INSTITUTO TECNOLÓGICO DE COSTA RICA

MICRO-ARCHITECTURAL SPECIFICATION

# TEC-RiscV

*Author:*
DCI-LAB

*Supervisor:*
Dr-Ing Renato Rimólo
Dr-Ing Alfonso Chacón
MSc-Ing Ronny García

DCI-Lab
Electronics Engineering Department

November 7, 2020

*"The cheapest, fastest, and most reliable components are those that aren't there"*

C.Gordon Bell

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **ALU** | **A**rithmetic and **L**ogic **U**nit |
| **CAM** | **C**ontent **A**ddressable **M**emory |
| **CPU** | **C**entral **P**rocessing **U**nit |
| **CSR** | **C**ontrol and **S**tatus **R**egister |
| **FIFO** | **F**irst **I**n **F**irst **O**ut |
| **GPR** | **G**eneral **P**urporse **R**egister |
| **ISA** | **I**structions **S**et **A**rchitecture |
| **MBC** | **M**emory and **B**us **C**ontroller |
| **MCAUSE** | **M**achine **Cause** Register |
| **MEIE** | **M**achine **E**xternal **I**nterrupt **E**nable bit |
| **MEIP** | **M**achine **E**xternal **I**nterrupt **P**ending bit |
| **MEPC** | **M**achine **E**xeption **P**rogram **C**ounter |
| **MIE** | **M**achine **I**nterrupt **E**nable Register |
| **MIP** | **M**achine **I**nterrupt **P**pending Register |
| **MISA** | **M**achine **I**nstructions **S**et **A**rchitecture Register |
| **MPIE** | **M**achine **P**revious **I**nterrupt **E**nable bit |
| **MSTATUS** | **M**achine **Status** Register |
| **MTIE** | **M**achine **T**iming **I**nterrupt **E**nable bit |
| **MTVEC** | **M**achine **T**rap **V**ector Base Address register |
| **RISC** | **R**educed **I**nstruction **S**et **C**omputer |

# Chapter 1

# The Big Picture

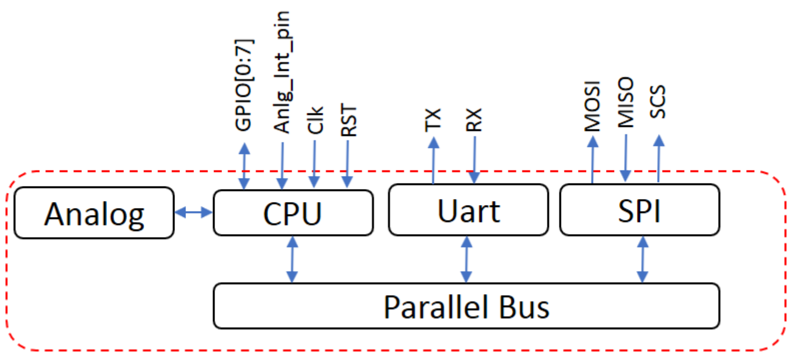


FIGURE 1.1: High level description of the System on Chip(SOC)

## 1.1 Introduction

This document is intended to specify the micro-architectural implementation of a system on chip (SOC) including a RISCV RV32I central processing unit (CPU), a universal asynchronous receiver-transmitter interface (UART), a serial peripheral interface (SPI), 8 general purpose input/output interfaces (GPIO), and an analog block intended for biological stimulation. Figure 1.1 present the main functioning blocks for this system; a basic description of each unit in the micro-architecture as well as the signals attached to them is provided latter in this chapter. There will be an specific section explaining the implementation details for the most complex structures in the micro-architecture. It is not the intent of this document to describe the details of the RISCV 32I programming model and instructions, in order to get a deeper knowledge in these areas you can refer to [1]–[3] and the official cite of the RISCV foundation.

### 1.1.1 BOOT

The boot process for this system is supported by a an external 16MB flash memory connected to the SPI interface of the system, in this memory the program is stored and automatically loaded into the internal RAM memory after full system reset. The details about the internal functioning of the boot process will be covered in the SPI and MBC descriptions.

### 1.1.2 Memory Map

In order to save area in the implementation the memory map for the system is hard-coded as presented in figure 1.2.

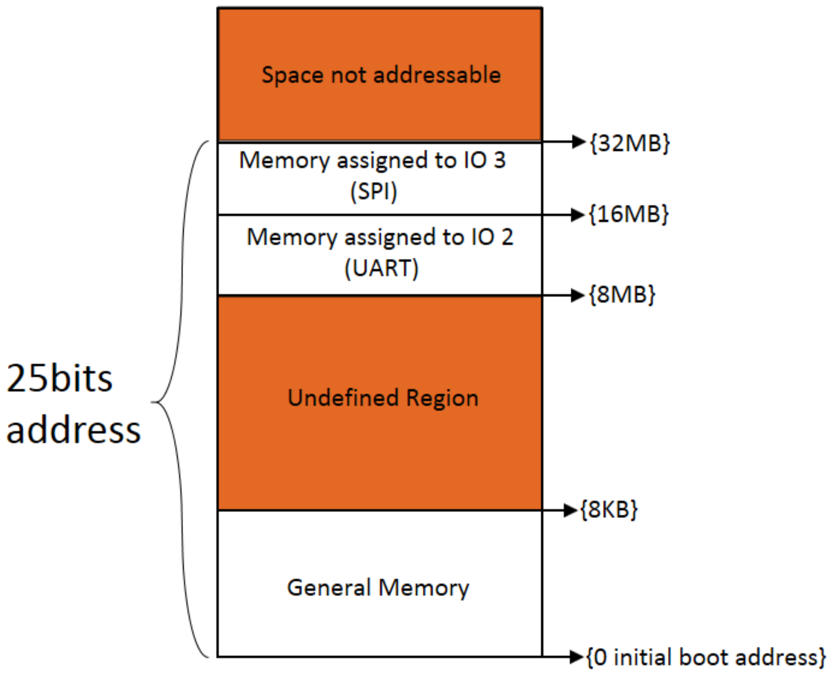


FIGURE 1.2: Memory Map

The MBC will direct every access in the range from 0 to 8KB to the RAM memory while the accesses in the range from 8MB to 32MB will be directed to the SPI and the UART; accesses to the region from 8KB to 8MB will trigger a bad address interruption and will be managed by the interrupt handler software.

### 1.1.3 Interruptions

There are two sources of non-maskable interruptions in this implementation: the ones generated when there is an invalid instruction in the instruction decoder and the ones generated when there is an access to an invalid address. Also there are three sources of maskable interruptions: the ones coming from external devices connected to the bus, SPI or UART, the ones coming from the internal timer and the ones coming from an external pin (these are called analog interruptions in this document). The central control unit of the CPU will check for interruptions after loading the next instruction in the program counter (PC) and before the fetch of the instruction from memory and into the ID; the behavior of the machine whenever an interruption is detected is the same for both maskable and non-maskable interruptions: The PC will be loaded into the mepc CSR, all the interruptions will be disabled for the execution of the interrupt controller and the PC will be loaded with the content of the pointer to the interrupt controller software which is stored in the mtvec CSR.

Ones the interrupt handler software is loaded, it will check the mcause_A and mcause_B CSRs in order to obtain information of the interruption which need to be serviced. In figure 1.3 it is a description of the all the possible interruption vectors for this implementation, the interrupt handler software can identify the interruption

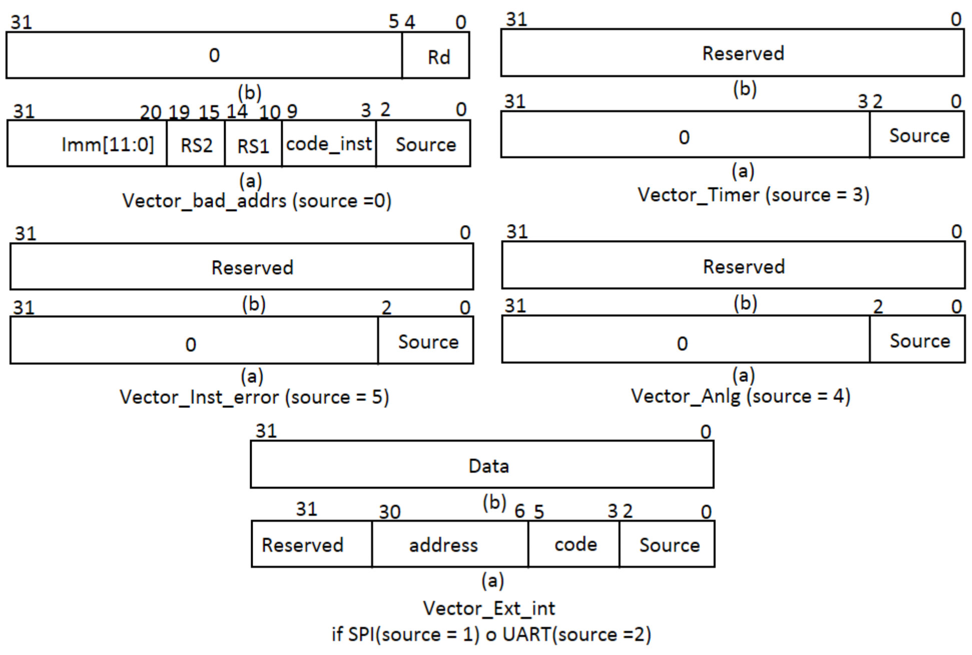


FIGURE 1.3: Interrupt Vectors loaded into the Mcause CSRs

by checking the source field in the mcause_A register, and the rest of the fields will provide valuable information in order to service each possible interruption. In the case of interruptions coming from the SPI or UART (also called external interruptions) the code correspond with the intention of the transaction and is codified as presented in table 1.1 (A). Table 1.1 (B) correspond with teh IDs used by the bus to identify each external device and will be discussed next in the BUS documentation

