Cybersecurity National Lab

CyberChallenge

MC2101: AFTAB-based microcontroller
User Manual

Juan José Restrepo

November 29, 2023

Contents

CHAPTER 1

Introduction

The MC2101 is a microcontroller that is a synthesizable embedded system entirely described in VHDL language, meant to be used as a reliable platform on which it is possible to run real applications, as well as integrate and evaluate security solutions for IoT in a realistic environment. Furthermore, the platform can be used for training activities in the cybersecurity domain, e.g., for modeling specific hardware security issues in Capture-the-Flag exercises, where vulnerabilities are intentionally inserted with the aim of being exploited and/or mitigated.

MC2101 microcontroller integrates an AFTAB core based on RISC-V with a proper set of peripherals necessary to provide all basic I/O functionalities for running software. In particular, the peripherals are a GPIO module for handling input and output digital signals and a UART module used to allow serial communication between a computer and the microcontroller itself.

The development of the microcontroller also included a software design part. In particular, all system libraries used for driving and configuring the peripherals were written, and also interrupt service routines have been included in the processor's bootloaders.

In this way, the next chapter introduces the AFTAB microprocessor and its main features. The last chapter introduces the MC2101 microcontroller and its software libraries.

CHAPTER 2

AFTAB

2.1 General Description

The AFTAB processor is based on RISC-V. RISC-V is an open-source instruction set architecture(ISA) that provides a foundation for processor design. It is based on RISC and offers a modular and extensible ISA that can be customized for specific applications and use cases. The base integer ISAs are very similar to that of the early RISC processors, except for the branch delay slots and the support of optional variable-length instruction encodings.

Some advantages of RISC-V include its open-source nature which allows for greater customization and flexibility in processor design. In particular, RISC-V groups 4 base ISAs. Each one of them is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers:

1. RV32I and RV64I are the integer variants that provide 32-bit or 64-bit address spaces respectively
2. RV32E subset variant of the RV32I base instruction set which is added to support small microcontrollers and has half of the number of integer registers
3. RV128I variant of the base integer instruction set supports a flat 128-bit address space

Moreover, a set of standard extensions is defined to provide integer multiply/divide, atomic operations, and single/double-precision floating-point arithmetic:

1. **The base integer ISA** is named "I" and contains integer computational instructions, integer loads, integer stores, and control-flow instructions.
2. **Integer multiplication and division extension** is named "M", and adds instructions to multiply and divide values held in the integers registers
3. **Standard atomic instruction extension** is denoted by "A", and adds instructions that atomically read, modify and write memory for inter-processor synchronization
4. **Standard single-precision floating-point extension** is named "F", and adds floating-point registers, single-precision computational instructions and single-precision load and stores
5. **Standard double-precision floating-point extension** is denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores
6. **Standard compressed instruction extension** is denoted by "C", provides narrower 16-bit forms of common instructions

The AFTAB processor is based on RISC-V and it implements the RV32I instruction set and the RV32M extension. In addition, AFTAB includes the *Zicsr* which is associated with Control and Status Registers.

2.2 Instruction Set Architecture

An Instruction Set Architecture(ISA) is part of the abstract model of a computer that defines how the CPU is controlled by software. It also acts as an interface between the hardware and the software, specifying what the processor is capable of. AFTAB has an ISA that supports 52 instructions encoded in 32 bits, where the 7 least significant bits are related to the *opcode*. The instructions are grouped in 4 different formats:

- **R-type:** instructions using 3 register inputs
- **I-type:** instruction with immediates, loads
- **S-type:** store instructions
- **B-type:** branch instructions
- **U-type:** instructions with upper immediates
- **J-type:** jump instructions

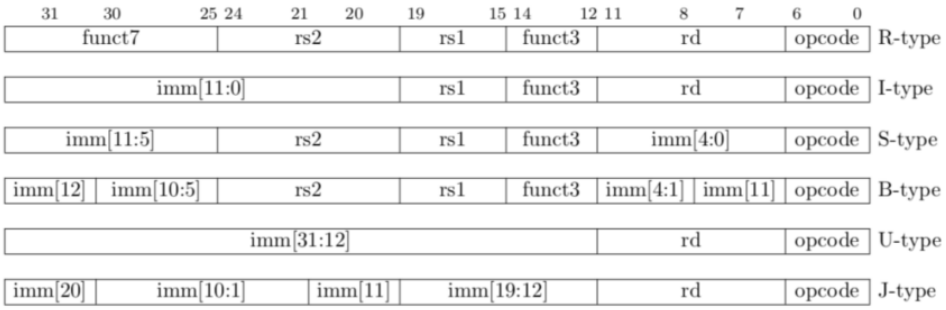

Figure 2.1: RISC-V base instruction format showing immediate variants

Figure 2.1 shows the format of every type of instruction. The following explains in more detail the purpose of each field in the formats:

R-type:

- *opcode:* partially identifies the instruction
- *func7 and func3:* describe what the operation to perform is
- *rs1:* first operand - source register 1
- *rs2:* second operand - source register 2
- *rd:* destination register. It receives the result of the computation

I-type:

- *opcode:* identifies the instruction
- *rs1:* specifies a register operand
- *rd:* destination register. It receives the result of the computation
- *func3:* describe what the operation to perform is
- *imm[11:0]:* 12-bit number. It has to be extended to 32 bits because all the operations are done in words.

