The PC choice can be selected between five different inputs: current PC incremented by 4, register file port 1 output, branch or jump result, Machine Trap Vector Base Address (MTVEC), and Machine Exception Program Counter (MEPC)
- `GetInstr`: since only 32-bit instructions are implemented, this stage always requires 4 clock cycles to complete, each one of them reading an instruction byte. Starting from the PC address, the DARU2 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 IR.

Decode

The decode stage always lasts one clock cycle and aims at extracting the instruction fields from the Instruction Register. The extracted fields are the following (also shown in Figure 2.1), 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 Control 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 Control Unit. At the end of the clock cycle, the Register File (RF) synchronously updates the two read ports `p1` and `p2`, while the immediate is computed by the Immediate Selection and Sign Extension Unit (ISSEU).

Execute

During the execution, the operands (`p1` and `p2`) and immediate are ready to be used by all the units inside the execute stage. The actual values provided to the units are selected by two bypass multiplexers that select 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
- Extended unsigned immediate used in CSR Instructions.

A second multiplexer chooses between the output of the read port 2 (`p2`) and the immediate computed by the ISSEU. The execute stage lasts a variable number of clock cycles that depend on the operation to perform.

In the following all the datapath's computational units are listed:

- **Comparator**: takes one clock cycle to compute the result of three comparisons: *greater-than* (`gt`), *lower-than* (`lt`) and *equal-to* (`eq`).;
- **Adder/Subtractor Unit (ASU)**: takes one clock cycle to compute a 32-bit result that can be selected between Adder and Subtractor;
- **Logical Logic Unit (LLU)**: takes one clock cycle to compute the result of three possible logic operations: *XOR*, *OR* and *AND*;

- **Barrel Shifter Unit (BSU)**:takes one clock cycle to perform on of the three possible operations: *Shift Left Logic* (SLL), *Shift Right Logic* (SRL) and *Shift Right Arithmetic* (SRA);
- **Attached Arithmetic Unit (AAU)**: allows to perform both multiplication and division on 32 bit. Multiplication is implemented exploiting Booth Algorithm and takes 64 clock cycle to compute signed or unsigned product. Division exploits Restoring Algorithm and takes 64 clock cycle to compute quotient or reminder(both signed and unsigned can be executed

Control Status Register Unit and **Interrupt Controller** will be described in the following section since they have been implemented during the thesis work.

Memory

This stage is performed only in case a load or store instruction has to be executed. In this case, the CU respectively starts DARU1 and DAWU1 and enters the states `getData` (for load) or `putData` (for store), and keeps each state for a number of clock cycles that depends on the number of bytes exchanged from/to the memory. In particular, the clock cycles required for instructions `sb` and `lb` are one, while two clock cycles are taken for `sh` and `lh` and four for `sw` and `lw`.

Write-Back into register file is performed at the end of the Execute stage. Since there is no ALU output register, the operations result and data read from memory are directly written to the Register File.

3.2 Programmers Model

This Section is intended to show the AFTAB software model available to programmer for programs execution. AFTAB implements RISC-V Base Integer Instruction Set (RV32I), the Integer Multiplication and Division Extension (RV32M) and ı*Zicsr*" Control and Status Register Instructions.Moreover, it 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 **??** explains how to setup privilege execution.

When writing a program for AFTAB, a programmer may choose to provide both an Assembly source file (.s) Assembly or 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.

General-purpose registers are labeled from `x0` to `x31`. RISC-V programming convention assigns standardized names to most of them specified in the application binary interface (ABI. They are listed in the following:

- **Saved registers:** (`s0` to `s11`) are used to store data that must be preserved across function calls;
- **Argument registers:** (`a0` to `a7`), are used to pass arguments;
- **Temporary registers** (`t0` to `t6`) are used for function for computation;

- **Specialized registers** Stack Pointer (`sp`), the Global Pointer (`gp`), the Thread Pointer (`tp`) and Link Register (`ra`).

All naming convention is listed in Table 3.1.