TABLE 1.1: Codes for the data package

| **Action** | **Code**$_{hex}$ $\{\overline{R/W},\mathrm{H,B}\}$ |
|---|---|
| Read 1 byte | 1 |
| Read 2 byte | 2 |
| Read 4 byte | 0 |
| Write 1 byte | 5 |
| Write 2 byte | 6 |
| Write 4 byte | 4 |
| Boot_start | 7 |
| Boot_end | 3 |

(A) Operation codes

| **IO** | **Code**$_{hex}$ |
|---|---|
| CMB | 0 |
| SPI | 1 |
| UART | 2 |
| ⋮ | ⋮ |

(B) IO identification

### 1.1.4 The Bus

The central bus used to communicate the MCB with the IO devices. Notice that in this implementation there is only one block of memory and it is embedded in the MCB. The bus interface is implemented using two FIFOs to interface the memory controller with each one of the devices, one for the input data and one for the output data; this is illustrated in figure 1.5. If there is any data in the input FIFO, meaning

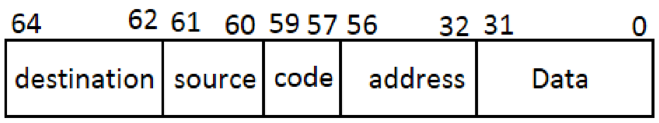


FIGURE 1.4: Format of the package of information used by the BUS

there is a new message for the device, the “pndng” signal of the input FIFO will be asserted, ones the data is processed by the device, it can be removed from the FIFO with a pulse in the “pop” signal. Is important to notice that the pulse in the pop and push signal must endure only one clock cycle if it takes more than that it may push or pop more that one entry. If there are still pending messages in the Input FIFO the “pndng" signal will remain asserted, other wise it will go down. If the FIFO is full, the “full” signal will be asserted and if a new data is pushed while the FIFO is full, then the oldest entry in the FIFO will be evicted. In order to send any data using the bus the device must push the data into the output FIFO with the ID of the intended recipient of the message in the 8 most significant bits. For the received messages, the most significant 8 bits will always match the BUS ID of the device,so these can be ignored and not sent to upper application layers. The depth of the FIFOs is configurable and will depend on the speed at which the devices are capable of processing the data compared with the traffic expected from the system. A detailed description of the internal design of the bus is provided later.

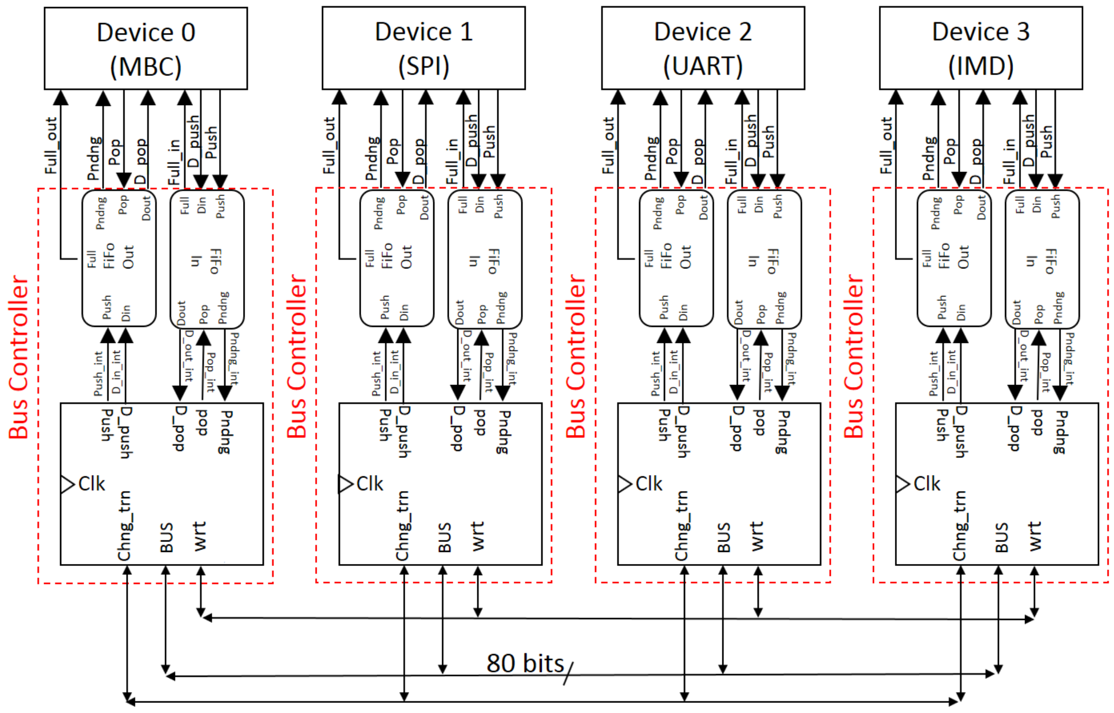


FIGURE 1.5: Example of the bus implementation, in this figure there is a missing interfaces to the Analog actuators

The package used to transfer information in the bus is formated as presented in figure 1.4 here the destination and the source of the package of information are encoded using the IDs presented in table 1.1 (B) while the code of the intended transaction is encoded as presented in table 1.1 (A).

## 1.2 Description of the Central Processing Unit

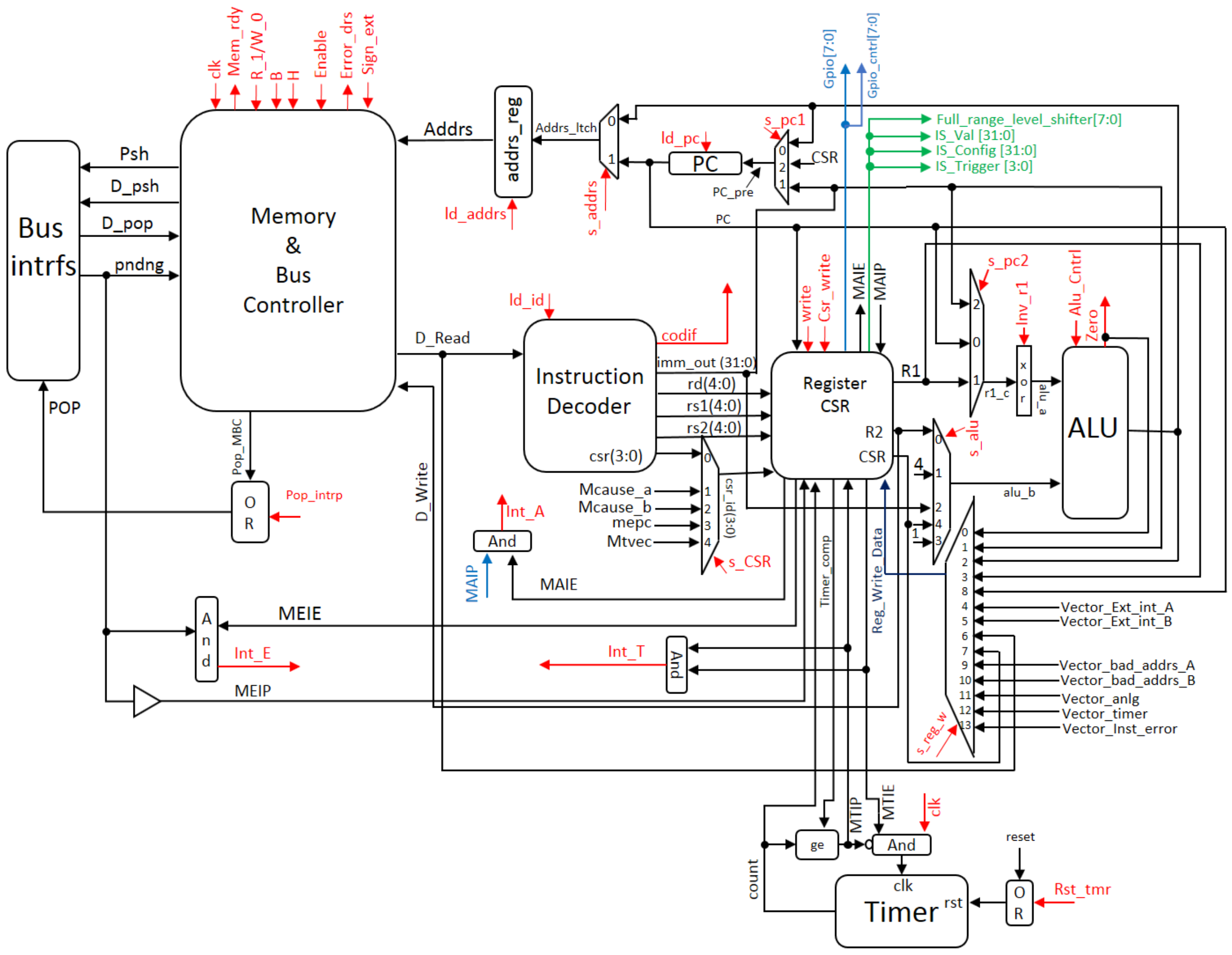


FIGURE 1.6: Basic Diagram of the implemented CPU micro-architecture

The Central Processing Unit for this SOC is a microprogrammed implementation of a RISCV 32I architecture, the general block diagram for this micro-architecture is presented in figure 1.6, four general blocks: the Memory and bus controller (MBC), instruction decoder(ID), the register file (RF) and arithmetic and logic unit (ALU) will be in charge of performing all the instructions in the architectural implementation [1], coordinated by a central control unit. In figure 1.6 all the signals going to and from the central control unit are colored red; however, the signals going to the analog interface are colored green, and the signals directly connected to the chip boundary are colored blue; it is important to highlight that there are some connections missing in figure 1.6 diagram in order to make it readable.