S-type:

- *opcode:* identifies the instruction
- *rs1:* register whose value is used as a base memory address
- *rs2:* register that contains the data to be stored in memory
- *func3:* describe what the operation to perform is
- *imm[11:5] and imm[4:0]: imm[11:5]* is concatenated with *imm[4:0]* and this value corresponds to the offset value that is added to *rs1*. The result of this addition is the address to access the memory.

B-type:

- *opcode:* identifies the instruction
- *rs1:* source register 1
- *rs2:* source register 2
- *func3:* describe what the operation to perform is
- *imm[12], imm[10:5], imm[4:1] and imm[11]:* 12-bit immediate composed of the concatenation of imm[12] & imm[11] & imm[10:5] & imm[4:1]

This format is used for branching instructions. There could be two cases:

- If the branch is taken: PC = PC + 4
- If the branch is not taken: PC = PC + (immediate * 4)

One important aspect to consider is that the branch actually uses Relative addressing of 13-bit immediate. However, the LSB is always assumed to be 0 with the purpose of getting a proper alignment. One can notice that by forcing the LSB to 0, the address is constrained to be a multiple of 2 instead of 4 as it is normally considered. This is because the RISC-V also has a 64-bit version and a proper encoding for both cases was needed:

- For RV32I, if Imm[1] !=0, aligment exception
- For RV64I, if Imm[2:1] !=0, aligment exception

AFTAB works with the RV32I version, hence it only forces the LSB to be 0, obtaining only a 12-bit immediate operand.

U-type:

- *opcode:* identifies the instruction
- *rd:* destination register
- *imm[31:12]* 20-bit immediate

The purpose of this format is to deal with 32-bit immediates which are not supported by the *I-type* format. I-type only supports 12-bit operands and there is a need to consider the other 20 bits. Thus, an I-type instruction can be used together with a U-type instruction to consider 32-bit immediates.

J-type:

- *opcode:* identifies the instruction
- *rd:* destination register. In the case of a *j*(jump), the assembler automatically sets rd = x0 to discard return address
- *imm[20], imm[10:1], imm[11] and imm[19:12]:* 21-bit immediate composed of the concatenation of imm[20] & imm[10:1] & imm[11] & imm[19:12]. However, the LSB of that concatenation should be set to 0. This value corresponds to the target address.

The jump actually uses Relative addressing of 21-bit immediate. However, the LSB is always assumed to be 0 with the purpose of getting a proper alignment. One can notice that by forcing the LSB to 0, the address is constrained to be a multiple of 2 instead of 4 as it is normally considered.

On another matter, as it was said before, the AFTAB processor implements the RV32I instruction set, the RV32M extension and *Zicsr* which is associated with Control and Status Registers. The following exposes more details about them:

- **RV32I:** Essential base for a compiler target and for the execution of modern operating systems. The internal state of base integer ISA is given by 32 unprivileged general-purpose registers and the privileged Program Counter (PC).
- **RV32M:** 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 × quoent + remainder*

- ***Zicsr*:** Extension that defines Control and Status registers (CSRs) Instructions. RISC-V defines a separate addressing for 4096 CSRs on which it can operate. Figure 2.2 shows the control and status registers of AFTAB.

  On the one hand, a control register is a processor register that changes/controls the general behavior of a CPU. They are commonly associated with interrupt controls, switching the addressing mode, coprocessor control, etc. On the other hand, a status register consists of a collection of status flag bits for a processor whose purpose is to keep track of the state of the processor. In this case, individual bits are implicitly or explicitly read and/or written by the machine code instructions which are executed on the processor.

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

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

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

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

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

Figure 2.2: Control and Status registers of AFTAB

AFTAB datapath provides a set of combinational and sequential units, customized to perform the supported instructions.

In each state of the controller, a set of proper control signals are set to activate the appropriate units and paths in the datapath. After the reset, the execution states are repeated until exceptions and interrupts. The overall behavior of AFTAB is summarized in the following set of states:

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

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

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

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

2.3 Privilege levels

A privilege level is an attribute of the software execution. It is 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.

The machine level has the highest privilege and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User mode (U-mode) and supervisor mode (S-mode) are intended for conventional application and operating system usage respectively. Figure 2.3 exposes the possible combinations of privilege modes.

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

Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports an optional standard extension for memory protection.

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code.

Particularly, AFTAB is intended to be used as a Secure embedded system, as it implements M-mode and U-mode.

2.4 Programmer's model

When writing a program for AFTAB, a programmer may choose to provide both an Assembly source file (.s) or a C/C++ source file. Since the address bus is 32-bit, the **.text** section for instructions and the **.data** section for variables can reach up to 4 GB. Actually, to allow fast simulation even without

resorting to professional licenses for simulator or server equipment, the addressing space has been truncated to 8 KB.