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
```

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 3.1. RV32I general register map.

Chapter 4

Extension and Test implementation

Chapters 2 and 3 have presents the background knowledge needed for working with RISC-V ISA on AFTAB. The one we are introducing describes the performed activities that have advanced the status of the RISC-V simulation platform born from the collaboration between CINI and University of Teheran. Inspiration for the elaborate has been take form one of the better organized and featured projects published on GitHub, PULPino [10].

Our intent is to implement a simulation environment similar to PULPino. For this reason, it has been decided to start integrating the RISC-V toolchain adopting its same structure. This CMake tool combined to the RTL description makes much simpler the compilation of both HDL description and C applications, but also allows to easily run simulation on Modelsim. One of the main advantages is that all the commands run inside the RISC-V platform share the same "make"-like format and details are hidden inside simulation and compilation scripts. Another interesting feature is the possibility of adding custom targets.

This setup has allowed move our focus from compilation and simulation details to the ones related to the RTL design description. The first activity performed on the RTL is to check implemented instructions' correctness and implement the one missing for completing RV32IM and "*Zicsr*″ extension. Where needed, some instructions has also been corrected.

The next phase of the thesis has been dedicated to the Test environment setup. To verify corrected and implemented instruction, functional tests of all the instructions have been performed. Since it was necessary to check single machine instruction's result, scripts containing multiple call for every one of them has been developed. For the testbench, a Von Neumann architecture has been set up connecting the core to a memory, whose content has been loaded and dumped thanks to dedicated files. Verification of the instructions gave us the certainty of their functioning and it has allowed us to start implementing C and C++ applications. In order to collect data regarding AFTAB performances, it has been thought that could be interesting to run some benchmark applications. Since the implementation of many of the standard benchmarks uses standard I/O with `fscanf()` and `frpintf()`, some changes have been made. Basically, due to the absence of peripherals for handling files and UARTs, all printed output have been removed and files have been replaced by static variables. The source from which the benchmarks have been chosen is MiBench [9], a standard suite for embedded systems distributed by the University of Michigan. In particular, we have chose three common algorithm: Quick Sort, Rijndael, and Dijkstra. Finally, we have analyzed the performances obtained running benchmarks on both AFTAB and PULPino. The resulting performances has allowed to better understand the status AFTAB with respect to a well-known platform as PULPino.

4.1 ISA Extension and Correction

As anticipated in the previous Section, the base structure of AFTAB has been provided by the University of Teheran who also shared some related diagrams. Since the test program have been developed during the thesis work, at first we were unaware of which instructions worked correctly and if any of them needed corrections. Therefore, we have performed an analysis on all the modules and afterward, changes have been made to some of them. In the following, a brief description of the modules and how they have been edited:

- **Core**: this unit describes the Core which internally connects Datapath and Controller. It has been edited adding one memory port and 16 external interrupt lines.
- **Controller**: this unit describes the Control Unit which is internally implemented as a Finite State Machine. The performed changes are the following:
  - Interface refactor as shown in Figure A.2 and described in Tables A.2 and A.3;
  - Extension of the decode stage including the opcodes of the implemented instruction. In particular the instruction implemented are listed in Table 2.3;
  - Added CU states for implemented instructions, in particular: system, control and status register operations, exception handling, and illegal instructions. A brief description is provided in Table A.4;
  - VHDL style refactor.
- **Attached Arithmetic Unit (AAU)**: the AAU, internally instantiates two components for Signed-Unsigned Division (using restoring Algorithm) and Booth Multiplication. Neither algorithm has been relevantly modified. The only change that has been performed concerns the Division, in which at the first operation cycle we added check for zero divisor. In chase this condition is verified, a signal connected to AAU output port is set to 1;
- **Data Adjustment Read Unit (DARU)**: interface signals related to instructions and data errors have been added. The unit has been extended inserting a check for misaligned memory addresses. Since misaligned read occurs whenever the address is not a multiple of the number of bytes exchanged, the read amount of data has been considered. A dedicated signal has been added to the port;
- **Data Adjustment Write Unit (DAWU)**: DAWU interface listed signals for instructions and data errors, but did not implement any of them. The unit has been extended inserting a check for misaligned memory address, as for DARU;

- **Comparator**: this unit describes a generic Comparator able to perform three comparisons: *greater-than* (`gt`), *lower-than* (`lt`) and *equal-to* (`eq`).;
- **Control and Status register Unit (CSRU)**: this unit describes a Control and Status Register Unit managing operations on CSRs. The unit has been added to the initial state and its development has been inspired by an open-source structure available online. Further details will be provided in the following Sections.
- **Datapath**: this unit describes the Datapath on which changes have been performed:
  - Renaming of both interface and internal signals: some of the initial ones did not allow an easy understanding of the unit.ì;
  - All the multiplexers were implemented using one-hot encoding for every input selection. This has been simplified by changing the selecting signals to a logarithmic quantity with respect to the inputs;
  - Instantiation of the CSRU;
  - Instantiation of the Interrupt Controller;
  - Interrupt and exception signals;
  - Added signals for handling Exceptions and CSR Instructions to the interface;
  - VHDL style refactor;
  - Duplication of the DARU to have a dedicate DARU for instructions and another for data.
- **Interrupt Controller**: the unit describes the Interrupt Controller that collects all exception and interrupt signals. The Control Unit is notified according to the exceptions priority. Further details will be provided in the following Sections.

In order to automate the process of solving VHDL style issues, we have exploited an open-source tool named *VHDL-style-guide* [15]. Its features allow to report and fix common errors in VHDL coding standard running a simple command, thus avoiding having to solve them by hand.

4.2 Privileged Modes and Exceptions design

Privilege Mode implementation has been necessary when implementing System Instructions belonging to both RV32I and "*Zicsr*" extension. This is because they require a control and status register bank that can be accessed respecting *privilege levels*. All the operations involving CSRs are handled by the Control and Status Register Unit (CSRU), which is a synchronous component hosting the implemented ones. The unit design was inspired by an available open-source project found on Github [11]. This unit, together with the Interrupt Controller, also allows handling Exceptions and Interrupts.