### 1.2.1 Caveats for this RISCV 32I micro-architectural implementation

- This implementation assume all the code has machine mode privileged, then, it does not provide support for user, supervisor and hipervisor mode. For example, Physical Memory Protection (PMP) [1], [2].

- This implementation is a single thread implementation, so there is no support for features intended to multiple hardware/software threads. For example, Fences [1], [2].
- The Control and Status Register (CSR) IDs from the standard RISCV ISA have been changed in order to improve the area utilization also merging some CSRs in the architectural implementation into only one CSR.
- Misaligned access to the memory is not allowed and will trigger an exception.

### 1.2.2 Instruction decoder

This block is in charge of decoding the instructions stored in memory and provide the system with the information required for the correct execution of the code, as well as to identify illegal instructions. There are 6 different instruction formats in RISCV 32E available for the user and privileged mode as shown in figure 1.7.

| 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 1.7: Format of the instructions included in the RISCV specification for 32I [1]

Based on these instruction formats RISC-V provides 47 instructions (plus the pseudo instructions) which are available from user mode [1], in this implementation the instructions "Fence" and "Fence.i" are not implemented because this micro-architecture has only one physical thread. In the case where there is an instruction not recognized, the code returned to the control will be all bits in 1 and this will be equivalent to have an illegal instruction exception. For each one of the supported instructions this block generates the fields required in order to execute the instruction correctly, for example: in some cases the immediate output (imm) is sign extended while in other it is zero extended.

#### Description of inputs and outputs

A basic description of the inputs and outputs of the Instructions decoder is presented in figure 1.8.

**Data_Read**: This is a 32 bits input containing the data coming from memory which in some cases may be a 32 bits instruction and in other just data being read from memory.

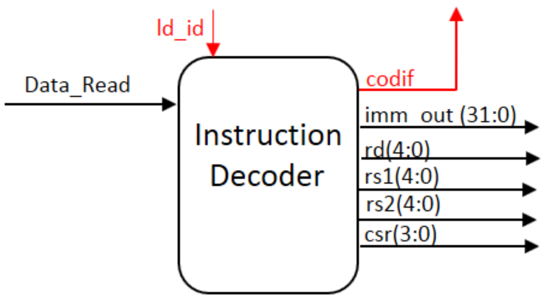


FIGURE 1.8: Block diagram for the instructions decoder

**Ld_id**: This single bit input is a rising edge trigger to capture the data coming from memory and start the data decoding process. This input comes from the control unit.

**Codif**: This 7 bits output correspond to a unique code assigned to identify which instruction has been decoded, the codes assigned to each instruction are listed below in table 1.2. The different kinds of instructions can be identified according to the codes shown in table 1.3.

**Imm**: This 32 bits output preset any immediate operant included in the instruction, in some cases is going to be shifted or sign/zero extended.

**rd**: This 5 bits output correspond to the identifier of the target register, this is a register where information is going to be written. In the cases where there are no register modifications this output will be zero.

**rs1**: This 5 bits output correspond to the identifier of the first source register. In the case where this is not required this output will be zero.

**rs2**: This 5 bits output correspond to the identifier of the second source register. In the case where this is not required this output will be zero.

**CSR**: This 4 bits output is an identifier for a target CSR. In the cases where no CSRs are accessed, this output should remain in zero. In future iterations of this architecture this field could be merged with the rs2 field for the CSR instructions because this field is not used in CSR instructions. In the RISCV standard the CSR IDs are 12 bits long, however since the standard CSR IDs are not being used here this output has been reduced to 4 bits.

### 1.2.3 Register File

This block is responsible for accessing and writing all the system internal registers. There are 32 general purpose registers in the RISCV 32I definition, and 16 in the RISCV 32E definition. [1], and it is possible to read and write from the same register in the same instruction (first read then write).

TABLE 1.2: List of instructions supported and the code assigned by the instruction decoder to each one of them

| **CODIF[6:0]** |
|---|
| 0001000 –> LUI (No ALU) |
| 0000000 –> AUIPC (No ALU) |
| 1000000 –> JAL (No ALU) |
| 0000010 –> BEQ (ALU: EQU) |
| 0001010 –> BNE (ALU: NEQU) |
| 0100010 –> BLT (ALU: LESS_THAN) |
| 0101010 –> BGE (ALU: GREATER_THAN) |
| 0110010 –> BLTU(ALU: LESS_THAN, unsigned) |
| 0111010 –> BGEU(ALU: GREATER_THAN, unsigned) |
| 0000011 –> SB (No ALU) |
| 0001011 –> SH (No ALU) |
| 0010011 –> SW (No ALU) |
| 0000110 –> JALR (No ALU) |
| 0000100 –> LB (No ALU) |
| 0001100 –> LH (No ALU) |
| 0010100 –> LW (No ALU) |
| 0100100 –> LBU (No ALU) |
| 0101100 –> LHU (No ALU) |
| 1100100 –> ADDI (ALU: add signed) |
| 1110100 –> SLTI (ALU: less_than signed) |
| 1011100 –> SLTIU (ALU: less_than unsigned) |
| 1100100 –> XORI (ALU: bit wise xor) |
| 1110100 –> ORI (ALU: bit wise or) |
| 1111100 –> ANDI (ALU: bit wise and) |
| 0001101 –> SLLI (ALU: shift left logical inmediate) |
| 0101101 –> SRLI (ALU: shift right logical inmediate) |
| 1101101 –> SRAI (ALU: shift right aritmetic inmediate) |
| 0000111 –> ADD (ALU: add unigned aritmetic overflow ignored) |
| 1000111 –> SUB (ALU: sub usigned aritmetic overflow ignored) |
| 0001111 –> SLL (ALU: Shift left) |
| 0010111 –> SLT (ALU: compare less than signed) |
| 0011111 –> SLTU(ALU: compare less than unsigned) |
| 0100111 –> XOR (ALU: bitwise Xor) |
| 0101111 –> SRL (ALU: Shift rigth logic) |
| 1101111 –> SRA (ALU: Shift rigth aritmetic) |
| 0110111 –> OR (ALU: bitwise or) |
| 0111111 –> AND (ALU: bitwise and) |
| 0100001 –> ECALL (No ALU) |
| 0101001 –> EBREAK (No ALU) |
| 0111001 –> MRET (No ALU) |
| 1100001 –> WFI (No ALU) |
| 0010001 –> CSRRW (No ALU) |
| 0010001 –> CSRRS (ALU: bitwise OR) |
| 0011001 –> CSRRC (ALU: bitwise AND) |
| 1001001 –> CSRRWI (No ALU) |
| 1010001 –> CSRRSI (ALU: bitwise OR) |
| 1011001 –> CSRRCI (ALU: bitwise AND) |
| 1111111 –> INVALID INST |

TABLE 1.3: Different kinds of instructions grouped by code

| **CODIF[6:0]** |
|---|
| 0XXX000 –> U-type Instruction |
| 1XXX000 –> J-type Instruction |
| XXXX010 –> B-type Instruction |
| XXXX011 –> S-type Instruction |
| XXXX100 –> I-type Instruction |
| XXXX101 –> R-type Instruction (SLLI,SRLI,SRAI) |
| XXXX110 –> I-type jalr |
| XXXX111 –> r-type instructions (add,sll.slt,sltu,xor,srl,or,and,sub,sra) |
| XXXX001 –> Machine-Mode-I-type Instruction |