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

In the RISC-V standard register convention, there are the *saved registers* **s0** to **s11**, that are used to store data that must be preserved across function calls. Then, the *argument registers* **a0** to **a7** are used to pass arguments, and the *temporary registers* **t0** to **t6** are used inside a function for computation. The convention also defines specialized registers such as the *Stack Pointer* **sp**, the *Global Pointer* **gp**, the *Thread Pointer* **tp** and *Link Register* **ra**. Figure 2.4 shows the register map of the AFTAB processor.

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_

Figure 2.4: AFTAB general register map

The *Stack Pointer* indicates the location of the last item put onto the stack, the *Global pointer* enables fast access to global data in the AFTAB, the *Thread Pointer* is designed for thread-local data and the *Link Register* is used for holding the return address when calling a function or subroutine.

2.5 Memory Map

As a 32-bit RISC-V implementation, AFTAB has a virtual byte-addressable address space of 4 GB for all memory accesses. A word of memory is defined as 32 bits (4 bytes). Correspondingly, a half-word

is 16 bits (2 bytes), a double-word is 64 bits (8 bytes), and a quad-word is 128 bits (16 bytes).

An important thing to consider is that the physical address space has been limited to 8 KB. Moreover, the memory unit provided by this project automatically manages the conversion from virtual to physical.

AFTAB simulation environment currently uses the memory map in Figure 2.5 from which one can observe:

- **Instruction Memory:** 4 KB RAM with base address 0x00000000.
- **Data Memory:** 1.5 KB RAM with base address 0x00100000.
- **Stack:** 2.5 KB RAM with base address 0x00100600.
- **Peripherals:** The original AFTAB simulation environment di not include any peripherals and memory addresses. However, the MC2101 considers two peripherals: UART and GPIO(see chapter 3)

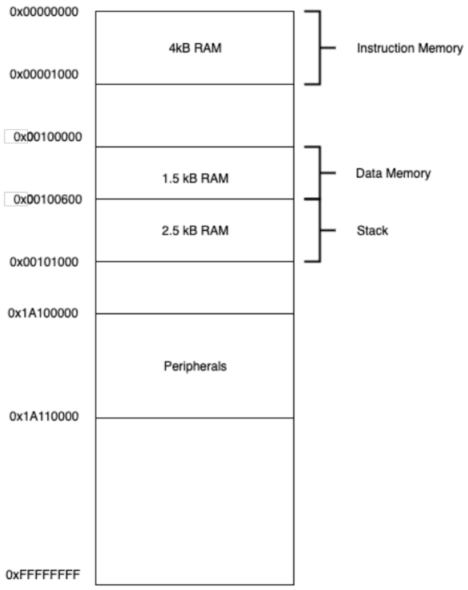

Figure 2.5: AFTAB Memory map

2.6 General description of Datapath and Control Unit

AFTAB is a 32-bit microprocessor with a multiple-cycle architecture which means that the states are performed serially during the execution of instructions. Thus, there is no pipelining. Figure 2.6 exhibits the pinout of the AFTAB microprocessor.

The following describes the signals shown in the Figure:

Memory interface

- **memReady:** Ready signal for memory. Its purpose is to stall the processor in the fetch stage. By doing so, the bus infrastructure can spend the proper time required to complete the read-and-write operations

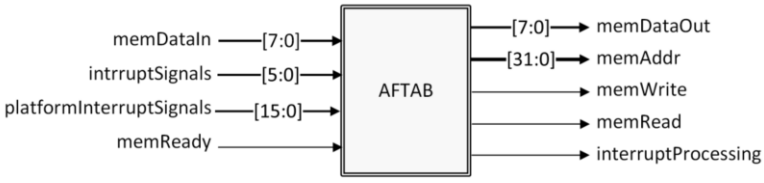

Figure 2.6: Pinout of the AFTAB microprocessor

- **memRead:** Read signal for memory. This signal is forwarded to the MC2101 bus to start a transaction that can target the main memory or any peripheral.
- **memWrite:** Write signal for memory. This signal is also forwarded to the MC2101 to start a transaction that can target the main memory or any peripheral.
- **memAddr:** 32-bit address bus for addressing the memory
- **memDataIn, memDataOut:** Two 8-bit ports for data input and data output from/to memory. During load-and-store operations, they carry a variable number of bytes(B, H, W) per transaction such that only one byte can be written or read per clock cycle

Interrupt interface

- **interruptSignals:** A set of 6 single-bit interrupt sources
- **platformInterruptSignals:** 16 interrupt lines for platform use
- **interruptProcessing:** Indicates if the processor is in the interrupt processing states

As for the Interrupt Interface set, there are a total of 22 independent interrupt lines, where 16 of them are for platform use, plus the additional signal *interruptProcessing*. In addition, the core is also able to handle the following hardware-level exceptions:

- **Illegal Instruction:** the decode phase does not recognize the opcode as a valid one
- **Illegal CSR instruction:** raised when there is an attempt to read or write a CSR register with an inappropriate privilege level, or when trying to access a non-existing CSR register
- **Instruction address misaligned:** raised after an attempt to access the instruction memory without the proper 4-byte alignment.