4.2.1 Control and Status Register Unit (CSRU)

The unit works as a simple memory bank allowing to update and read every single register 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. CSRU also allows to handle exceptions and interrupts, requiring an entry and an exit phase that is started through the signals `int_entry` and `int_exit`. Figure 4.1 shows the CSRU interfacing ports.

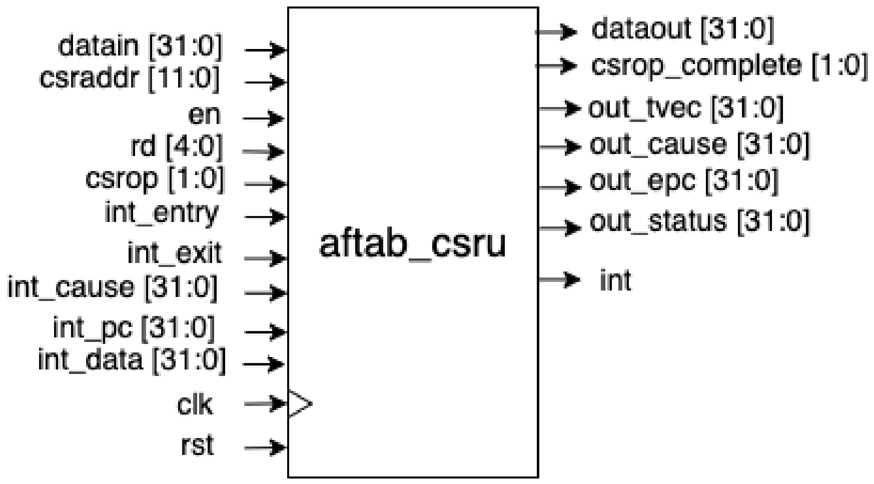

Figure 4.1. Control and Status Register Unit.

- `clk` and `rst`: all CSR instruction and exception handling are performed synchronously on clock rising edge while the active high reset is asynchronous;
- `rd`: it is a 32-bit input used for indicating the CSR operation destination register;
- `datain`: it is a 32-bit input used for data to be written in CSR registers;
- `csrop`: 2-bit input encoding the CSR operation: `10` for write, `01` for set and `11` for clear;
- `csraddr`: 12-bit input used for addressing CSRs;
- `int_cause`: 32-bit input signal used for encoding exception/interrupt cause;
- `dataout`: 32-bit output signal used for data to be written into register file;
- `csrop_complete`: 2-bit input encoding the CSR operation status. `00` for not complete, `01` for complete and not write, and `11` for complete and write;
- `int`: output signal to the Interrupt Vector Table (IVT) set whenever an interrupt is risen (i.e., invalid CSR Instruction);

- `int_entry`: input signal from the Control Unit set whenever an exception/interrupt has to be risen;
- `int_exit`: input signal set to when exception/interrupt is concluded;
- `out_status`: 32-bit output of the register MSTATUS;
- `out_cause` 32-bit output of the register MCAUSE;
- `out_tvec`: 32-bit output necessary for the computation of the destination PC;
- `out_epc`: 32-bit output of the register MEPC.

CSRU Finite State Machine

The Control and Status Register Unit operates over three different states: **READ_OR_IDLE**, **MODIFY** and **WRITE**. The transitions between the different states are shown in the FSM in Figure 4.2.

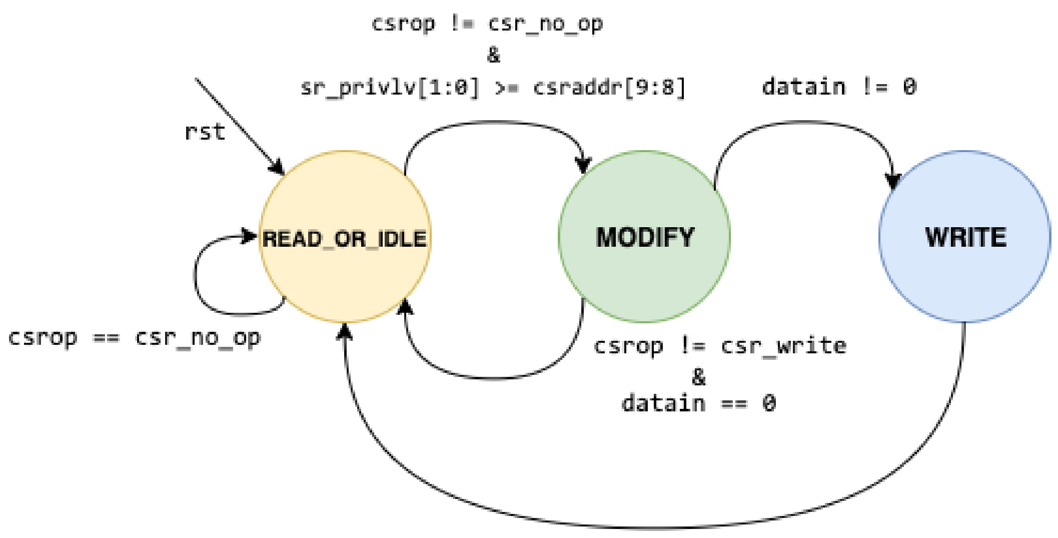

Figure 4.2. Control and Status Register Unit Finite State Machine.

In CSRU, Mealy state machine operations may last both two (no write) or three (write) clock cycles depending on the values of the signals `datain` and `csrop`. The behavior of the single states is the following:

- **READ_OR_IDLE**: the CSR unit keeps this state until the input `csrop` requires to execute an operation (`csrop`). When an operation is requested and the privilege level is over the minimum one (i.e., `csrop >= csraddr[9:8]`), the CSR bank is read at the requested address, while if the privilege level is lower than the minimum, the signal `int` is set for rising an illegal instruction exception. The same exception is risen in case the requested address is not among the implemented ones. The next state is set to **MODIFY** if an operation is started, while it remains at **READ_OR_IDLE** if an exception is risen or it is not asked to perform any operation;

- **MODIFY**: in such a state, the next value to be written at `csraddr` is computed. In case the operation is a write, it is simply set to `datain`, while if the operation is set or clear, the computed value is computed performing OR (between `datain` and the current value) and AND (between the negated `datain` value and the current value). If the register is read-only (i.e., `csraddr[11:10] = "11"`) and a write operation has to be performed, an illegal instruction exception is risen. The next state is set to **WRITE** if a new value has to be assigned to a CSR, while it switches back to **READ_OR_IDLE** if it is only read or an exception is risen;
- **WRITE**: in this state, the next control and status register are set and the operation is completed. The next state is unconditionally set to **READ_OR_IDLE**.

Interrupt/Exception Entry and Exit

Interrupt/exception entry and exit process follow these steps:

- `int_entry`: whenever the signal `int_entry` is set, the core enters Machine mode setting the privilege level to the maximum value (3). In addition, **MPIE** (corresponding to bit `mstatus[7]`) is set to the current **MIE** (`mstatus[3]`), **MIE** is set to zero and **MPP** is set to the current privilege mode, **MCAUSE** and **MEPC** are set and the `out_tvec` output value is computed;
- `int_exit`: whenever the signal `int_exit` is set, the core enters the privilege mode stored in the bits **MPP**. In addition **MIE** is set to **MPIE**, **MPIE** is set to 1 and **MPP** is set to `"00"`.

Both the procedures are concluded in 2 clock cycles.

4.2.2 Interrupt Controller

The Interrupt Controller is a unit aiming to collect all the exceptions/events and provide a unique notification to the Control Unit to preempt its normal activity. Figure 4.3 shows its interfacing ports.

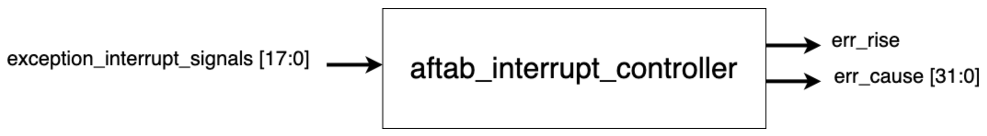
exception_interrupt_signals [17:0]
aftab_interrupt_controller
err_rise
err_cause [31:0]

Figure 4.3. AFTAB Interrupt Controller.

Whenever an exception/interrupt is encountered, one of the input lines is set to 1. If this happens, the Interrupt Controller outputs the cause on the 32-bit signal `err_cause` (following the MCAUSE encoding) and sets the `err_rise` signals, telling the Control Unit that an exception has to be handled.

If more events occur concurrently, the priority is assigned statically, following the lowest-to-highest bit ordering of the input signal. Therefore, the user of this processor can decide to connect his/her peripheral interrupt lines depending on which is the desired priority.

For exceptions, the priority is fixed by the specification, and all internal exceptions must precede external interrupts. In conclusion, AFTAB features such a priority list:

1. Illegal instruction;
2. Instruction address misaligned;
3. Environment call;
4. External interrupt;

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

4.2.3 Bootloader

Since the simulation environment is thought for secure embedded systems, AFTAB 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 set to zero;
- stack and BSS start addresses are set;
- BSS region is initialized;
- `main()` routine is entered.

Regardless of the execution mode chosen (M or MU), this operation is performed in Machine privilege mode, which is default 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 chosen. 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 registers can be accessed, whereas 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 is 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.

The following Boot Loaders are available:

- Boot Loader for Machine-only mode;
- Boot Loader for Machine-User mode;
- Boot Loader for Machine-User mode customized for testing CSR instructions.

4.3 Testing

This part concerns the AFTAB test environment and it is used. As Figure 4.4 shows, the testbench consists of core and memory instantiation, plus a test process tasked to drive test signals.

4.3.1 Functional Test of the Environment

The testbench internally makes connections between the 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;
- `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 file produced by the compilation within the build folder. 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 at the end of the execution, for automatic results checks.