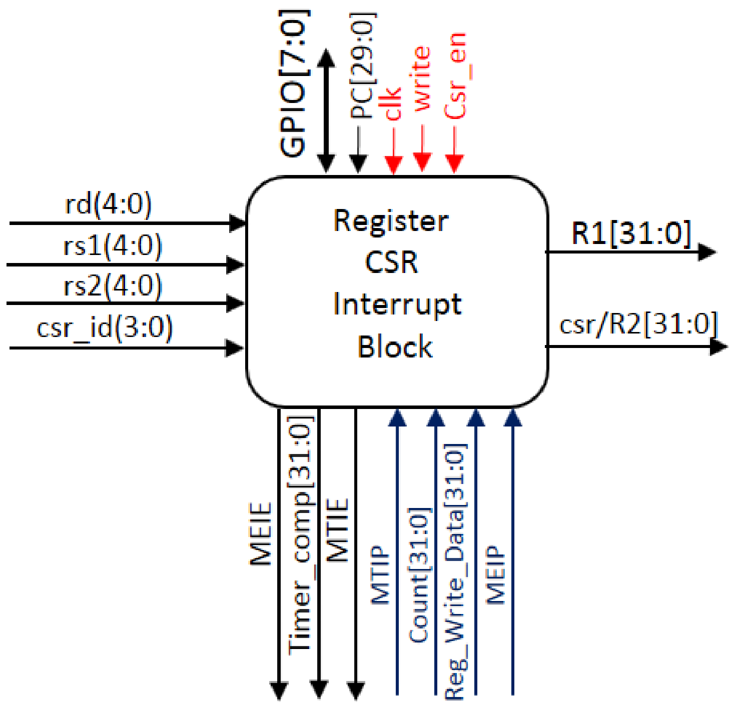


FIGURE 1.9: Basic Diagram of the Register file proposed

### Input and output description

The proposed main block for the Register File, CSRs and interrupt block, is presented in Figure 1.9, with its inputs and outputs:
**Control signals:**

- **Write**: This signal will write the data coming in the Reg_write_Data interface into the selected register.
- **Csr_en**: When asserted the register file will show the contents of the CSR selected in the csr_id input in the csr/R2 output. Also if this signal is asserted and a write is flagged then the CSR selected in the csr_id input will be written with the data coming from the Reg_write_Data interface

**Outputs going to the ALU:**

- **CSR/R2**: This output will be equal to the content of the R2 register except when the CSR input is not zero, in that case the output will be equal to the content of the CSR register pointed in the CSR_ID input.
- **R1**: This 32 bit output is always going to show the content of the register pointed by the rs1 input.

**Inputs coming from the Instruction decoder:**

- **rd**: This input contains the identifier of the destiny register for write operations.
- **rs1**: This input contains the identifier of the first source register.
- **rs2**:This input contains the identifier of the second source register.
- **csr_id**: This four bits input contain the identifier of the csr that is going to be used in the case that the csr_id input is asserted.

**Reg_Write_Data**: This 32 bit input contain the data which is going be written in the general purpose register or the CSR pointed by the rd or the CSR input respectively.

**Interface to timer**

- **Count**: This input can be observed in the CSR_id 7; and contain the current value of the count for the internal timer used to generate interruptions.
- **Timer_comp**: This output contain the frontier value of the timer before it trigger an interruption and can be configured using the CSR_id 6.
- **MTIE**: This output inform the timer logic when the timer interruptions are enabled.
- **MTIP**: This input signals set a bit in a CSR when there is a timer interruption pending to get service.

**Other signals of the Register File:**

- **Clk**: This is the general clock of the system, which is required to get the state machine that controls the access to the interrupt controller.

- **Reg_write_Data**: This is the data to be written in the CSR or General Purpose Register being pointed by the rd or the csr_id input when the write signal is asserted.

- **PC**: This input contain the current value of the Program counter Register.

- **GPIO**: This is an interface with the General Purpose Input Output pins the system.

- **MEIE**: This output inform the control logic when the external interruptions are enabled.

- **MEIP**: This input flags that there is a pending External interruption to be serviced.

### General Purpose Registers

There are 32 general purpose registers included in the RISCV-32I specification and 16 in the RISCV32E specification, all of them with 32 bits. All the general purpose registers can be written by the software except for the register R0, which is hardcoded to 0X00000000 and is used to implement several pseudo-instructions[3].

### CSR Registers

The Control and Status Registers CSRs are used to configure the way the processor will work as well to show the current status of the machine. Some of these registers (or part of some registers) are not writable using software because these represent the status of the hardware.

The addresses for the CSR registers proposed in [2] are not being followed in order to reduce the area of the register file module; in table 1.4 the address of each CSR register is presented, the “original Addrs" column have the directions of the specification in [2] and the “Implementation Addrs" column have the location of each machine mode CSR in the implemented micro-architecture.

A Brief description of the implemented CSRs is provided next; a complete description of all the possible registers is provided in [2].

**Machine Interrupt Pending register (Mip):**
The mip register save information on pending interrupts,there are three different kinds of interrupts: I/O, software, and timing; these are not writable from software. The MTIP (Machine Timer Interrupt pending) bit is set when there is a pending interrupt coming from the timer. in order to clean this bit the timer must be reset. There are some bits in this register related to functionalities in user and supervisor mode which are not implemented because in this architecture there is only one hart and it works in machine mode. The MEIP bit (Machine External Interrupt Pending) is asserted when there is and interruption pending from I/O. The implementation

TABLE 1.4: CSR Identification for the implemented control and status registers

| Register | Original Addrs | Implementation Addrs |
|---|---|---|
| mie | 0x304 | 0x00 |
| mip | 0x344 | 0x01 |
| mepc | 0x341 | 0x02 |
| mcause | 0x342 | 0x03 |
| trapB | - | 0x04 |
| trapC | - | 0x05 |
| mvtec | 0x305 | 0x06 |
| CSR_IO | - | 0x07 |

of this register is shown in Figure 1.10; the bits in blue are wired to zero. The MIP CSR has been implemented in this micro-architecture using the CSR Id 1. The two implemented bits of this register comes from the Interrupt controller.

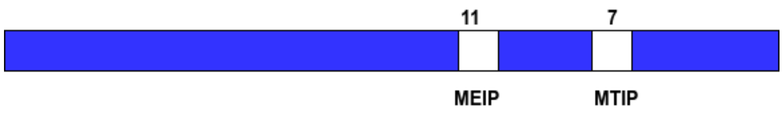


FIGURE 1.10: Mip Implementation

**Machine Interrupt Enable register (Mie):**
Mie contains interrupt enable bits [2]. As in the MIP register there are only two bits applicable to the micro-architecture proposed here and all the other bits are wired to 0. The mtie (machine timer interrupt enable) bit enables the timer interrupts. this implementation this bits enable the timer to start counting; this timer is physically implemented in the Memory controller because the CSRs controlling the timer are mapped in memory.The MEIE bit enables external interrupts. Both the MEIE and the MTIE are used in the In figure 1.11 the implementation of this register for the proposed micro-architecture is presented, ones again the bits wired to zero are presented in blue. The CSR Id selected for this register is 0.

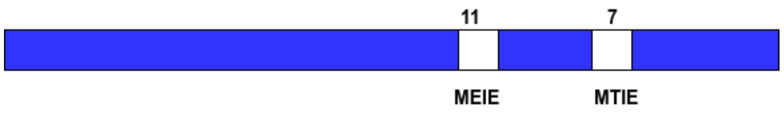


FIGURE 1.11: Mie Implementation

**Machine Exception Program Counter (Mepc)**
The two low bits (mepc[1:0]) are always zero because all instructions must be aligned to four bytes. When a trap is taken into M-mode, mepc is written with the virtual address of the instruction that encountered the exception. Otherwise, mepc is never written by the implementation, though it may be explicitly written by software. In figure 1.12 the implementation of this CSR is presented. The CSR Id selected for this register is 2. The program counter is directly connected to the input of this register.

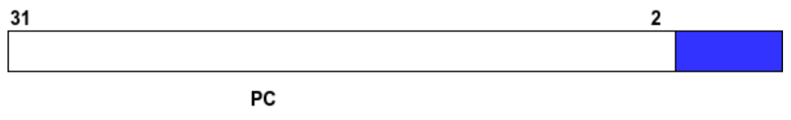


FIGURE 1.12: Mepc implementation

**Machine Trap-Vector Base-Address Register (Mtvec):**
Holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

TABLE 1.5: Encoding of mtvec MODE field.

| Value | Name | Description |
|---|---|---|
| 0 | Direct | All exceptions set PC to BASE |
| 1 | Vectored | Asyncronous interrupts set PC to BASE +4X cause |
| $\geq 2$ | $--$ | Reserved |

In the Specification [1], the encoding of the MODE field is shown in Table 1.5 When MODE=Direct, all traps into machine mode cause the pc to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the PC to be set to the address in the BASE field, whereas interrupts cause the PC to be set to the address in the BASE field plus four times the cause. In order to save hardware, this feature has been disabled and in all the exceptions and interruptions the machine will jump to the base address, and the lower two bits of the MTVEC has been wired to zero. The CSR Id selected for this register is 6.

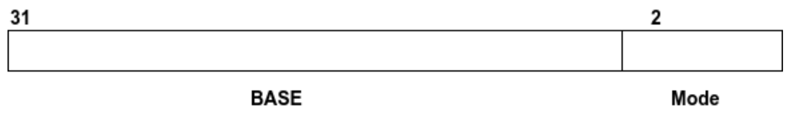


FIGURE 1.13: Mvtec implementation

**Machine Cause Register (Mcause):** When a trap or interruption is taken into M-mode, mcause is written with information about the event that caused the trap. Otherwise, mcause is never written by the implementation, though it may be explicitly written by software. According to [2] this is a 32 bits register, however here is implemented as three complementary 32 bits registers which store information that may be required by the software routine used to provide service to the interruptions. The mcause register configuration is shown in figure 1.14; the registers represented in (a), (b) and (c) can be addressed in the CSR ids 3-5 in that order.