On another matter, AFTAB works with Von Neumann convention. Therefore, there is only one memory that contains both data and instructions and follows the little-endian convention(the least significant byte is stored in the smallest address).

Figure 2.7 shows an abstraction of the datapath and controller of the AFTAB.

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

The control word of the AFTAB has 83 control signals in total, with 78 single-bit signals and 5 multi-bit signals. When IR loads a new instruction, the *opcode* (IR[6:0]), the *funct3* (IR[14:12]) and the *funct7* (IR[31:25]) fields are read by the CU to compute the control word. The CU is internally

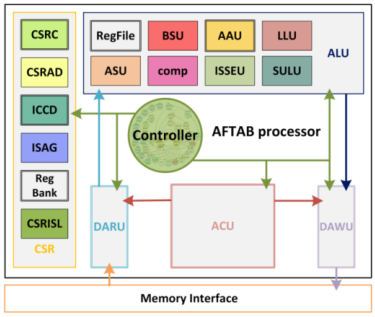

Figure 2.7: High-level abstraction of the datapath and controller

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 units. The Control Unit is a completely synchronous block, so the internal state is updated every clock cycle.

NOTE: For further documentation related to the AFTAB processor, please refer to the AFTAB GitHub repository

CHAPTER 3

MC2101

3.1 Architecture

3.1.1 General description

The MC2101 is a microcontroller system based on AFTAB. At the current state, the microcontroller is composed of a minimal set of peripherals properly selected for providing all necessary input/output functionalities useful for interacting with the external world, once synthesized on a FPGA.

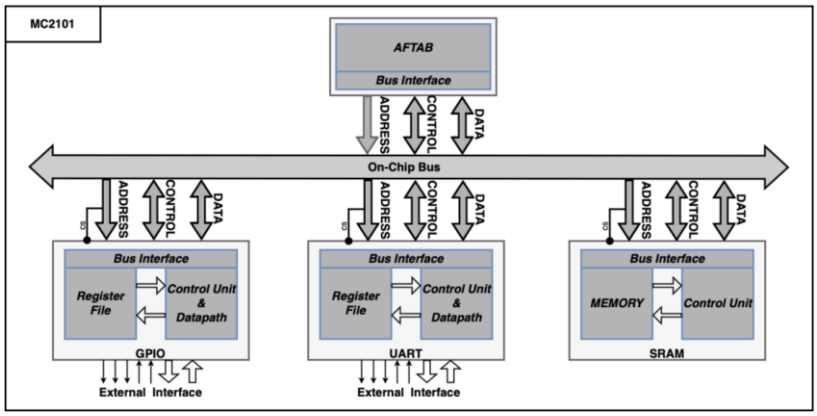

Figure 3.1: MC2101 microcontroller

Figure 3.1 exposes the design of the microcontroller. The architecture includes a single bus, which provides the system with the proper hardware infrastructure necessary to interconnect the AFTAB processor with all peripherals and the main memory. The processor plays the role of the master: it is the only component able to initiate transactions on the bus, which can be composed of single or multi-cycle read/write operations. Peripherals and main memory are accessed by AFTAB using the memory-mapped mode: this means that the entire address space includes both peripherals and main memory. Therefore, the processor uses the same load/store instruction for accessing all components attached to the bus.

The communication between the processor and any peripheral happens only through the peripheral's register file, which can contain three different types of registers, with different read/write policies:

- **Control Registers:** used to configure the peripheral functionalities. They are written by the processor and read by the peripheral

- **Status Registers:** used to report the current state of the peripheral. Moreover, a state register may contain information on whether the device is ready or if an error occurred. These registers are written by the peripheral and read by the processor
- **Data Registers:** used to exchange data. They hold data that is transferred to and from the peripheral by the processor core, the AFTAB core in this case. Both processor and peripheral can read and write on them

The size and the number of registers depends on the functionalities implemented by the peripheral, which can be for instance a 32-bit peripheral, with a 32-bit register file or an 8-bit peripheral with 8-bit registers. The bus implements three different types of interconnections:

- **Data Lines:** set of independent read and write data signals used to read and write data on peripherals registers and on the main memory
- **Control Lines:** used to provide signals for controlling read or write operations and for allowing peripherals and main memory to feedback to the processor about the current state of the transaction
- **Address Lines & Chip Select:** the most significant bits of the address lines pass through a decoder that drives all chip-select signals: in this way, the peripheral is activated only when necessary. On the other hand, the least significant bits of the address select the register.

Each component attached to the bus must include the Bus Interface, which is a sort of wrapper to be placed around each component in such a way to bridge signals coming from the bus to the internal hardware (FSM + Datapath) of the peripheral and vice versa. Peripherals can implement an additional External Interface, which is used to handle the communication to the external world. In particular, the conversion of incoming external asynchronous digital signals into the synchronous domain of the microcontroller.

3.1.2 Bus infraestructure

This architecture preserves a minimal set of AMBA specifications to build a simple infrastructure, that remains modular and upgradable at the same time. Figure 3.2 shows the block diagram of the bus system. It is an infrastructure that supports a single master with multiple slaves; thus, it does not have master arbitration.