The memory supports a 32-bit address, which corresponds to 4 GB virtual addressing space, but it is internally designed for a physical dimension of 8 KB (cfr. Figure 4.5). 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.

As shown in Figure 4.4, the instruction memory has been provided with a signal `log_en` used to dump the content of the data memory into an external file. After the simulation, this file can be optionally compared to a file containing the expected memory dump, to verify the correct behavior of the AFTAB design.

The check operation is authomatically performed running a simulation command. The test result can be analyzed through the terminal that outputs possible mismatching memory addresses, with their expected and actual memory content.

4.3.2 Test Applications

As it happens for PULP platform, AFTAB simulation environment allows to add custom applications and places its expected output in the dedicated folder. Applications folder already contains some applications allowing to test specific RV32IM and "*Zicsr*" 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;

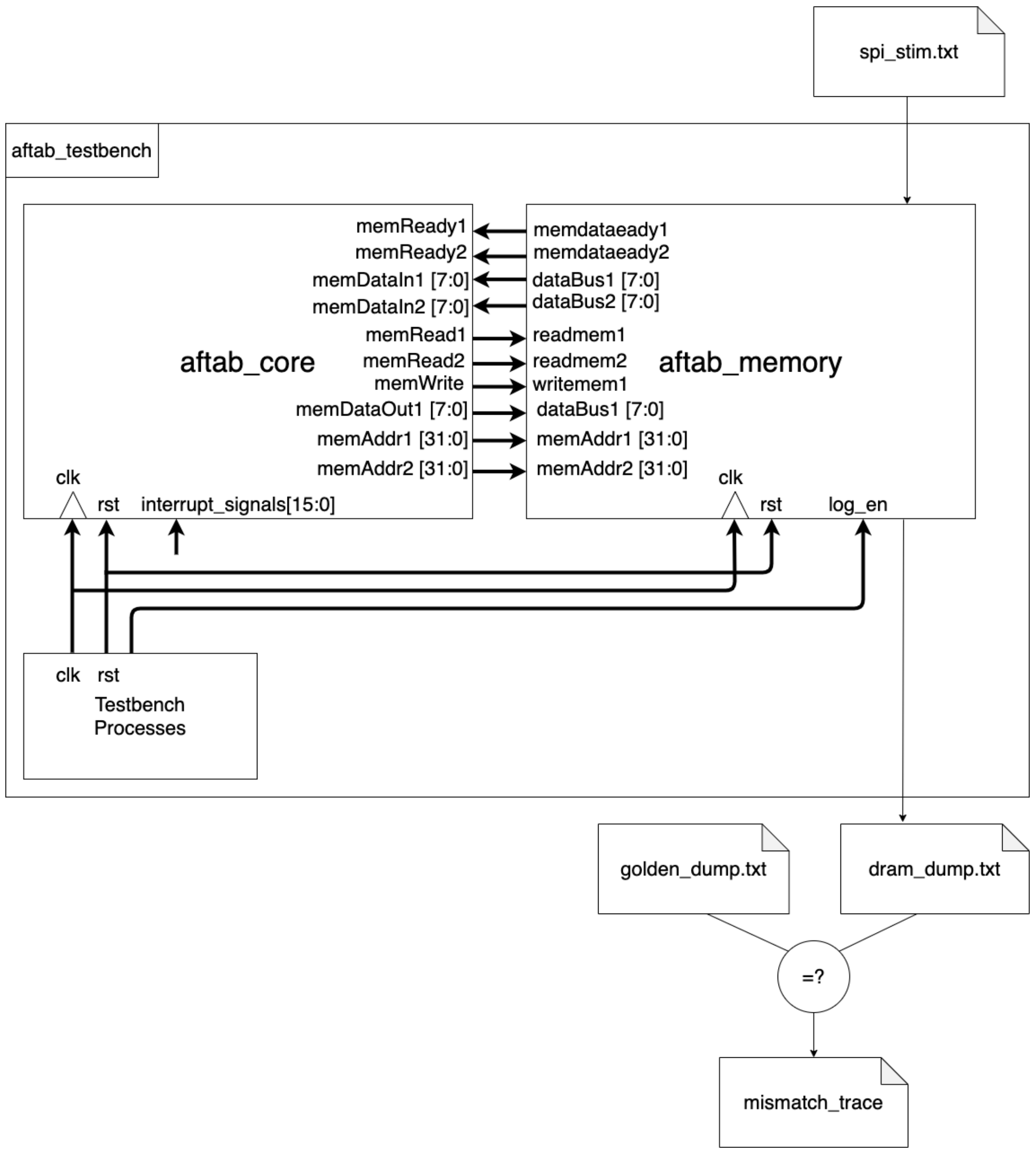

Figure 4.4. AFTAB simulation environment.

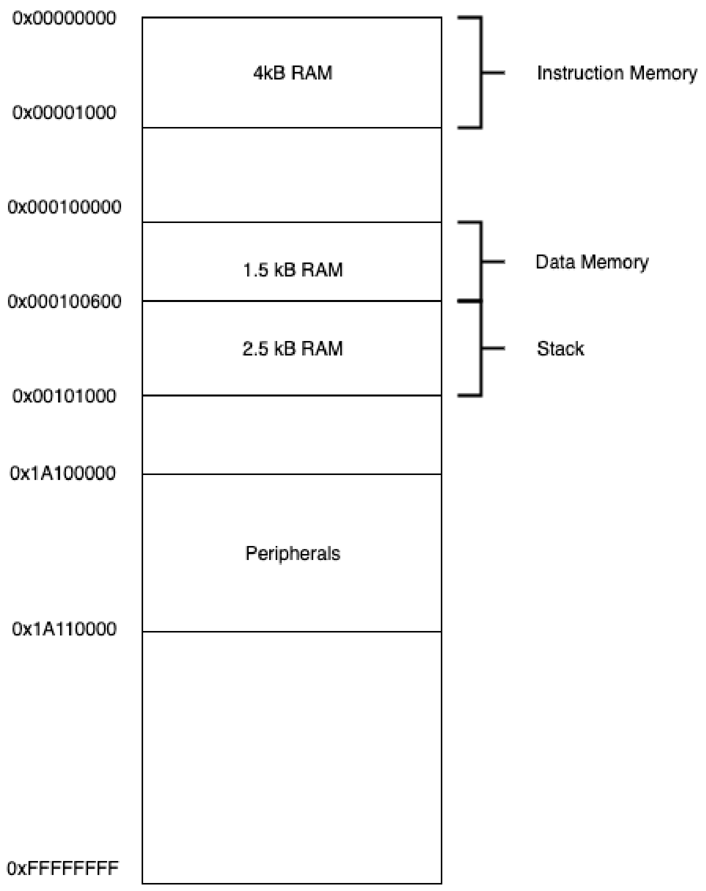

Figure 4.5. AFTAB current memory map.

- Logical shift and load immediate instructions.

In order to automate test update some scripts have been developed. These have been used to generate both test application codes and related golden dumps that are directly updated inside their folders. In order to understand their general functioning, here some info is provided:

- **RV32IM test generator**: this Python script allows to automatically generate application and expected result for the following group of instructions: Arithmetic, Branches, Jumps Data Transfers, Logical, Shift and Upper Immediate operations. Inside the script, some functions are defined and used to provide an essential generalization of the test. The data used to set registers and generate expected values are contained inside static arrays (named `REGS` and `IMM`) that can be modified if it is needed to perform a specific calculation. Support variables (named `ADDR` and `OFFSET`) are used to keep track of the address to be used. Functions used to write the files aim at:

  - **Set Registers**: writes to the test file instructions that set a certain amount of registers with the values `REGS` array;
  - **Register-Register instruction**: writes to test file some register-register instructions taking as input a specific operation;
  - **Register-Immediate instruction**: writes to test file some register-immediate instructions taking as input a specific operation;
  - **Load and Store instruction**: writes to test file some load and store instructions;
  - **Load Upper Immediate Instruction**: writes to test file `lui` or `auipc` instructions;
  - **Jump instruction**: writes to test file some jump instructions taking as input a specific specified operation;
  - **Branch instruction**: writes to test file some branch instructions taking as input a specific operation.

  All the functions write the expected results to the proper golden dump file and the tests into the test files.

- ***Zicsr* test generator**: 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_generator.py`, some functions allow defining a general method to write into files. Similar is also the variables and arrays. The function used to write the files are:

  - **Set Registers**: writes to the test file instructions that set a certain amount of registers with the values `REGS` array;
  - **CSR instruction**: this function allow to write test for a specific CSR instruction on all the control and status registers implemented. 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` and `privlv` (at address `0xC10`);
  - **CSR instruction on specific CSR**: this function allow to write test for a CSR instruction for 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"`;
  - **Illegal CSR Instruction**: this function writes to the text file some illegal instruction that should rise illegal instruction exception. In particular, these instructions are those accessing CSR registers without having privilege.

Since CSR instructions require privileges in order to be executed, the test is split between its dedicated bootloader 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 a pattern. Memory map is the same as Figure 4.5.

4.4 Benchmarking

After Design and Test, we have decided to compare AFTAB and PULPino performances running real applications running real applications. These have been selected form MiBench, a standard benchmark suite shared by University of Michigan. The suit applications are grouped into specific fields from which we have chosen three benchmark belonging to Automotive, Network and Security that are respectively Quick Sort, Dijkstra and Rijndael (base algorithm for AES).

The benchmarks distributed by MiBench have required some changes because of the internal usage of files and print. These could not be exploited because of the absence of the peripherals required to handle such operations. Therefore, the applications have been simply modified substituting files with static variables and removing all printf functions.

4.4.1 Quick Sort

Quick Sort is a well-known algorithm for sorting data inside arrays [13]. This operation is particularly important for embedded systems but also for general-purpose computers because it allows to define priorities, interpret results, organize data and reduce execution time. As expressed by its name, the algorithm aims to sort elements as fast as possible, regardless of the length of the array. The used approach is a divide-and-conquer method that splits an array into two smaller ones and it repeats this process sub-arrays until the algorithm is complete. The Quick Sort steps are the following:

- **Check array size**: proceed with the algorithm if the array size is at least two;
- **Pivote choice**: the first step is to identify a pivot through partition routine (maybe random);
- **Sort Partitions**: reorder the elements of the partitions, in such a way that values lower than the pivot come before a division point, while all elements with greater values come after it; Equal values can go either way. elements that are equal to the pivot can go either way.
- **Recursion on sub-ranges**: Recursively apply the quicksort to the sub-range lower than the pivot and to the greater one, excluding the values equal to the pivot;

According to the partition, Quicksort complexity varies from the best case $O(n \log n)$ to the worst one that is $O(n^2)$. The benchmark code is shown in **??** while a graphical example is shown in Figure 4.6.

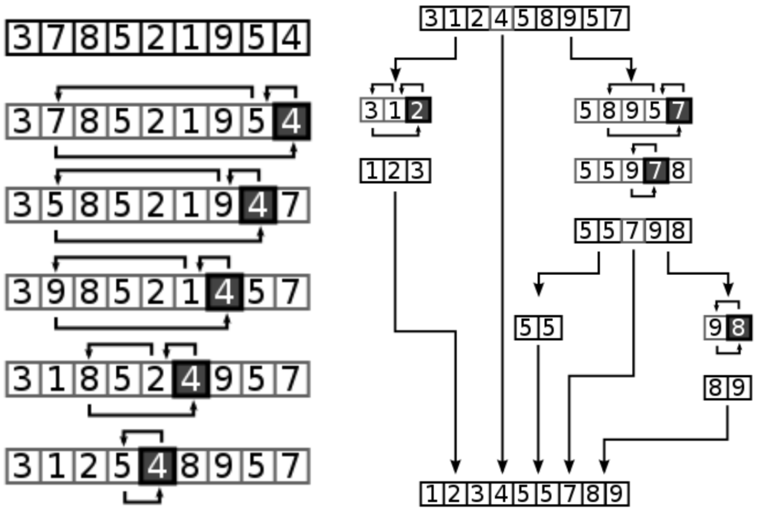

Figure 4.6. Quick Sort Example.

4.4.2 Rijndael

Rijndael is the original name for Advanced Encryption Standard (AES) [8], a widely employed algorithm for encryption of digital data. For the thesis it has not been necessary to deepen the algorithm details, however, this section provides a description of its general behavior. The algorithm embraces the principle of substitution-permutation network and it has good performances for both hardware and software implementations. Moreover, we can also highlight that it features a simple design, ensures compactness and speed, and can resist well-known attacks;

The size of the symmetric key that has been used coincide with one of the standards 128,192 or 256. In order to show the algorithm behavior, a 128-bit key will be considered. Rijndael operates on a square matrix of bytes of dimension 4 and column-major ordered. This contains the current state and is typically represented with a two-dimensional array.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