The fields presented in figure 1.14 can be interpreted as follow: the “Interrupt" bit in the mcause register is set to 1 if the trap was caused by an interrupt and is set to 0 if the trap is caused by an exception, (the main difference is that interruptions can be masked while exceptions cannot), The “Code" field contains a code to identify the interruption within the source, the “Source" field identify the source of the interruption within the micro-architecture (he available sources are presented in the

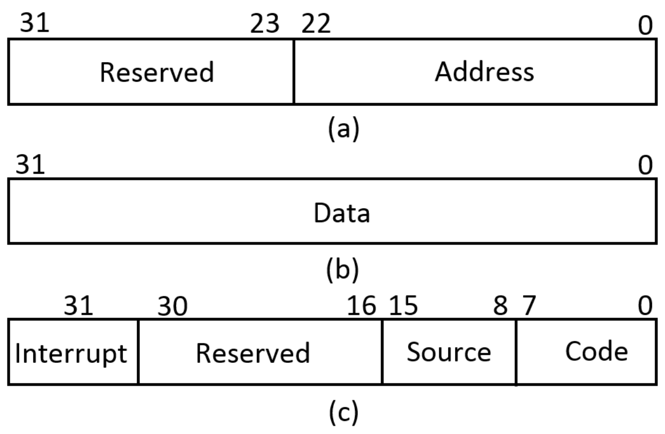


FIGURE 1.14: Mcause implementation

table 1.6, (B)), the “Address" field may represent different things depending on the specific source and code combination, for example, if the interruption comes from the MBC and represent a misaligned address, the "address" field will represent the offending address, while if the interruption comes from an IO it may represent a memory address that the IO is trying to read or write. Table 1.6 (A) lists the possible machine-level interrupt codes for the transactions coming from IO devices. Exception Code in this implementation do not follow the codes specified in [2]. Finally the “Data” field will contain information which could be interpreted indifferent ways by superior software implementation layers. In order to save space in this implementation all the “reserved” bits will be tied to zero.

TABLE 1.6: Possible sources and codes for interruptions coming from the IOs

| **Action** | **Code**$_h$ |
|---|---|
| Read 1 byte | 03 |
| Read 2 byte | 01 |
| Read 4 byte | 00 |
| Write 1 bytes | 13 |
| Write 2 bytes | 11 |
| Write 4 bytes | 10 |
| Boot_start(only for the SPI) | 14 |
| Boot_end(only for the SPI) | 18 |

(A) Operation Codes for interruptions coming from the IO

| **Device** | **ID**$_{hex}$ |
|---|---|
| CMB | 00 |
| SPI | 01 |
| UART | 02 |
| Reg_File | 06 |
| Inst_deco | 07 |
| IO_1 | 03 |
| IO_2 | 04 |
| IO_3 | 05 |

(B) Source IDs for interruptions

Besides the general IO interruptions described in the table 1.6 there are additional exceptions presented in table 1.7.

**CSR_IO Register:**

This register is used to indicate the nature of each one of the IO devices. As shown in figure 1.15 each bit from 0 to 5 in the register is associated to an IO device, if any of these bits is asserted this will indicate to the MBC that, in case of a read to an address associated to such IO device, the MBC will wait until the IO device respond with the information, and then respond to the cpu as if the information was read from memory. Depending on the IO specifics, and the traffic in the bus this may take

TABLE 1.7: Interruption and exceptions available

| **Source$_h$** | **Code$_h$** | **Address** | **Data** | **Description** |
|---|---|---|---|---|
| 00 | 80 | 24'd0 | 32'd0 | The timer in the MCB reached its limit |
| 00 | 01 | Address[23:0] | 32'd0 | non existent address |
| 00 | 02 | Address[23:0] | 32'd0 | address misaligned |
| 00 | 03 | 24'd0 | 32'd0 | Bus output FiFo is full |
| 00 | 04 | 24'd0 | 32'd0 | CSR base address overlaps with memory |
| 00 | 05 | 24'd0 | 32'd0 | Unfinished MMIO read (watch_dog) |
| 00 | 06 | 24'd0 | 32'd0 | Internal FiFo is full. |
| 06 | 00 | Inst address | Reg ID | access to inexistent GPR. |
| 06 | 01 | Inst address | CSR ID | access to inexistent CSR. |
| 07 | 00 | Inst address | 32'd0 | Instruction not found |
| 07 | 01 | 24'd0 | 32'd0 | Ebreak |
| 07 | 02 | 24'd0 | 32'd0 | Ecall |

quite some time with the processor waiting for the read operation to finish. If the CSR_IO bit associated with an IO is not asserted, and a read operation to an address associated to such IO occur, then the MBC will send the request to the IO to the bus, but instead of keep the cpu waiting for an answer it will just respond as if the data requested was all zeros, and when the answer from the IO device arrives in the bus, it will be handled using an interruption; this way the CPU will be free to execute useful code while waiting. The CSR ID assigned to this register is 7.

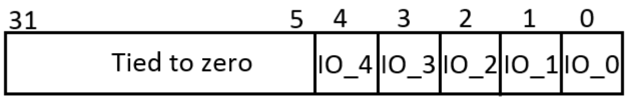


FIGURE 1.15: Illustration of the csr_IO register

**Machine Time Registers (watchdogcmp, mtime and mtimecmp)**
There are only three memory mapped registers in this implementation, all related to a timers. Mtime and Mtime cmp registers are intended to provide a real-time counter, exposed as a memory-mapped machine-mode register, mtime. mtime run at constant frequency (the main clock frequency). The mtime register has a 64-bit precision in order to comply with RISCV specification [2]. Also there is a 64-bit memory-mapped machine-mode timer compare register (mtimecmp), which causes a timer interrupt to be posted when the mtime register contains a value greater than or equal to the value in the mtimecmp register. Both mtimecmp and mtime registers are mapped as little endian; mtimecmp is in the address immediately after the main memory finish, and the mtime register (read only) is mapped immediately after the mtimecmp. In this implementation the mtime registers will be reset when the MTIE bit in the Mie register is not asserted, and will automatically reset after it is equal to the mtimecmp register and send an interruption. these registers can be used to generate timing interruptions.
The watchdogcmp register is a 32 bits memory mapped register located next to the mtime register. This is used to set a watchdog timer that will generate an interruption if the time waiting for a data coming from an IO read (in the case where its bit in the CSR_IO register is set) takes too much. If this watch dog is reached then the

MCB will respond with zeros to the read request from the CPU.

TABLE 1.8: Default CSR values after reset.

| Register | Default value |
|---|---|
| mie | 0x00000800 |
| mip | 0x00000000 |
| mepc | 0x00000000 |
| mcause | 0x00000000 |
| trapA | 0x80000000 |
| trapB | 0x00000000 |
| mvtec | 0x00000000 |
| CSR_IO | 0x00000000 |

### Default values of the CSRs after reset

Table 1.8 shows the default value of the registers implemented in the Register file after reset. The MISA register doesn't have a reset because it is hardwired to a constant. The reset signal doesn't have effect in counter register because they use the MTIE bit in the MIE register as a reset.

TABLE 1.9: Functions accepted by the ALU. The operands in bold are unsigned

| ALU_cntrl | Function ID | OUT | Zero |
|---|---|---|---|
| 0000 | EQU | 0 | A==B ? 1:0 |
| 0001 | LESS_THAN | 0 | $A < B$ ? 1:0 |
| 0010 | LESS_THAN UNSIGNED | 0 | $\mathbf{A} < \mathbf{B}$ ? 1:0 |
| 0011 | GREATER_THAN | 0 | $A > B$ ? 1:0 |
| 0100 | GREATER_THAN | 0 | $\mathbf{A} > \mathbf{B}$ ? 1:0 |
| 0101 | ADD | A+B | 0 |
| 0110 | ADD UNSIGNED | **A** + **B** | 0 |
| 0111 | SUB UNSIGNED | **A** - **B** | 0 |
| 1000 | SHIFT LEFT LOGICAL | $A << B$ bits filled 0s | 0 |
| 1001 | SHIFT RIGTH LOGICAL | $A >> B$ bits filled 0s | 0 |
| 1010 | SHIFT RIGTH ARITMETIC | $A >> B$ bits sign ext | 0 |
| 1011 | BITWISE OR | A or B | 0 |
| 1100 | BITWISE XOR | A xor B | 0 |
| 1101 | BITWISE AND | A and B | 0 |

### 1.2.4 Arithmetic and Logic Unit (ALU)

This is a combinational block intended to execute arithmetic and logic computations, its basic implementation is shown in figure1.16 and its accepted functions are presented in table 1.9