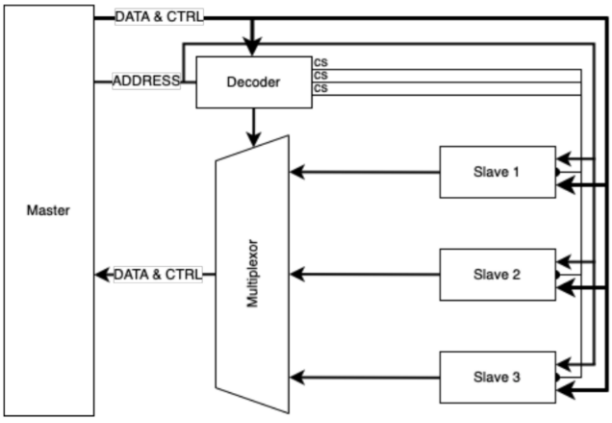

Figure 3.2: MC2101 bus block diagram

The bus is a shared resource. Hence, a multiplexer drives the shared resource with the purpose of avoiding concurrent driving of wires. Consequently, the bus interconnection logic consists of a

single centralized address decoder and a slave-to-master MUX. The decoder monitors the address lines driven by the master so that the appropriate slave is selected during each transaction. It also provides control to the multiplexor, which is in charge of routing the corresponding slave output back to the master.

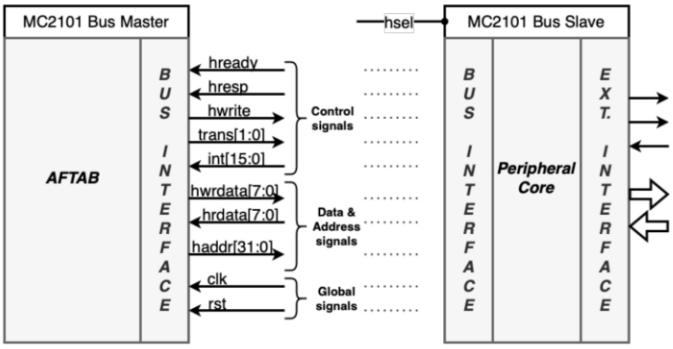

Figure 3.3: Master & Slave Interfaces

The bus master interface, on the left of Figure 3.3, provides address and control information to initiate read and write operations. The slave, whose interface is shown on the right of Figure 3.3, responds to transfers initiated by the master, by using a *hsel* (chip select) signal from the decoder to control when to respond to a request. When selected, the slave will start monitoring the bus lines in order to respond to commands from the master. Therefore, all the slaves will stay in an idle state until they are individually waken up by a signal.

In addition, each slave is also able to feedback to the master about the success, the failure or the waiting for the data transfer. To do so, the *hresp* and *hready* dedicated lines. In this way, a slave is able to extend the data phase when extra time is needed, but also to inform the rest of the system that some bad operations are happening in the bus. The full list of signals is described in Table 3.1.

3.1.3 GPIO peripheral

The GPIO (General Purpose Input Output) is a peripheral module present in all embedded processors. It is used to manage sets of SoC's incoming and outgoing digital signals, by driving and checking the logic state of physical pins. GPIOs can be used in a diverse variety of applications, limited only by the electrical and timing specifications of the peripheral's interface, and the ability of software to interact with it in a sufficiently timely manner. In most cases, GPIOs are used to switch LEDs, interface the microcontroller to buttons, user-selectable switches or electronic switches (relays). In other cases, there is also the possibility to use GPIO as a bit-banging communication interface, where software is used as a substitute for dedicated hardware in order to implement a specific communication protocol, e.g., a software-based SPI bus with 4 GPIO pins.

Figure 3.4 shows the core of the peripheral together with its relative set of signals coming from the Bus Interface and from the External Interface. The GPIO is designed as a 32-bit peripheral; hence, it provides a 32-bit processor's interface with 32-bit wide user registers and data bus (*dataIn* and *dataOut*). The conflict between MC2101's data bus, which is on 8-bit (see *hwrdata* and *hrdata* in Table 3.1), and the peripheral's data bus are solved by the bus interface, which provides a bridge for the communications between the two domains. The External Interface (in Figure 3.4 as Pads Interface) works as an intermediate between the bidirectional bus connected to physical pins and the peripheral's core. In particular, through the Pads Interface, the asynchronous bidirectional external lines (*gpio_pads*) are separated into independent input and output lines, synchronized with the global clock.