In Rijndael, encryption with a 128-bit key is performed through nine transformations (named rounds) and is based on byte replacement, swap, and XOR operations. The procedure is as follows:

- **Key expansion**: in this phase, the 128-bit round keys required by the algorithm are generated. Nine are the keys for rounds plus an additional one. All of them are stored into a dedicated four-by-four matrix (array).
- **Plain-text setup**: in this phase the plaintext is divided into 128-bit four-by-four matrices.

- **Round execution**: in this phase, each one of the plain text matrices goes through the following computations:
  - **Byte substitution**: in this step each of the matrix bytes are substituted with a value read from an S-Box Lookup table;
  - **Rows Shifting**: in this step the last three rows of the matrix are shifted by a certain amount of steps;
  - **Column Mix**: the bytes contained by each column are combined performing a linear mixing operation;
  - **Add Round Key**: each byte of the matrix is combined with the round key performing a xor.

  These operations are not performed in all rounds. In the first round, only “Add Round Key” step is performed, whereas in the last one all of them are except “Column Mix”. In the remaining steps, all the steps are executed.

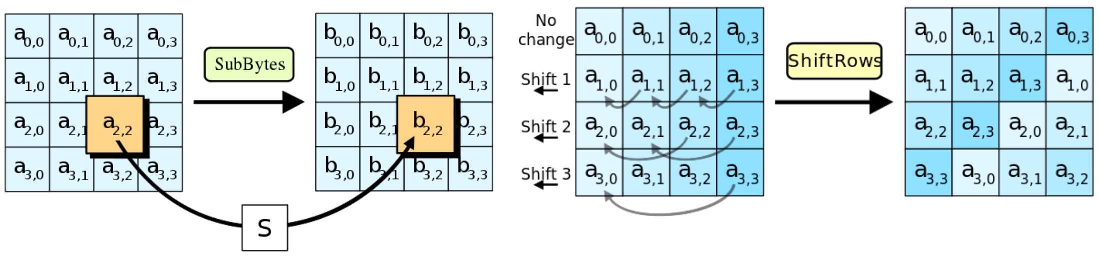

Figure 4.7. Byte Substitution and Rows Shifting operations.

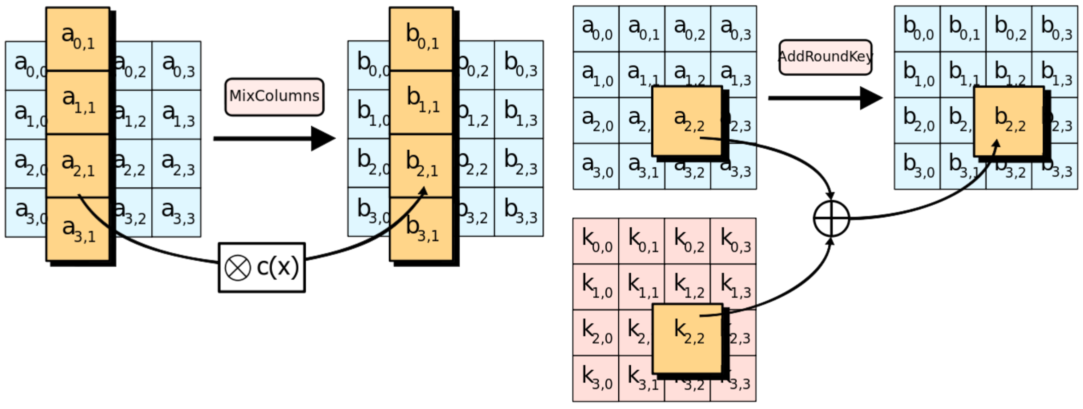

Figure 4.8. Column Mix and Add Round Key operations.

4.4.3 Dijkstra

Dijkstra is widely used algorithm in networking field [6]. Its aim is to identify the shortest path between a couple of nodes in a graph that can be associated to different entities in a real world. As for the precedent algorithm, this Section will not go through the benchmark detail, but it will only describe the algorithm.

In Dijkstra benchmark, a graph is described as an adjacency matrix where every index I,J corresponds to the distance between the two nodes. Every time the algorithm is run two node index are taken as input and one of them is labeled as initial node. Then the algorithm sets the initial distance to infinite and tries to reduce it through step by step graph visits. Here are listed the main steps:

- **Unvisited Set**: the first step is lable all the unvisited nodes including them in a set.
- **Node distances**: the second step aims at setting the distance of the shortest path between the initial node and all the others. Obviously the initial it at zero distance by itself while all the other distance are set to infinite because they are still not known.
- **Visit neighbors**: once distances are set the visit starts form the current node, that is assigned to the initial when the algorithm starts. For all the visited nodes their possible new distance is computed as distance of the current node plus the arch weight. Once the value has been computed it is assigned to the current node if it is lower than the current distance. For example, if current distance is 8 and the computed one is 6 it is changed while if it was 9 it will be kept. This step is repeated for all the neighbors.
- **Mark visited**: when all neighbors have been visited the current node is removed from unvisited set. This implies that it will not be visited again.
- **Conclusion check**: the algorithm ends when the destination node is visited or when the smallest distance between the unvisited nodes is infinitive (this means that there is no connection between source and destination). The algorithm can stop also when the destination has the smallest distance between the unvisited;
- **Set Current node**: if conclusion condition is not verified the current node is set to the one with the lowest distance between the unvisited. After this continue from “Visit Neighbors" step;

Chapter 5

Performance Evaluation

The aim of this Chapter is to show the results obtained during the performed activities focusing on each one of them. In the first phase, the ISA has been extended and some changes have been performed on the design, as explained in 4.1. From the interfacing point of view, the core now features two *read-write* ports for data memory and one *read* port for instruction memory. The addressing is still limited to 32-bit, but this memory interface offers the essential base for future pipeline implementation. The interface has also been extended with 16 external interrupt lines and, consequently, the core has been enabled to handle external interrupts. The internal structure still exploits the Controller-Datapath pattern while the control signals have been corrected and extended as in A.1.1. For the Controller, the internal states have been changed as in A.1.2, while the Datapath has been extended with Control and Status Register Unit (CSRU) and Interrupt Controller. These two-component have allowed handling Privilege Modes (User and Machine) and Exceptions. Moreover, a general refactor of the Datapath has helped to make it more readable. Final result for Control Unit and Datapath are shown respectively in Figure A.2 and A.3.

The design changes have allowed the hardware to support some of the missing RV32IM instructions and implement all those belonging to "*Zicsr*" Extension. Their correct behavior has then been tested through dedicated Assembly applications and validated thanks to the automated test environment. This has allowed us to verify the correct behavior of every instruction logging the memory content into an external file and comparing it to a golden reference at the end of every simulation. Even if this strategy did not allow to perform the test of every single unit, the validation process has speeded up considerably when extending or correcting the RTL description. The implemented tests are internally designed in order to test multiple times every single instruction, checking for each operation both basic behavior and critical cases (division by zero, overflow, signed-unsigned variants, etc..). All the instructions listed by Tables 2.2 and 2.3 have been validated, therefore the correctness of 100% of the implemented instructions can be verified running the tests.

As a final activity, we have adjusted and simulated three well-known algorithms taken from the MiBench benchmark suite. The applications have been run both on AFTAB and one of the most notable examples of the RISC-V-based platform, PULPino. This has allowed us to compare the two core performances. The benchmark algorithms are Quick-Sort, Dijkstra and Rijndael. All of them have been run with the same logic stimulation frequency of 200 MHz (clock period: 5 ns) on both cores. For the QuickSort, we have used one hundred randomly ordered integers that have been sorted in a one-dimensional static array. The Rijndael algorithm has been performed on a simple 16-byte plaintext using a 128-bit key and a 128-bit initialization vector. The application both encrypts and decrypts the interested plaintext and the results are stored in static variables. Dijkstra algorithm has been run ten times on a graph consisting of ten nodes described by an adjacency matrix and the resulting paths have been stored into a static array. The applications simulation times on both AFTAB and RI5CY are listed in table 5.1.

Algorithm	Core	Clock Period	Simulation Time	Boot Time	Program Time
Dijkstra	AFTAB	5 ns	685580 ns	6285 ns	679295 ns
Dijkstra	RI5CY (PULPino)	5 ns	240280 ns	160260 ns	80020 ns
Quick Sort	AFTAB	5 ns	1571840 ns	2220 ns	1569620 ns