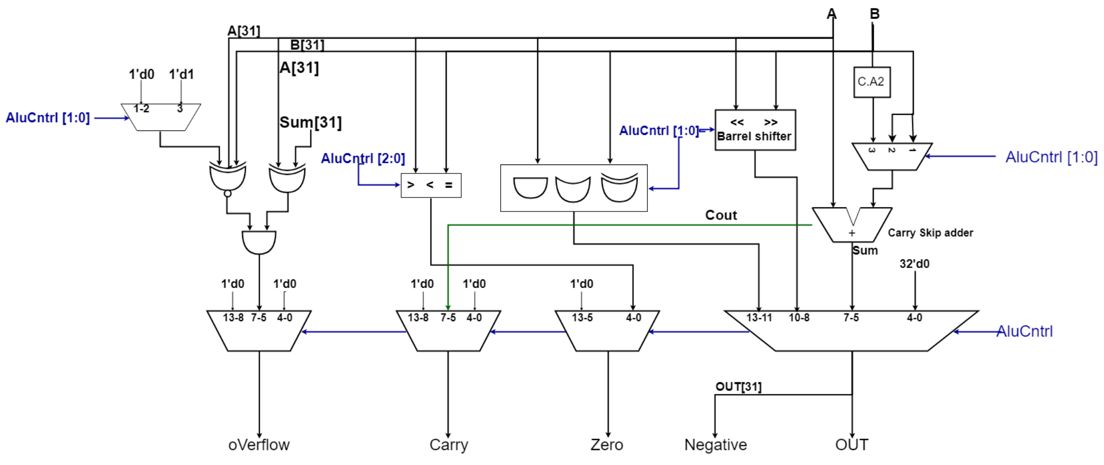
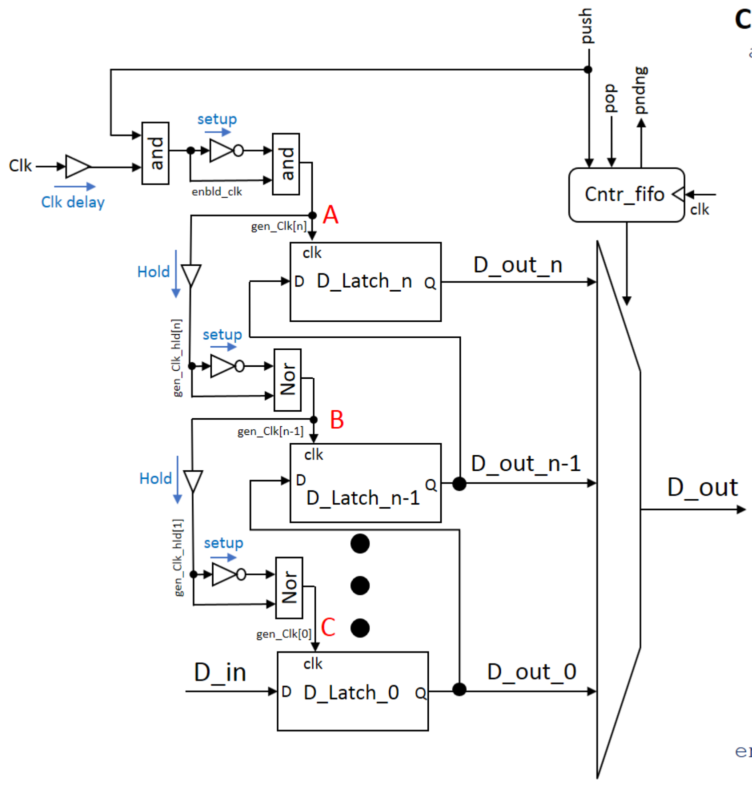


FIGURE 1.16: Basic implementation of the ALU module

### 1.2.5 Synchronous First In First Out Queue (FIFO)

Even when the FIFO structures are not a micro-architectural block of the processor by themselves, they are a central design block in many units of the system on chip and inside the cpu, for example, in the CPU, the bus, the interrupt controller and the memory and bus controller, and the UART, in the IOs. That is the reason why is important to give an introduction to the architecture used in the FIFOs.
Since one of the biggest challenges in this design is the area efficiency, and the FIFOs are blocks instantiated many times, an architecture based on latches instead of flops is proposed. Also, instead of using two pointers in the FIFO architecture, which would imply two counters and some additional synchronization logic, an implementation using only one counter is proposed.
The implemented FIFO from the user perspective is very standard. There are two signals intended to "Push" and "Pop" data which work at rising edge, the input and output data goes through two buses: "D_push" and "D_pop". Ones the data is captured by the receiving entity, a "Pop" signal must be asserted during one clock cycle; this will allow the next data in the FIFO to go to the "D_pop" bus. If the FIFO gets full a "Full" flag will be asserted and in the case of new data being pushed, then the FIFO will evict the oldest data in the queue.

In figure 1.17 the proposed architecture is presented, it is based on edge detectors, which will generate a cascade of pulses in the clocks of the different levels of latches in the FIFO structure in order to save the data, starting at the top level of the FIFO. Since the data is pushed in the base of the queue this architecture require only one counter and no additional logic; also the use of latches instead of flops reduces the area of the implementation. The additional complexity that this structure requires compared with traditional implementations using two counters and flops, is the careful setting of the delays marked blue in figure 1.17. An illustrative timing diagram for these delays is presented in figure 1.18, three delays are important to notice: the first one is the one highlighted in green and is related to the clock, since the Push signal is assumed to be generated using the rising edge of the clock in a state machine, there should be some delay in the clock in order to ensure that there will not be an overlap between the clock and the push signal at the end of the push cycle that may cause a glitch in the signals named as "gen_clk" in figure 1.17, also labeled as "A, B and C" in figures 1.17 and 1.18. The second sensitive time is the one



**Cntr_fifo:**

```
 always@(posedge clk)begin
  if(rst) begin
    count <= 0;
  end else begin

    case({push,pop})
      2'b00: count <= count;
      2'b01: begin
        if(count == 0) begin
          count <= 0;
        end else begin
          count <=count - 1;
        end
      end
      2'b10:begin
         if(count == depth)begin
           count <= count;
         end else begin
           count <= count+1;
        end
      end
      2'b11: count <= count;
    endcase
  end
  pndng <= (count==0)?{1'b0}:{1'b1};
  full <=(count == depth)?{1'b1}:{1'b0};
end
```

FIGURE 1.17: Synchronous FIFO architecture proposed

labeled as “setup" which should be bigger that the setup time of the latches, and the third one is the one labeled as “hold" which should be bigger than the hold time of the latch

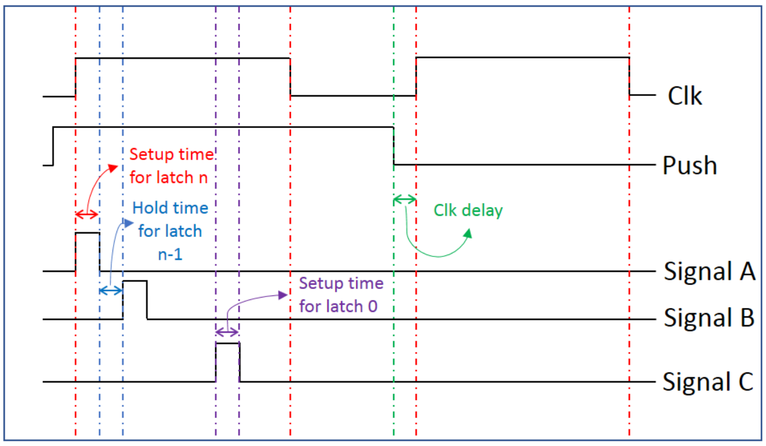


FIGURE 1.18: Illustrative timing diagram for the basic FIFO implementation presented in figure 1.17

### 1.2.6 Memory and bus controller Specification (MBC

The memory and bus controller or MBC will manage the access to memory and IO devices using a memory mapped scheme. In order to save hardware the memory map is parametrized at compilation time (is hard-coded for each implementation) The memory map used in this implementation ins shown in figure 1.19.

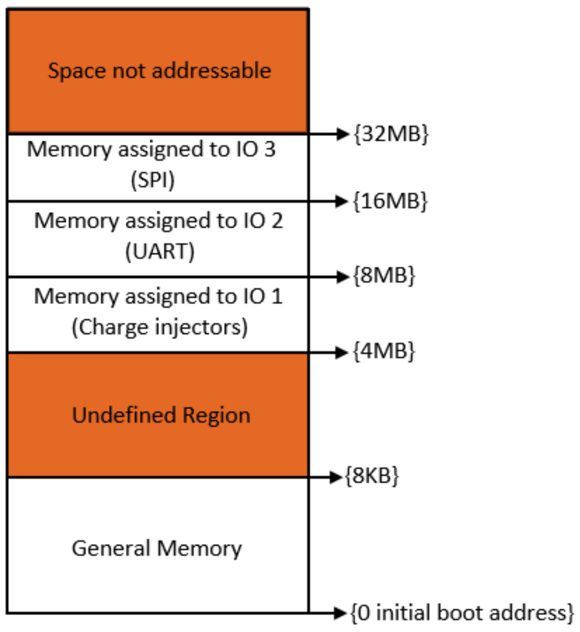


FIGURE 1.19: Illustration of the memory map

In Figure 1.20 all the signals from and to the control unit, processor, interruption handler, register file and bus interface are shown.

**Input and output description:**

1. Signals in the bus interface:
   - **Pndng:** one-bit signal that is asserted if there is data at the bus waiting to be processed.
   - **D_pop:** data coming from the IO, that are waiting in the bus. Initially, it is 10 Bytes long, but can be changed because it is parameterized.
   - **Pop_MCB:** one-bit signal that tells the bus that the data has already been received and can be deleted from it.
   - **D_push:** data to be send to the IO using the bus.
   - **Push:** one-bit signal that tells the bus to take the information from D_push.
2. Signals in the control interface:
   - **clk:** main system clock. This clock should be enabled by the control unit.
   - **Mem_rdy:** this signal will be asserted when the requested write or read is performed and the data is ready in the corresponding bus. It will last one clock cycle asserted.