Bus signals		
Name	**Type**	**Description**
hready	Control	When driven LOW, the transfer is extended
hresp	Control	When HIGH, indicates that the transfer status is on error
hwrite	Control	Indicates the transfer direction. When HIGH the signal implies a write transfer, on the contrary, when LOW a read transfer
htrans[1:0]	Control	Shows the current state of the bus
int[15:0]	Control	Independent interrupt lines, can be driven by peripherals to issue IRQs
hwrdata[7:0]	Data	8-bit data lines from the master to the slaves
hrdata[7:0]	Data	8-bit data lines from the slave to the master
haddr[31:0]	Address	Address space is on 32-bit; thus, also the address line
clk	Global	Global clock signal
rst	Global	Global reset signal

Table 3.1: MC2101 bus signals

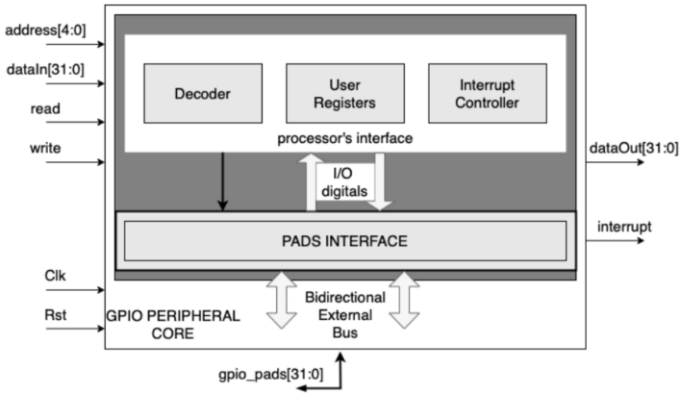

Figure 3.4: MC2101 GPIO Peripheral core

The *5-bit address* signal selects which user register is being accessed by the processor. As explained before, read and write operations on peripherals are enabled by a proper chip select signal. When the chip select condition is met, the bus interface raises one of the two strobe signals, read and write, for accessing the user registers. The GPIO provides also an *interrupt* line to the processor. This line will rise as soon as an interrupt condition occurs on any of the 32-pad lines. When this happens, the processor will have to execute appropriate *read* or *write* operations to de-assert the interrupt.

The peripheral, programmed through the user registers, is able to support the following functionalities:

- Control the input/output direction of each GPIO pad
- Enable interrupts for each input bit and configure the triggering behavior on logic levels or rising/falling edges
- Drive and control external pins

From the programmer's point of view, the GPIO peripheral can be programmed through the set of registers in Table 3.2.

3.1.4 UART peripheral

Microcontrollers and computers mostly use UART as a form of device-to-device communication protocol, which only requires two wires to implement transmission and reception of data.

The UART module designed for MC2101, whose block diagram is shown in Figure 3.5, provides a transmitter-receiver pair, configurable for different speeds, data widths, parity codifications and information status for several error conditions. The implementation provides a subset of the standard UART 16550 specifications, without including some more advanced functionalities for supporting DMA and MODEM communications.

The module is designed as an 8-bit peripheral: hence, it provides an 8-bit data interface to the processor. This characteristic makes the Bus Interface lighter than the GPIO's ones, because in this case, UART's internal data lines are already compatible with the bus infrastructure of the microcontroller. Also, the external interface is lighter because only two external lines, *Rx* and *Tx*, are controlled. Similarly to the GPIO, the signals in Figure 3.5 are generated by the bus interface, except for the *Rx* and *Tx* lines which came from the external interface, with the difference that, in this case, the user register file is more compact and requires only 3-bit for the addressing. Both receiver and transmitter use a dedicated queue, implemented in hardware as a FIFO memory, used to hold data either received from the *Rx* serial port or to be written to the *Tx* serial port. This buffering feature

GPIO User registers			
Name	**Address**	**Access**	**Description**
PADDIR	0x1A100000	R/W	Control the direction of each of the GPIO pads. A value of 1 means the pin is configured as output
PADIN	0x1A100004	R	Saves the input values coming from input pins
PADOUT	0x1A100008	R/W	Drives the output lines with its content
PADINTEN	0x1A10000C	R/W	Interrupt enable bits for input lines
INTTYPE	0x1A100010	R/W	Two registers: *INTTYPE0*, *INTTYPE1* that are used to control the interrupt triggering the behavior of each interrupt-enabled pin
INTSTATUS	0x1A100018	R	Contains interrupt status for each GPIO line. The interrupt line is high when a bit is set in this register and will be de-asserted when this register is read

Table 3.2: GPIO User Registers

is particularly useful when the interrupt mechanism is enabled at the receiver side, which can raise interrupts when its queue surpasses a certain fill level, instead of triggering every time a new character is received. The processor can also benefit from this buffering feature by filling the transmitter's FIFO when multiple characters must be transmitted, without the need of wasting polling cycles in waiting for the completed transmission of each character sent. With both the FIFO empty, is possible to have 17 characters simultaneously: in the transmitter, 1 being sent and 16 buffered, while in the receiver, 16 ready to be read and 1 being assembled.