Quick Sort	RI5CY (PULPino)	5 ns	355900 ns	159582 ns	196320 ns
Rijndael	AFTAB	5 ns	606540 ns	2810 ns	603730 ns
Rijndael	RI5CY (PULPino)	5 ns	235510 ns	159667 ns	75845 ns

Table 5.1. Benchmaking simulation time on both AFTAB and RI5CY.

The Table shows that the simulation time observed on AFTAB is always greater than the ones obtained running the same programs on RI5CY. If the boot time is excluded, it can be observed that Dijkstra Program Time (Tp) is more than 8 times bigger than the one registered for RI5CY, almost 8 times for the QuickSort, and almost 8 times for Rijndael. These informations are particularly important because they are the first source of comparison with a well-known project such as PULPino, and can be considered as a starting point for reaching similar or better performances. As we expected, in fact, the two cores have very different performances, as RI5CY adopts pipelining while AFTAB does not. For this reason, AFTAB throughput is still much lower and also depends on the instruction distribution of a specific program.

Chapter 6

Conclusions and Future Work

To summarize this thesis work, it is good to emphasize the importance of implementing a RISC-V platform allowing research and experiment in academia. As covered in detail inthe introductory part, RISC-V ISA is an open-source project based on the original Reduced Instruction Set Computer, that has been proposed as an instruction set standard by both academia and companies due to its characteristics. In addition to the saving due to fees absence, it gives the possibility to research groups such as RHES Group Torino to work on several possible projects and train a wide range of skills. During the work on AFTAB, it has been possible to gain experience in several fields both related to software and hardware. From the hardware point of view, the initial version of AFTAB has been extended completing the base instructions set and implementing the extension for handling privileges modes, control and status register instructions, and interrupts. In particular, the most relevant implemented units are the Control and Status Register Unit (CSRU) and the Interrupt Controller. This part has been particularly interesting because has both required to design units but ensure that they could be integrated into the existing system respecting the constraints of the instruction set.

From the software side, in addition to the utilities used during the project, to validate both the parts of the system already present and those implemented later, an environment to perform functional testing has been set up. This environment allows to check the test result at every simulation and compare it with the expected results that have to be prepared for every application. Thanks to it, it has been possible to test some Assembly programs for which both code and results have been generated through dedicated Python scripts. As the last activity, the performance of AFTAB and PULPino has been compared to three well-known benchmarks. The comparison results in Table 5.1 show that AFTAB performance is lower with respect to PULPino, as AFTAB is not a pipelined architecture, and therefore its throughput is necessarily lower. Even if the result is still not close to our target, I consider myself satisfied, as the goal of structuring a stable toolchain around the design has been fully reached.

Moreover, I think that the actual state offers a good base from which several future projects can start. Some of the most relevant are listed in the following:

- **Synthesys on FPGA**: a micro-architecture such as AFTAB can be used in many fields, and one of the RHES Group Torino aims is to exploit it in Capture-the-Flag challenges domain [5]. In this scenario, a possible future work could be based on the synthesis of the core. For this purpose, the platform could be integrated with an automated tool for FPGA synthesis able to speed up the process of generating versions of AFTAB with particular vulnerabilities. In addition, it could also be used to extract and analyze data about the area, timing, and power;

- **Pipelining**: since performance is one of the main limits for AFTAB, implementing pipelining would be a precious update;
- **Units Choice**: AFTAB currently features a Multiplier and Divisor that take respectively 32 and 64 clock cycles to complete. These are designed in order to save space at expense of timing, therefore it could be considered to substitute them with a different algorithm implementation. For the multiplication, it could also be considered pipelining;
- **ISA Extension**: in order to enlarge the range of applications that can be run on the platform, some of the RISC-V extensions can be implemented. For example, the first one could be "*F*" and "*D*" extensions, for single and double-precision floating-point instructions;
- **Interrupt Handling** the interrupt controller currently defines the static priority for Interrupts and Exceptions, but does not implement any queuing mechanism and does not allow to enable each one of them selectively. This would require the extension of the implemented CSRs (see Machine Interrupt Registers MIE and MIP) and the check of the related informations by the interrupt controller.
- **CSRU Extension**: in parallel to AFTAB, one of the RHES Group Torino projects is to design a Secure Real-Time Operating System that could not be run on AFTAB at the current state and would need some extensions. As shown in Figure 2.3, the minimum requirements for running a Unix-like operating system include the Supervisor privilege level that is currently not supported. Therefore, Control and Status Register Unit should be extended to implement all the required CSRs. At the moment, both CSRU structure and register list are essential, and for this reason, it is suggested to revisit the unit starting from the good base developed during this elaborate. For example, all the register individual signals might be grouped into an individual register bank;
- **Peripherals design**: to extend the platform functionalities, possible future extensions may introduce peripherals to the system. As in the PULPino platform, some peripherals that could be implemented are UART, MPU, GPIO, SPI, and I2C. For example, UART could be exploited to log results through the "print" instruction. To realize this idea, it would be good to extend the testbench so that it can redirect the output of the peripheral towards the standard output;
- **Secure Real-Time Operating System**: the RHES Group Torino aims at starting the design of such OS because of the growing need for systems capable of defending against cyberattacks targeting embedded devices. This path can be pursued independently of the AFTAB platform, but can be run on it only after the implementation of the required feature (e.g., supervisor mode);

- **Control Flow Integrity**: as a secure embedded core, AFTAB may be extended to resist binary attacks that aim at changing the order of instruction execution. For this purpose, the AFTAB platform may be extended both with hardware and software solutions. From the hardware point of view, an idea could be to integrate into the core a unit able to perform Control-Flow Integrity. Then, to check the effectiveness of the implementation, it could also be interesting to adapt a software tool for vulnerability assessment;