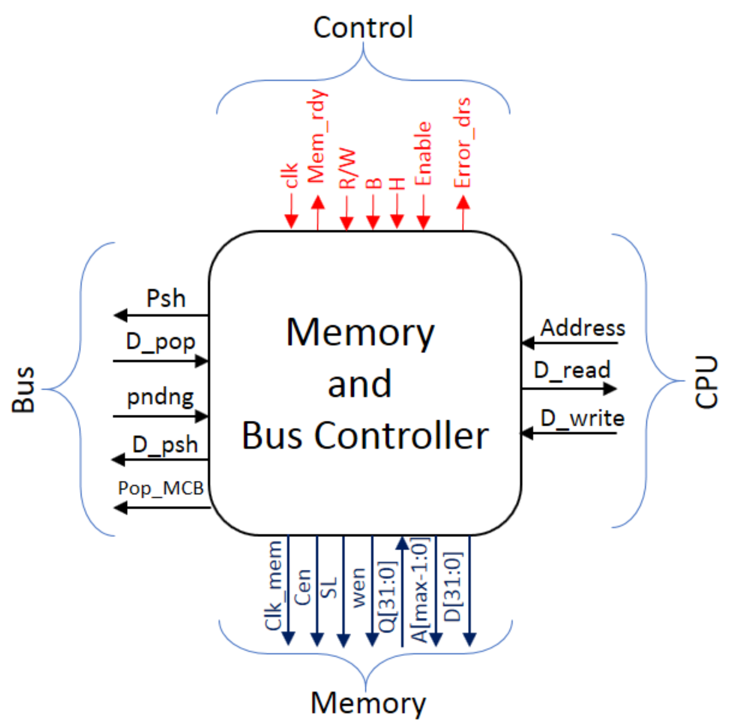


FIGURE 1.20: Memory and bus controller input and output diagram

- **R/W:** one-bit input. High means write, low means read.
- **B:** one-bit input that, when asserted, implies the read or write of only one byte at the address pointed in the address input. This signal has precedence over the H input.
- **H:** one-bit input when asserted imply the read or write of only 2 bytes starting at the address pointed in the address input sorted as Little Endian. This requires B to be low. When both B and H are low each read and write will be done in 4 bytes.
- **Enable:** enables the read or write of the data from the processor.

3. Signals in the CPU interface:

   - **Address:** 25 bits bus points to the base address at which the read or write should take place.
   - **Data_Write:** 32 bits input bus receives the data which should be written.
   - **Data_Read:** 32 bits output bus gives the data read from memory or IO.

4. Signals in the interface of the memory:

   - **Clk_mem:** This is the clock of the memory, basically this is the neation of the system clock with the cen signal as enable.
   - **Cen:** This signal is the chip enable, and is active in cero.
   - **SL:** This signal is active in high and is used to set the memory in sleep mode; in this implementation this signal is tied to 0.
   - **Wen:** this signal is used to define if the memory is being read (1) or written (0).

- **Q:** This is the data coming from the memory.
- **A:** These are the address lines to the memory.
- **D:** These are the data lines going to the memory

In figures 1.21 1.22 the timing diagrams for the read and write operations in memory are presented

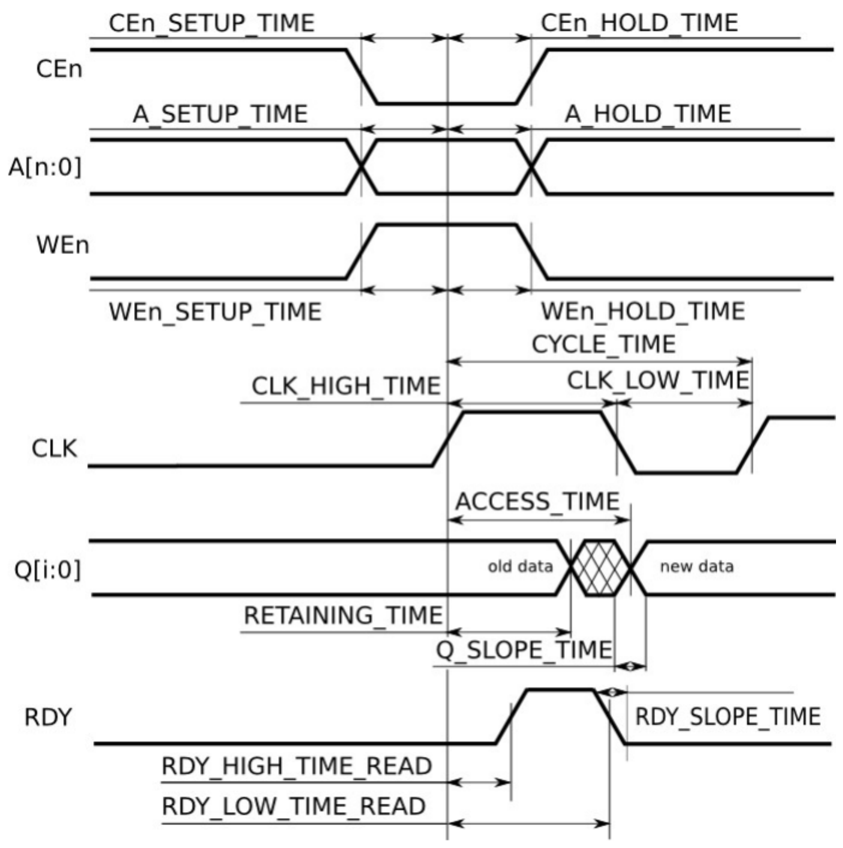


FIGURE 1.21: Read cycle timing diagram

The general functions of the memory and bus controller (CMB) can be characterized as a group of different events:

1. **Boot load process**: This is the first process executed by the MBC, here the code required by the CPU to start working is fetched from an external memory connected to the SPI interface. The process can be described as a basic succession of steps.

    - After reset, the first action of the CMB is to send a package though the bus interface to the SPI with the structure shown in Figure **??** Here, the *destination* field is set to the ID of the SPI, the *source* field is the ID of the CMB, the *code* field correspond to the boot_start code and finally the *Address* field and the *Data* field are not used in this package, so they are set to zero in order to save some power. This package inform the SPI that the MBC is ready to start the boot load process. The codes used for the ID of the devices and the actions that can be performed in the communication t the IO devices are presented in Table 1.10.

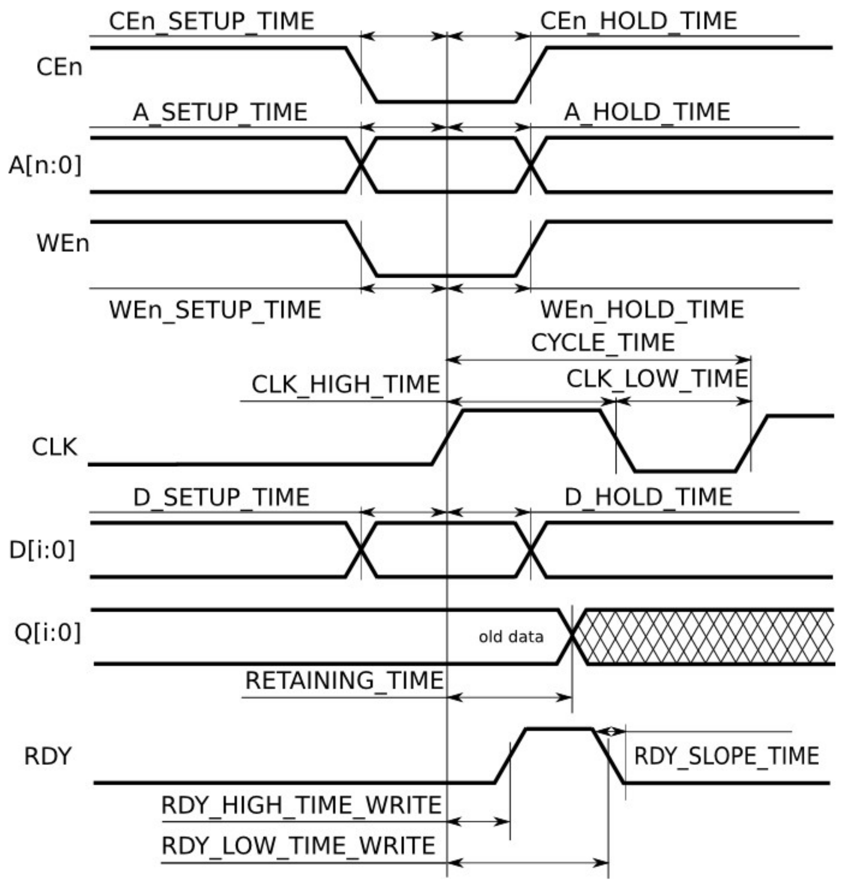


FIGURE 1.22: Write cycle timing diagram

TABLE 1.10: Codes for the data package

| **Action** | **Code**$_{hex}$ $\{\overline{R/W}$,H,B$\}$ |
|---|---|
| Read 1 byte | 1 |
| Read 2 byte | 2 |
| Read 4 byte | 0 |
| Write 1 byte | 5 |
| Write 2 byte | 6 |
| Write 4 byte | 4 |
| Boot_start | 7 |
| Boot_end | 3 |