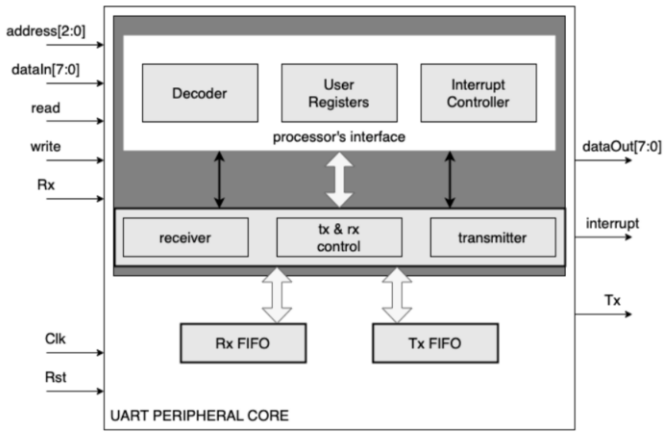

Figure 3.5: MC2101 UART Peripheral core

As anticipated, parity conditions and error controls are part of the implemented functionalities. The following error conditions can be detected by the receiver:

- **Break Interrupt:** Error flag asserted when the *RX* line remained stuck at 0 for the entire character time. This error is usually generated from incorrect wiring, e.g., the *Rx* or *Tx* line is mistakenly wired to a ground pin
- **Framing Error:** Asserted when the stop bit was not detected. Usually, this type of error is generated when a device is sending data at a different speed with respect to the one used by the receiving device for sampling
- **Parity Error:** Asserted when the parity of the received character is wrong according to the current one configured. This provides a very simple and useful error-detection feature
- **Overrun Error:** Produced when a character is assembled but there is no more space inside the receiver's FIFO. The UART peripheral must be able to inform the processor that it will lose data if the FIFO is not read

Regarding the interrupts, the UART module can be configured to assert an interrupt when different conditions are detected, each one with an associated priority.

Table 3.3 summarizes the different conditions that can be a source of interrupt and their relative priorities. From Table 3.4 is possible to see that all registers are on 8-bit and in fact are all byte-aligned, with respect to the GPIO register file which is word-aligned, because all registers are on 32-bit. Another detail is in the *RHR* and the *THR registers*, it is possible to see that they are on the same address. This is not a typo. Since *RHR* is accessed for reading and *THR* is accessed for writing, it is possible to use the same address for accessing both in an exclusive way. In particular, during a read operation, *RHR* is accessed, instead, during a write operation the *THR* is addressed. This trick allows to include all 9 registers in a space that theoretically can only be allocated for 8, saving a bit in the address line.

UART interrupt sources		
Name	**Priority**	**Description**
Receiver Line Status	Level 1(MAX)	There is an overrun error, parity error, framing error or break interrupt indication in the received data on the top of the receiver's FIFO
Receiver Data Ready	Level 2	The number of characters in the reception FIFO is equal or greater than the programmed trigger levels
Reception Timeout	Level 2	There is at least one character in the receiver's FIFO and during a time corresponding to four characters at the selected baud rate no new character has been received and no reading has been executed on the receiver's FIFO
Transmitter Empty	Level 3	The transmitter's FIFO is empty

Table 3.3: UART Interrupt Sources

UART User Registers			
Name	**Address**	**Access**	**Description**
IER	0x1A100000	R/W	Used to individually enable each of the possible interrupt sources
ISR	0x1A100001	R	Used to identify the interrupt with the highest priority that is currently pending
FCR	0x1A100002	W	Used to reset the FIFOs and program the receiver trigger level for the Received Data Ready interrupt
LCR	0x1A100003	R/W	This register controls the way in which transmitted characters are serialized and received characters are assembled and checked
LSR	0x1A100004	R	Used to inform the user about the status of the transmitter and the receiver
DLL & DLM	0x1A100005	R/W	Two different registers that together form the 16-bit Divisor Latch, which contains the divisor value used to program the baud rate of the communications
RHR	0x1A100007	W	Contains the most recent received character
THR	0x1A100007	W	Contains the character to be transmitted

Table 3.4: UART User Registers

The UART peripheral can be programmed through the set of registers in Table 3.4.

3.2 Software libraries

3.2.1 General description

A proper set of libraries is included in the design with the purpose of facilitating the programming of the microcontroller, but also integrating the possibility to use all the classical string manipulation functions as well as the *printf* and *scanf* that are useful also for testing activities.

Before entering into details, the first thing that usually is presented when talking about the programmer's point of view is the memory map of the microcontroller. MC2101 architecture supports a 32-bit address space, which virtually corresponds to 4 GB of memory. Figure 3.6 shows the default memory map of MC2101. By carefully observing the size of the memory sections it is possible to note that the physical portion of memory currently mapped is in the order of KB, there is a huge free space that can be used in future expansions. In particular, the main memory has a lot of available space for being extended to support more advanced software implementations, e.g., hosting an operating system.

Moreover, Figure 3.6 can be compared with the memory space of the AFTAB processor(see Figure 2.5. The MC2101 expands the size dedicated to the instruction memory with 4 KB and also the sizes for the Data memory and Stack are increased. The size of the peripherals remains the same; however, Figure 3.6 explicitly describes the addresses corresponding to the GPIO and the UART.