To sum up, I can say that I feel gratified by this thesis experience, as a student, and as a person. First of all, I consider myself lucky to have taken part in the RHES Group Torino and collaborated with the University of Teheran, because here I found a young and dynamic environment where I also had the opportunity to work on a project that allowed me to observe a microarchitecture from different abstraction levels. It has been particularly interesting to deal with a wide range of activities from the hardware description to the bare-metal programming. In general, I like being able to have an overview and find links between different but similar areas, so I found this context particularly enjoyed this context.

Finally, I would also like to express my gratitude to all those I shared time with during this experience. Within the RHES Group Torino, I was lucky enough to find people with whom I could collaborate but also share moments of leisure. For the thesis experience, I particularly thank Gianluca, who always managed to constantly support both me and all the graduate students. A sincere thanks to Professor Prinetto, who gave me the opportunity to participate in this project.

Appendix A

AFTAB Tecnical documentation

This Appendix aims to provide technical documentation for anyone wishing to understand more detailed aspects about AFTAB. Figure A.1 shows AFTAB pinout:

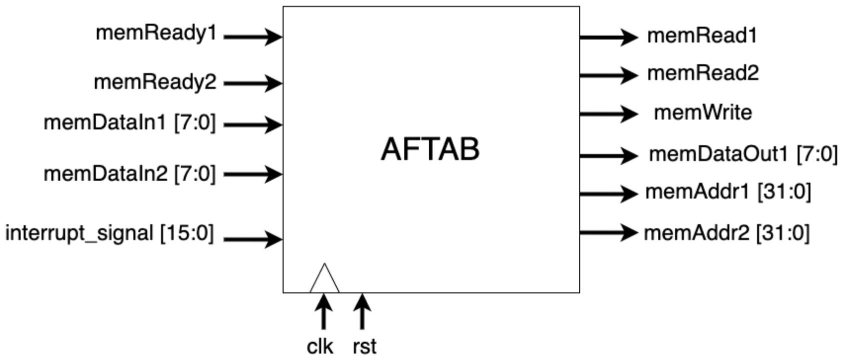

Figure A.1. Pinout of the AFTAB microprocessor.

Signals	Description
`clk, rst`	Fundamental clock and active high synchronous reset
Port 1	Data Interface
`memReady1`	Ready signal for data memory
`memRead1`	Read signal for data memory
`memAddr1`	32-bit address bus for addressing the data memory
`memWrite`	Write signal for data memory
`memDataIn1, memDataOut1`	Two 8-bit data bus for data input and data output from/to the data memory
Port 2	Instruction Interface
`memReady2`	Ready signal for instruction memory
`memRead2`	Read signal for instruction memory
`memAddr2`	32-bit instruction address bus which carries out the content of the Program Counter
`memDataIn2`	8-bit data bus; during load and store, it carries variable number of bytes (B,H,W) in multiple clock cycles
`interrupt_signals`	16-bit interrupt lines bus

Table A.1. AFTAB interfacing port.

A.1 Control Unit

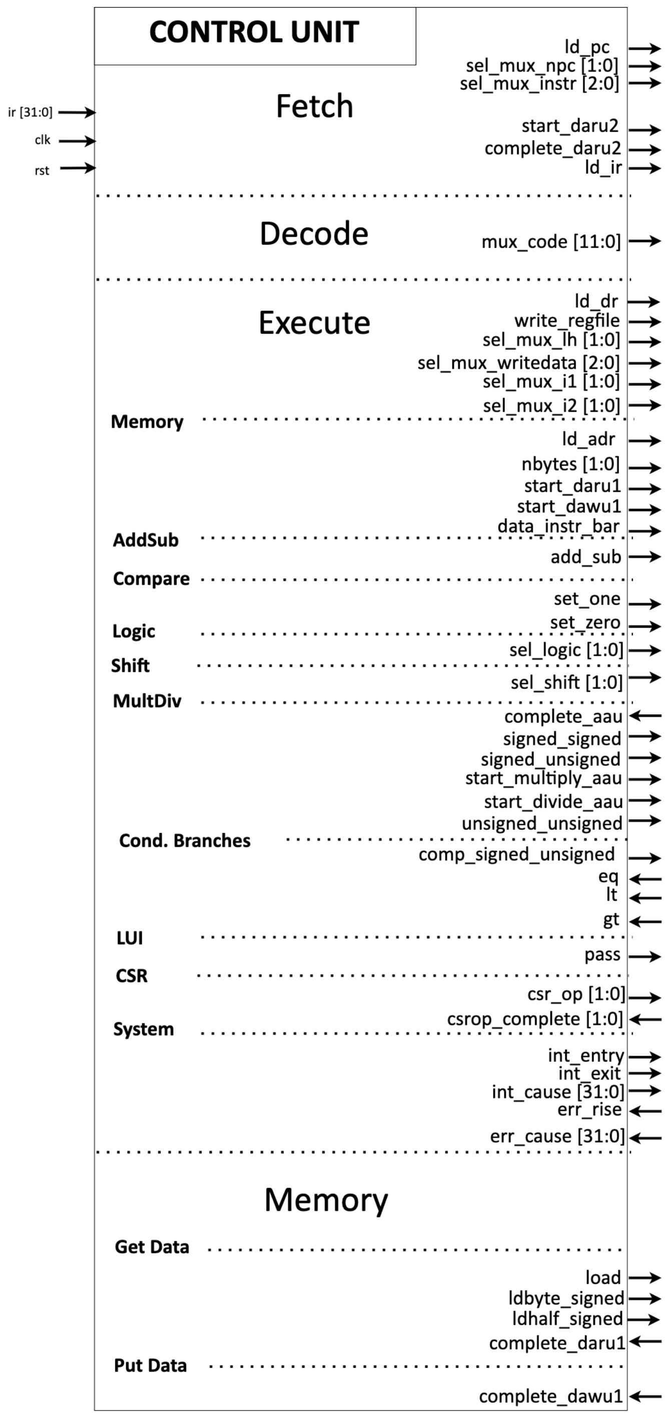

Figure A.2. AFTAB Control Unit.

A.1.1 Control Word

All the control word signal are set to be asserted in the state in which they operate. Tables A.2 and A.3 describe each one of them.

Stage	Control Signal	I/O	Description
`Fetch`	`ld_pc`	`O`	esnable PC load
	`sel_mux_npc[2:0]`	`O`	selects next program counter between: - `csr_epc (010)`; - `csr_tvec (011)`; - pc incremented by 4 `(000)`; - `addResult (001)`; - no selection `(100)`.
	`sel_mux_instr[1:0]`	O	selects instruction between: - `PC (01)`; - read port 1 of register file `(00)`; - no selection`(10)`.
	`start_daru2`	`O`	starts `DARU2` read process
	`completeDARU2`	`I`	set when DARU2 read process is completed
	`ld_ir`	`O`	enables IR load
`Decode`	`mux_code[11:0]`	`O`	code indicating how ISSEU unit has to choose immediate value
`Execute`	`ld_dr`	`O`	enables data register load
	`write_regfile`	`O`	RF write enable
	`sel_mux_lh[1:0]`	`O`	selects result in multiplications or divisions between : - lower part `(00)`; - higher part `(01)`; - no value `(10)`.
	`sel_mux_writedata[2:0]`	O	selects unit result to be written in RF between: - CSRU `(110)`; - AAU `(101)`; - ASU `(100)`; - LLU `(011)`; - BSU `(010)`; - DARU2 output `(001)`; - PC incremented by 4 `(000)`.
	`sel_mux_i1[1:0]`	`O`	selects execute operand 1 between: - output port 1 of RF (`p1`) `(10)`; - current `PC (01)`; - extended unsigned immediate `(00)`; - no selection `(11)`.
	`sel_mux_i2[1:0]`	`O`	selects execute operand 2 between: - output port 2 of RF (`p2`) `(00)`; - immediate `(01)`; - no selection `(10)`.

Table A.2. Control word signals (first part).

Stage	Control Signal	I/O	Description
`Execute-Mem`	`ld_adr`	`O`	enables address load
	`nbytes[1:0]`	`O`	selects number of bytes: - word (11); - halfword (10); - byte (01).
	`start_daru1`	`O`	starts `DARU1` read process
	`start_dawu1`	`O`	starts `DAWU1` write process
`Execute-addSub`	`add_sub`	`O`	selects addition or subtraction
`Execute-compare`	`set_one`	`O`	writes 1 into destination register (for comparison instructions)
	`set_zero`	`O`	writes 0 into destination register (for comparison instructions)
`Execute-logical`	`sel_logic[1:0]`	`O`	selects bitwise logic instruction: - XOR (00); - OR (10); - AND (11).
`Execute-shift`	`sel_shift[1:0]`	`O`	selects shift instruction: - SLL `(00)`; - SRL `(10)`; - SRA `(11)`.
`Execute-multiplyDivide`	`complete_aau`	`I`	tells whether AAU operation is completed
	`signed_signed`	`O`	if set to 1, both multiplication operands are signed
	`signed_unsigned`	`O`	if set to 1, op. 1 is signed and op. 2 is unsigned
	`start_multiply_aau`	`O`	starts multiplication
	`start_divide_aau`	`O`	starts division
	`unsigned_unsigned`	`O`	if set to 1, both multiplication operands are signed
`Execute-conditionalBranch`	`comp_signed_unsigned`	`O`	elects unsigned or signed comparison
	`eq`	`I`	set if comparison returns equal
	`lt`	`I`	set if comparison returns lower than
	`gt`	`I`	set if comparison returns greater than
`Execute-LUI`	`pass`	`O`	set to perform `lui` instruction adder/subtractor consider shifted immediate only
`Execute-CSR`	`csr_op[1:0]`	`O`	selects CSR instruction: - no op `(00)`; - write `(01)`; - set `(01)`; - clear `(11)`.
	`csrop_complete[1:0]`	`I`	tells whether and how CSR instruction is completed: - not completed `(00)`; - complete no write `(01)`; - complete write `(01)`.
`Execute-system`	`int_entry`	`O`	set when an exception is risen
	`int_exit`	`O`	set for exiting an exception
	`int_cause[31:0]`	`O`	Input encoding the system call cause. The ecall is encoded as: - ECALL form U `(00000008)`;
	`err_rise`	`I`	set in case of internal exception (for example, zero division)
	`err_cause[31:0]`	`I`	encodes internal exception cause: - instruction access fault `(00000001)`; - illegal instruction `(00000002)`; - instruction address misaligned `(00000000)`; - store address misaligned `(00000006)`; - load address misaligned `(00000004)`; - store access fault `(00000007)`; - load access fault `(00000005)`.
`Memory`	`load`	`O`	selects word after load data adjusting
	`ldbyte_signed`	`O`	selects signed or unsigned byte after load data adjusting
	`ldhalf_signed`	`O`	selects signed or unsigned halfword after load data adjusting
	`complete_daru1`	`I`	tells whether DARU1 operation is completed
	`complete_dawu1`	`I`	tells whether DAWU1 operation is completed

Table A.3. Control word signals (second part).

A.1.2 States

At each clock cycle, the internal Control Unit FSM can be in one of the states shown in Table A.4.

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 stage 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.
`system`	Exception for System instruction is risen. The instruction may require a change of privilege level.
`CSR`	Datapath is configured to execute a Control and Status register instruction. Control word specifies the instruction that normally requires 3 CCs (2 if write s not performed).
`HANDLE__EXCEPTION1-2`	Datapath is configured to handle a specific Exception specified by the control word.
`ILLEGAL__INSTRUCTION1-2`	Datapath is configured to handle an Illegal Instruction Exception
`multiplyDivide1-2-3`	Datapath is configured to perform Division or Multiplication both require 64 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.

Table A.4. Control Unit states.

A.2 Datapath

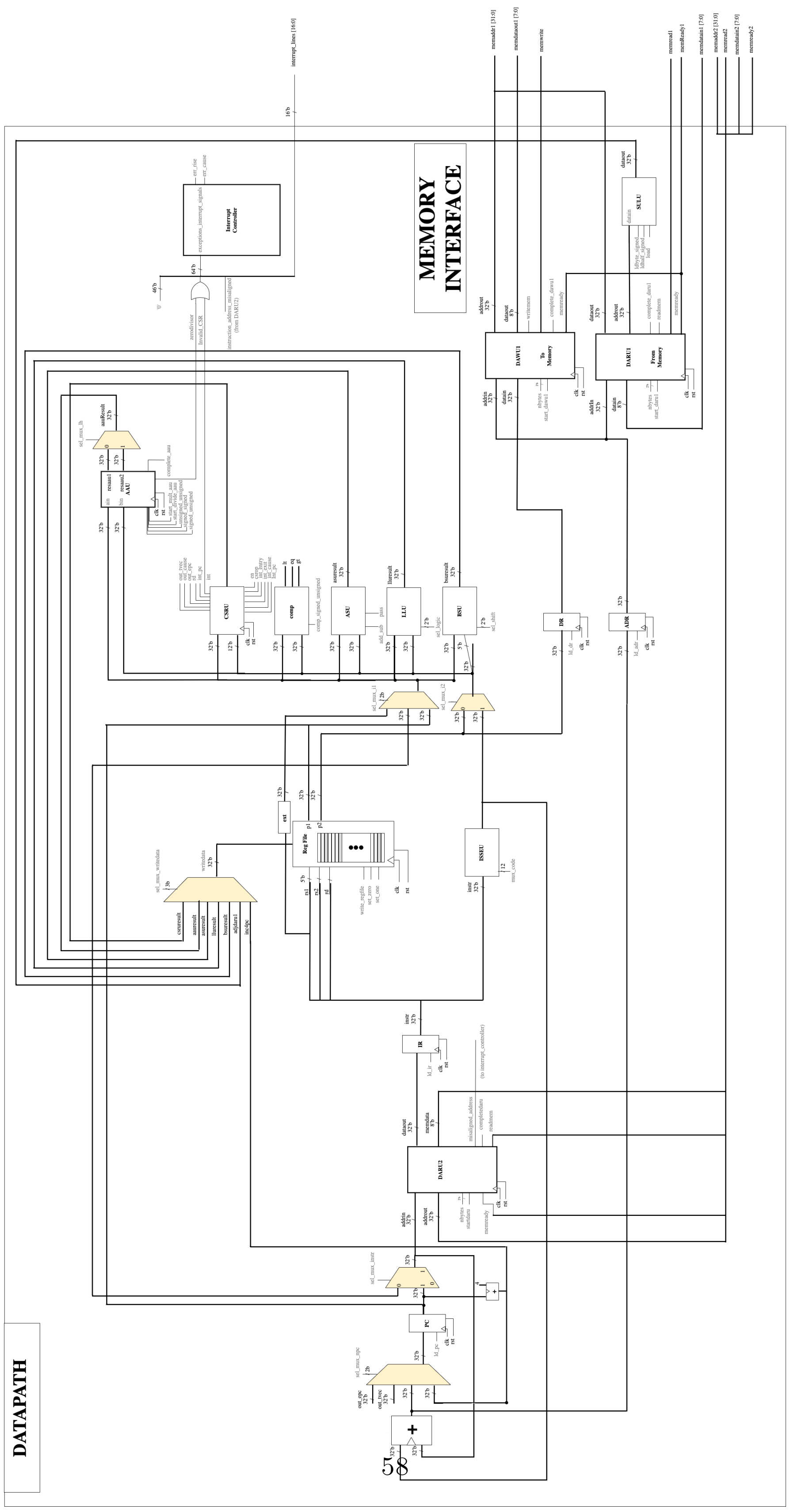

Figure A.3. AFTAB RTL diagram

Appendix B

Benchmarks

This Appendix aims to provide technical documentation for anyone wishing to understand more detailed aspects about benchmarks.

B.1 Quick Sort

