Simple Enough Processor : From Scratch Elaborated Simulated Educational CPU

This paper was downloaded from TechRxiv (<https://www.techrxiv.org>).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

30-08-2021 / 02-09-2021

CITATION

Kara, Abdelaziz; Mostefai, Messaoud (2021): Simple Enough Processor : From Scratch Elaborated Simulated Educational CPU. TechRxiv. Preprint. <https://doi.org/10.36227/techrxiv.16539843.v1>

DOI

[10.36227/techrxiv.16539843.v1](https://doi.org/10.36227/techrxiv.16539843.v1)

Simple Enough Processor : From Scratch Elaborated Simulated Educational CPU

Abdelaziz Kara and Messaoud Mostefai

Abstract—In many cases, teaching Computer Organization and Architecture (COA) involve the use of a simplistic processor to construct step-by-step by the student, but often those processors are unrealistic and far from depicting the behavior of the real-world processors. This work is an attempt to create a simplistic 8-bits CPU using an academic logic simulator called Logisim, yet an effort is made to close the gap with the real-world processors by supplying their major capabilities, like multiple programmable registers, multiple addressing modes, shift instruction, comparison and logic instructions, multiple branch and jump instructions the stack and subroutine mechanisms, flags indicators, input/output mechanism, a proper assembler... etc. The processor would comprise 21 instructions and 4 addressing modes, with the capability using the flexible control unit sequencer, to easily add more customizable instructions. In the final result the student would be familiar with hardware microarchitectural and computer organization concepts, and at the same time, acquired computer architectural knowledge in concordance with real-world processors and architectures.

Index Terms—Computer engineering, Computer Organization and Architecture, CPU design, Design projects, Electrical engineering, From scratch, Processor simulation, Project based learning.

I. INTRODUCTION

IT'S become more and more obvious for many universities to integrate into their computer engineering and computer science curriculum the design of a proper CPU (Central Processing Unit), and not just use it like an opaque black-box, yet the design of the CPU must remain simplistic and not overwhelming for the student. Many Benefits could be enumerated from designing a proper CPU, like for example the full understanding of the Hardware/Software interface and how each part could influence the other, the gain in hardware design and computer organization skills, paving the way for understanding more complex hardware mechanism like pipelining, caches, virtual memory, out-of-order execution... etc.

There are several attempts to create the ideal academic CPU, simple enough to be feasible by a student, and in the other hand, faithful to the genuine behavior of a real CPU. Our purpose was to go through a design of a CPU that resemble a real processor yet simple to design. We aimed to capture the major real-world CPUs abilities, like multiple programmable registers, inter-registers transfer instructions, different addressing modes, multiple arithmetic and logic operations, including shift operation, using the stack and subroutine mechanisms, using flags indicators, different branch and jump instructions, input/output mechanism, integrating the CPU inside a whole simplistic Von Neumann architecture, microprogramming the instructions, using a proper assembler that compiles to the

machine code. Still the microarchitecture design should be kept simple enough to be assimilable by students. We chose to design an 8-bits CPU inspired by the 80s 8-bits CPUs, like 6502, Z80, Intel 8080, owing to the fact that still the modern CPUs inherited most of their abilities and characteristics from those CPUs.

Different works tackled this architectural learning issue with different approaches, we can classify them in four different categories. First the hypothetical processors, where the CPU is described theoretically without a real implementation, we can mention MIX [1] and MMIX [2], SAP-1, SAP-2 and SAP-3 [3], MERIE [4]. Even though those CPUs were described theoretically by their authors, different implementations exist in academia realm. Second, CPU simulators or virtual processors, where a simulator for the architecture (external behavior) and the microarchitecture (internal behavior) of a CPU is simulated, often supporting the corresponding assembler. In this case, it's almost impossible to modify the hardware design, some examples are the MikroSim virtual processor [5], Very Simple CPU Simulator and Relatively Simple CPU Simulator [6] or the online simulator The Web Elemental Processor SIMulator [7]. The third type concerns the design of a CPU using a logic simulator, our CPU falls in this category, and we have used an intuitive and basic logic simulator called Logisim [8], different works were done in this regard including MIC-1 [9] [10] [11] [12], Linda Null's sixteen bit MARIE [13] and one-address CPU [14]. And finally, a physical implementation of the CPU using FPGAs or electronic logic circuits like 74-TTL series, some works using FPGAs include VSCPU [15], TINYCPU [16], ECOMIPS [17], and others using logic ICs (Integrated Circuit) like GIGATRON [18] and CHUMP [19].