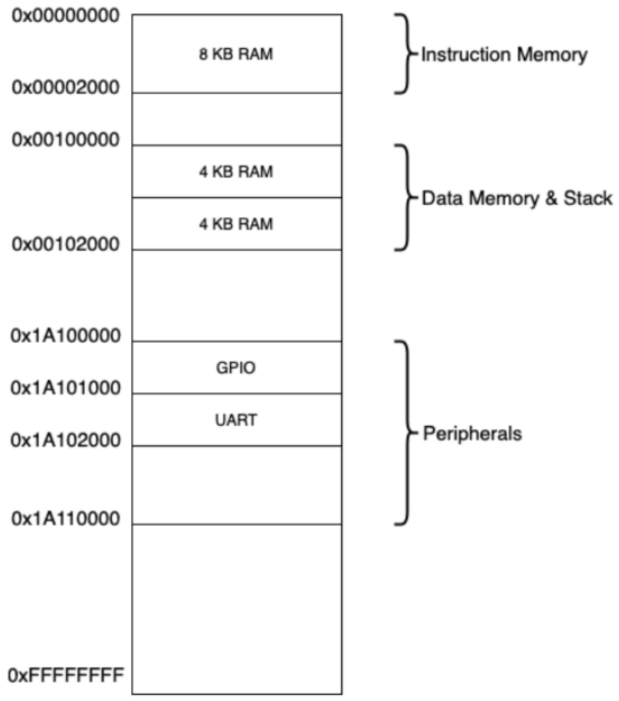

Figure 3.6: MC2101 Memory map

The following subsections present all the possible high-level functions that a programmer can use to write programs for MC2101, maintaining a certain degree of abstraction from the hardware. For this purpose, the GPIO and UART peripherals libraries include a wide amount of functions that, together with macro definitions, allow to program the functionalities without the need of manually access the user registers manually.

GPIO library functions	
Function	**Description**
set_pin_direction()	Used to set a pin direction to input or output
get_pin_direction()	Returns the direction of a given pin
set_pin_value()	Used to set a pin voltage level to low/high
get_pin_value()	Returns a given pin's voltage level
set_pin_irq_enable()	Enable or disable interrupt on a pin
get_pin_irq_enable()	Get the programmed pin's interrupt enable flag
set_pin_irq_type()	Used to configure the interrupt triggering behavior for a given pin. Logic Levels or Edges. The pin must have its interrupt flag enabled
get_pin_irq_type()	Returns the programmed pin's interrupt triggering behavior
get_gpio_irq_status()	Returns GPIO's current interrupt status register (*INTSTATUS*) value. Responsible also to de-assert the GPIO pending interrupt
ISR_GPIO()	GPIO interrupt handler. When the interrupt is raised, the bootloader will jump to this function

Table 3.5: GPIO Library functions

3.2.2 GPIO library

Table 3.5 shows a simplified prototype and explanation of the GPIO functions currently implemented in the library. The library implements all the necessary functions to be used for programming the peripheral without the need to directly access its registers; thus, providing an adequate level of abstraction. The library offers the possibility to configure the direction of each of the 32 pins, read and write values on them. But also, provide a set of functions to be used to enable interrupts on any pin and write a custom interrupt service routine.

3.2.3 UART library

The UART library offers a bigger set of functions with respect to the GPIO library, functions that are also more complex (in terms of execution time) because the UART features are more complicated. The library provides the possibility to configure the speed of the communications, the size of the frames transmitted/received as well as all the error detection mechanisms. Provides the programmer functions able to send single characters or strings and to customize the interrupt behavior. More details about the library are reported in Table 3.6.

3.2.4 String library

More advanced I/O functionalities have been implemented in the string library. This library is particularly important because it provides the principal string manipulation functions that together with

UART Library functions	
Function	**Description**
uart_set_cfg()	Used to program the LCR register for configuring the character width, number of stop bits, parity type and enable, even/odd parity and the baud rate
uart_get_cfg()	Return the current LCR register value
uart_set_int_en()	Configure the IER register to enable the different types of interrupt sources
uart_get_int_en()	Used to get the enabled interrupt sources by reading the IER content
uart_rx_rst()	Clear the content of the receiver's FIFO
uart_tx_rst()	Clear the content of the transmitter's FIFO
uart_set_trigger_lv()	Set the receiver's FIFO trigger level
uart_get_lsr()	Used to read the Line Status Register (LSR)
uart_sendchar()	Send a character on the transmitter line
uart_getchar()	Get the character received
uart_send()	Used to send a string on the transmitter line
ISR_UART()	UART interrupt handler. When the interrupt is raised, the bootloader will jump to this function

Table 3.6: UART Library functions

the UART Library are able to support the *printf* and *scanf* functions.

The following functions are part of the system's library: *strlen*, *strcpy*, *strcmp*, *puts*, *putchar*, *memset*, *printf* and finally the *scanf*.

3.2.5 Board library

This is the "top-level" library that should be included in every application developed for MC2101. It contains the *board_setup* function, which was prepared right after the pin planning phase. It is used to initialize the microcontroller hardware once synthesized on FPGA and connected with the various user buttons, switches and LEDs present in the board.

Once called, the microcontroller is configured in this way:

- UART peripheral is programmed with 115200 standard baud rate, no parity, 1 stop bit, and 8-bit character width. This is a standard configuration for the *printf* and *scanf* functions
- GPIO lines interconnected to LEDs are set as output.
- GPIO lines interconnected to User buttons are set as input
- GPIO lines interconnected to Switches are set as input
- All interrupts are disabled

The *board_setup* function should be first called in every application because brings the microcontroller in the correct configuration, according to the pin assignment used for the synthesis, in order to properly interface the external hardware.