```c
extern int nums[] = { // integers to be sorted};

void swap(int*  a,int* b){
 int t = *a;
 *a = *b;
 *b = t;
}

int partition (int arr[], int low, int high){
 int pivot = arr[high];
 int i = (low - 1);
 for (int j = low; j <= high - 1; j++){
  if ((arr[j] < pivot) ? 1 : 0){
   i++; swap(&arr[i], &arr[j]);
  }
 }
 swap(&arr[i + 1], &arr[high]);
 return (i + 1);
}
void quickSort( int arr[],int low, int high){
 if (low < high){
  int pi = partition( arr,low, high);
  quickSort(arr, low, pi - 1);
  quickSort(arr,pi + 1, high);
 }
}
```

```
int main(){
        int n = sizeof(nums) / sizeof(nums[0]);
        quickSort( nums,0, n - 1);
        return 0;
}
```

B.2 Dijkstra

```c
#define NUM_NODES                          10
#define NONE                               9999
#define SHORT_PATH_MAX                     8
#define N_SHORT_PATH                       10
#define NULL ((char *)0)

char graph_arc[SHORT_PATH_MAX*N_SHORT_PATH] = { // initialized}

// Index I,J is the cost for the transition to I to J.
int AdjMatrix[NUM_NODES][NUM_NODES] = {
        32,32,54,12,52,56,8,30,44,94,
        //.....
};

struct _NODE
{
        int iDist;
        int iPrev;
};
typedef struct _NODE NODE;

struct _QITEM
{
int iNode;
int iDist;
int iPrev;
};
typedef struct _QITEM QITEM;


NODE rgnNodes[NUM_NODES];
int ch;
int iPrev, iNode, npath=0;
int i, iCost, iDist;

QITEM qHead[20] = {{.iNode =NONE,.iDist=NONE,.iPrev =NONE},
... // all ad first };
int q_index = 0;
int i_head = -1;

void pop_path (NODE *rgnNodes, int chNode){
 char lines_buff[8];
 int j=0;
 int n_a= 0;
 int node = chNode;

 for (int i = 0; i < 8;i++){
  if (node != NONE || i == 0){
        lines_buff[j] = node;
        n_a++;
        int nodecp = node;
        node = rgnNodes[nodecp].iPrev;
  } else {
        lines_buff[j] = 0;
  }
  if ( j == 7 ) {
   for (int i = 0; i < 8; i++) {
         graph_arc[npath*8+i] = lines_buff[i];
   }
  }else{
        j++;
  }
 }
 npath++;
}
void enqueue (int iNode, int iDist, int iPrev){
        qHead[q_index].iNode = iNode;
        qHead[q_index].iDist = iDist;
        qHead[q_index].iPrev = iPrev;
        q_index++;
        i_head++;
}


void dequeue (int *piNode, int *piDist, int *piPrev){
 if (i_head != -1){
  *piNode = qHead[i_head].iNode;
  *piDist = qHead[i_head].iDist;
  *piPrev = qHead[i_head].iPrev;
  qHead[i_head].iNode =NONE;
  qHead[i_head].iDist=NONE;
  qHead[i_head].iPrev = NONE;
  q_index--;
  i_head--;
 }
}

int qcount (void){
 return(q_index);
```

```
}

int dijkstra(int chStart, int chEnd) {
 for (ch = 0; ch < NUM_NODES; ch++){
  rgnNodes[ch].iDist = NONE;
  rgnNodes[ch].iPrev = NONE;
 }

 if (chStart == chEnd){}
 else {
  rgnNodes[chStart].iDist = 0;
  rgnNodes[chStart].iPrev = NONE;

  enqueue (chStart, 0, NONE);

  while (qcount() > 0) {
   dequeue (&iNode, &iDist, &iPrev);

   for (i = 0; i < NUM_NODES; i++){
        int iCost = AdjMatrix[iNode][i];
    if ((iCost) != NONE){
         if ((NONE == rgnNodes[i].iDist) ||
                (rgnNodes[i].iDist > (iCost + iDist))){
          rgnNodes[i].iDist = iDist + iCost;
          rgnNodes[i].iPrev = iNode;
          enqueue (i, iDist + iCost, iNode);
         }
     }
   }
  }
  pop_path(rgnNodes, chEnd);
 }
}

int main(int argc, char *argv[]) {
        int i,j,k;
        /* finds 10 shortest paths between nodes */
        for (i=0,j=NUM_NODES/2;i<1;i++,j++) {
                j=j%NUM_NODES;

                dijkstra(i,j);
        }

        return 0;

}
```

B.3 Rijndael

```
#include "aes.h"// MiBench Library

/* A Pseudo Random Number Generator (PRNG) used for the     */
/* Initialisation Vector. The PRNG is George Marsaglia's    */
/* Multiply-With-Carry (MWC) */
#define RAND(a,b) (((a= 36969 *(a & 65535) + (a >> 16)) << 16)
+ (b = 18000 * (b & 65535) + (b >> 16)) )

char key[32] = "1234567890ABCDEFFEDCBA0987654321";
char key_char[16]= { /* all '0' */ };
char text_in[16] = {'A','B','1','4','7','9'};
char text_iv[16] = {  /* all '0' */ };
char text_out[16] = {  /* all '0' */ };
char text_out_dec[16] = {  /* all '0' */ };

void fillrand(char *buf, int len){
 static unsigned long a[2], mt = 1, count = 4;
 static char          r[4];
 int                  i;
 if(mt) {
  mt = 0;
  a[0]=0xeaf3;
  a[1]=0x35fe;
 }

 for(i = 0; i < len; ++i){
  if(count == 4){
   *(unsigned long*)r = RAND(a[0], a[1]);
   count = 0;
  }
  buf[i] = r[count++];
  }
}

int encfile( aes *ctx){
 char             inbuf[16], outbuf[16],c;
 unsigned long    i=0, l=0;

 for(i = 0; i < 16; ++i) inbuf[i] = text_in[i];
 fillrand(outbuf, 16);         /* set an IV for CBC mode */
 for(i = 0; i < 16; ++i) text_iv[i] = outbuf[i];
 for(i = 0; i < 16; ++i)      /*xor in previous cipher text*/
 inbuf[i] ^= outbuf[i];
 encrypt(inbuf, outbuf, ctx); /* and do the encryption */
 for(i = 0; i < 16; ++i) text_out[i] = outbuf[i];
 return 0;
}

int decfile( aes *ctx){
 char    inbuf1[16], inbuf2[16], outbuf[16], *bp1, *bp2, *tp;
 int     i, l, flen;

 for(i = 0; i < 16; ++i) inbuf1[i] = text_iv[i];
 for(i = 0; i < 16; ++i) inbuf2[i] = text_out[i];
 decrypt(inbuf2, outbuf, ctx);   /* decrypt it */
 for(i = 0; i < 16; ++i)         /* xor with previous input*/
  outbuf[i] ^= inbuf1[i];
 for(i = 0; i < 16; ++i) text_out_dec[i] = outbuf[i];
 return 0;
}

int main(int argc, char *argv[]){
 char    *cp, ch;
 int     i=0, by=0, key_len=16, err = 0;
 aes     ctx[1];
 cp = key;

while(i < 32 && *cp){   /* max key length is 32 bytes */
ch = *cp++;             /* process a hexadecimal digit  */
if(ch >= '0' && ch <= '9')
 by = (by << 4) + ch - '0';
else if(ch >= 'A' && ch <= 'F')
 by = (by << 4) + ch - 'A' + 10;
else                    /* error if not hexadecimal     */
 return 0;

/* store a key byte for each pair of hexadecimal digits*/
if(i++ & 1)
 key_char[i / 2 - 1] = by & 0xff;
}

set_key(key_char, key_len, enc, ctx);
err = encfile( ctx);
set_key(key_char, key_len, dec, ctx);
err = decfile( ctx);
return 0;
}
```

Bibliography

[1] Michael Gautschi Andreas Traber. *PULPino: Datasheet*. English.

[2] Pasquale Davide Schiavone Andreas Traber Michael Gautschi. *RI5CY: User Manual*. English. Version 4.0.

[3] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. English. Version 20191213.

[4] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. English. Version 20190608.

[5] Sanjeev Das, Wei Zhang, and Yang Liu. «A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.11 (2016), pp. 3193–3207. DOI: `10.1109/TVLSI.2016.2548561`.

[6] Saul I. Gass and Carl M. Harris. *Encyclopedia of Operations Research and Management Science."Dijkstra's Algorithm"*. English. 2013.

[7] John L Hennessy and David A Patterson. *A Quantitative Approach: Computer Architecture*. 1995.

[8] Federal Information. *Announcing the Advanced Encryption Standard (AES)*. English. 2017.

[9] Dan Ernst Todd M. Austin Trevor Mudge Richard B. Brown Matthew R. Guthaus Jeffrey S. Ringenberg. *MiBench: A free, commercially representative embedded benchmark suite*. 2001.

[10] *PULP platform*. URL: `https://github.com/pulp-platform/pulpino`.

[11] Colin Riley. *RISC-V CPU*. URL: `https://github.com/Domipheus/RPU`.

[12] *RISC-V GNU Toolchain*. URL: `https://github.com/riscv-collab/riscv-gnu-toolchain`.

[13] R. Sedgewick. *Implementing Quicksort programs*. English. 1978.

[14] Frank Vahid. *Digital Design with RTL Design, VHDL, and Verilog*. John Wiley and Sons.

[15] *VHDL Style Guide*. URL: `https://github.com/jeremiah-c-leary/vhdl-style-guide`.