(A) Operation codes

| **IO** | **Code**$_{hex}$ |
|---|---|
| CMB | 0 |
| SPI | 1 |
| UART | 2 |
| IO_1 | 3 |
| ⋮ | ⋮ |

(B) IO identification

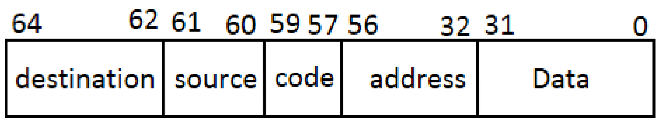


FIGURE 1.23: Format of the bus package

- When the SPI interface receive the start package send by the MBC then the boot code is sent to the MCB in packages with the same structure shown in Figure **??** with the *destination* field filled with code for the CMB ID, *source* field willed with the SPI ID, the *code* field containing any of the codes for write (see table 1.10a), the *Address* field with the address that corresponds to the target RAM address for the data and the *Data* field containing the source code that is going to be written. The data coming from the SPI interface can be discarded by the MBC if:
  - The *Address* doesn't corresponds to a valid RAM address.
  - The *Code* is not 0x10, 0x11 or 0x13, so it isn't a write, and the boot process can only write to the memory.
  - The data is misaligned.
- When the boot code is written into the RAM, the SPI send an specific data package to inform the MBC that the boot process has finished. This package is structured as shown in Figure **??** and filled as follows: the *destination* field contain the CMB ID, the *source* field contain the SPI ID, the *code* field contain the the boot_end code at Table 1.10a,and finally the *address* field and the *data* field are irrelevant for this package. After the boot process finish the MBC will generate a signal to allow the CPU to take control of the system and will start its normal operation.

2. **Data packages arriving from the Bus interface**:

   (a) If the bus has information from an IO, the flag *Pndng* is asserted, so the CMB takes it. If ***MEIE*** is high the information is processed, else it will be deleted by raising the ***Pop*** signal up.

   (b) If **MEIE** is asserted then the information will be processed, which means to place the *source* plus the *payload* marked at Figure **??** on the output called ***IO_Intr_D_push***, and push this information into the interrupt controller. Notice that in this process the segmentation of the *payload* is of no importance for the controller, it's treated just as pure information and not as the fields *data*, *address* and *code*.

3. **Processor write request**:

   - When ***Enable*** is asserted the data from ***Data_write***, ***Address***, ***B***, ***H*** are saved and processed. The signal ***RW*** must be asserted for a write request.
   - The alignment of the *address* will be verified as presented in the Table 1.11.
   - The *address* is decoded to know if the processor wants to write on memory, or IOs. Memory mapping is done as shown in Figure 1.19
   - There are four options for the *address*:

TABLE 1.11: Alignment decoding

| | Address[1:0] | BH |
|---|---|---|
| Aligned | 00 | 00 |
| Misaligned | 01 | 00 |
| Misaligned | 10 | 00 |
| Misaligned | 11 | 00 |
| Aligned | 00 | 01 |
| Misaligned | 01 | 01 |
| Aligned | 10 | 01 |
| Misaligned | 11 | 01 |
| Aligned | 00 | 1X |
| Aligned | 01 | 1X |
| Aligned | 10 | 1X |
| Aligned | 11 | 1X |

(a) It corresponds to memory: if the write is not for the whole word, first the address is read and then the modified data is written in that address; if the write is for the whole word then the write is performed directly to memory.

(b) It corresponds to an IO: in this case the MBC place the write request in the bus.

(c) It corresponds to a time CSR: in this case the MBC will write the data in the Memory mapped CSR.

(d) It does not exists: an interruption is sent. In Table 1.12 are shown the interruption codes that will be generated in the controller.

TABLE 1.12: Interruption codes

| **Source** | **Code$_h$** | **Address** | **Data** | **Description** |
|---|---|---|---|---|
| 00 | 80 | 24'd0 | 32'd0 | The Timer reached the limit |
| 00 | 01 | Address[23:0] | 32'd0 | The address does not exists |
| 00 | 02 | Address[23:0] | 32'd0 | The address is misaligned |
| 00 | 03 | 24'd0 | 32'd0 | External FIFO is full |
| 00 | 04 | 24'd0 | 32'd0 | The first CSR limit is lower than the established memory limit. |
| 00 | 05 | 24'd0 | 32'd0 | The read is ended abruptly (watch_dog) |
| 00 | 06 | 24'd0 | 32'd0 | Internal FIFO is full. |

- When the case ends the flag ***Mem_rdy*** will be raised.

4. **Processor read request**:

- The processing is done as in the previous process, until *address* is decoded. And again there are five options:

(a) It corresponds to an MMIO: a read is sent to the IO through the bus, the count to the Wtch_dog starts and the controller will wait until:

i. The information arrives from the IO, so it is placed in ***Data_read***.

ii. The Watch_Dog counter arrives to it's limit, the ***Watch_dog*** register, so the correspondent interruption is sent to the interruption handler and the output ***Data_read*** is set to zero.

(b) It corresponds to an PMIO: the data and the coded address is sent to the correspondent IO and the output ***Data_read*** is set to zero.
(c) It corresponds to a time CSR: the addressed register is read and the information in it is placed in ***Data_read***.
(d) It corresponds to memory: the data in the addressed memory space is read and the requested bytes are set in ***Data_read***.
(e) The address is not valid: an interruption is sent with the established code and the *address*.

- Then the ***Mem_rdy*** flag is asserted to indicate that the information was processed.

At Table 1.13 it is a summary of the results of the read or write done.

TABLE 1.13: Operation of memory readings and writings, by the core.

| RW | B | H | Address[1:0] | Data_write | Data in memory |
|---|---|---|---|---|---|
| 1 | 1 | X | 00 | XXXXXXDD | EEFF11DD |
| 1 | 1 | X | 01 | XXXXXXDD | EEFFDD22 |
| 1 | 1 | X | 10 | XXXXXXDD | EEDD1122 |
| 1 | 1 | X | 11 | XXXXXXDD | DDFF1122 |
| 1 | 0 | 1 | 00 | XXXXCCDD | EEFFCCDD |
| 1 | 0 | 1 | 01 | — | EEFF1122 |
| 1 | 0 | 1 | 10 | XXXXCCDD | CCDD1122 |
| 1 | 0 | 1 | 11 | — | EEFF0011 |
| 1 | 0 | 0 | 00 | AABBCCDD | AABBCCDD |
| 1 | 0 | 0 | 01 | — | EEFF1122 |
| 1 | 0 | 0 | 10 | — | EEFF1122 |
| 1 | 0 | 0 | 11 | — | EEFF1122 |

| RW | B | H | Address[1:0] | Data in memory | Data_read |
|---|---|---|---|---|---|
| 0 | 1 | X | 00 | EEFF1122 | 00000022 |
| 0 | 1 | X | 01 | EEFF1122 | 00000011 |
| 0 | 1 | X | 10 | EEFF1122 | 000000FF |
| 0 | 1 | X | 11 | EEFF1122 | 000000EE |
| 0 | 0 | 1 | 00 | EEFF1122 | 00001122 |
| 0 | 0 | 1 | 01 | EEFF1122 | 00000000 |
| 0 | 0 | 1 | 10 | EEFF1122 | 0000EEFF |
| 0 | 0 | 1 | 11 | EEFF1122 | 00000000 |
| 0 | 0 | 0 | 00 | EEFF1122 | EEFF1122 |
| 0 | 0 | 0 | 01 | EEFF1122 | 00000000 |
| 0 | 0 | 0 | 10 | EEFF1122 | 00000000 |
| 0 | 0 | 0 | 11 | EEFF1122 | 00000000 |

5. **Time CSRs**: they are used as a counter, and are organized as follows:

(a) The first two registers are the limits (*mtimecmp*) and, using the Figure 1.2, they are addressed by the address 0x2000 and 0x2004. In this registers the only option is to write all the 4 bytes, which means that all the data in ***Data_write*** will be written on it.
(b) The next two registers, 0x2008 and 0x200c, (*mtime*) are the ones that count when this count is enable. They can be read and written, but it is useless

to write in them because when the count is enabled they are going to be reseted to zero.

When the count is enabled by ***MTIE*** the *mtime* registers are set to zero and they will increase till they arrive to the limit, *mtimecmp*; at that moment an interruption will be sent. If ***MTIE*** is not set to zero after this, the counter will be reset and it will count again and again generating the interruptions, till the ***MTIE*** will be set to zero. When the ***MTIE*** is in zero the counter is stopped and the *mtime* register are set to zero.

# Bibliography

[1] A Waterman and K Asanovic, "The risc-v instruction set manual-volume i: User-level isa-document version 2.2", *RISC-V Foundation (May 2017)*, 2017.

[2] A. Waterman, "Risc-v privileged architecture", 2017.

[3] D. Patterson and A. Waterman, *The risc-v reader: An open architecture atlas*. Strawberry Canyon, 2017, ISBN: 099924910X, 9780999249109.