We tried in this paper to provide a learning foundation on how to design 8-bit like processors following step-by-step the construction of the processor SEP (Simple Enough Processor) using the simple and friendly logic simulator Logisim. Section 2 will introduce the core elements of SEP architecture and its integration with a simplistic Von Neumann Architecture. In section 3, a detailed elaboration of the internal hardware design, also known as the microarchitecture or computer organization, of the CPU is provided. In section 4, the machine language and the assembler of the CPU are presented with an example for demonstration, a third-party software called CustomAsm [20]. And finally in section 5, a discussion about the comparison table between the major educational processors and their significant abilities in the purpose to point out SEP contributions.

Fig. 1. SEP CPU as part of Von Neumann architecture.

II. SEP CPU DESCRIPTION

This paper describes the process of constructing a whole simplistic 8-bit processor called SEP CPU, all within a simplistic Von Neumann architecture, with a small memory of 256 Bytes and an input 8-bit binary keyboard device and an output signed 8-bit 3 digits seven segment display device, the whole architecture is shown in Fig. 1. This section treats the architectural side of the processor, by describing and giving information on how the architecture could be used by the programmer.

A. Machine language

SEP CPU is an 8-bit processor inspired by processors from 70s-80s era, like 6502, Zilog z80, and Motorola 6800, its binary format follows the same constructions, where instructions are binary encoded with 8-bit width opcodes (operation code), followed potentially by one, two or three 8-bit width operands, the operands can be values often called immediates, or memory addresses. Actually, the SEP CPU instructions are encoded only in one or two bytes, one for the opcode, followed potentially, depending on the instruction, by one operand.

B. Registers

A minimum number of registers are used for the processor, only five registers are used, two of them are programmable and could be used by the processor instructions, the others three are used for nonarchitectural purposes, typically for the inner functioning of the CPU. All the registers are 8-bit width. The Table I describes each one of them :

C. Instructions

The Instruction Set Architecture (ISA) for the SEP processor comprise a small set of 20 instructions (More accurately the ISA comprise 27 instructions, some instructions use two addressing modes, as shown in Table IX), easily extendable to add more instructions, thanks to the flexibility offered by the sequencer used within the controller unit. The instructions can be categorized in nine different classes, register-register

TABLE I
SEP REGISTERS TABLE

Register type	Register name	Register description
Programmable registers	Accumulator (Acc)	The CPU main register, used implicitly as the first operator in arithmetic and logic operations, and it is also the implicit result register.
	Stack Pointer (SP)	Contains the stack pointer address. Generally, this register is indirectly modified by the instructions.
Nonarchitectural registers	Program Counter (PC)	Contains the actual executing instruction address in memory.
	Instruction Register (IR)	Contains the actual executing instruction opcode.
	Address Bus (AB)	This register outputs directly to the address memory bus. It contains the address of the memory cell currently accessed by the CPU.

TABLE II
SEP INSTRUCTION SET TABLE

Register type	Name	Register description
register-register transfer	TAS	Copies the value of Acc to SP.
	TSA	Copies the value of SP to Acc.
Memory Access	LD	Load from memory to Acc.
	STR	Store the value of Acc into memory.
Arithmetic	ADD	Addition operation and saving the result on Acc.
	SUB	Subtraction operation and saving the result on Acc.
	LSR	Logical shift right by one bit of the Acc.
Logic	NAND	Not-AND bitwise operation and saving the result on Acc.
Comparison	CMP	Subtraction operation without saving the result on Acc, afterward the flags are used for the comparison condition.
	JMP	Unconditional jump to a nonsequential instruction address.
Jump and branch	BLT	Jump to a nonsequential instruction if the flags verify a condition "less than".
	BGE	Jump to a nonsequential instruction if the flags verify a condition "greater than or equal".
	BEQ	Jump to a nonsequential instruction if the flags verify a condition of "equality".
	BNE	Jump to a nonsequential instruction if the flags verify a condition of "not-equality".
Stack	PUSH	Push the value of Acc on the stack.
	POP	Pop from the stack top to the Acc.
subroutine	JSR	Jump to the address of the subroutine. In other words, it is the calling instruction for the subroutine, and at the same time the PC is stored on the stack.
	RTS	Return from the subroutine to the instruction just after the calling instruction JSR by recovering the PC from the stack.
input/output	IN	Get a value from the input CPU port to Acc.
	OUT	Put the value of Acc to the output CPU port.
No-op	NOP	This instruction do nothing.

transfer instructions, access memory instructions, arithmetic instructions, logic instructions, comparison instructions, jump and branch instructions, stack instructions, subroutine instructions, input/output instructions. The Table II lists all the SEP processor instruction set.

TABLE III
SEP ADDRESSING MODES TABLE

Addressing mode	Description
Implicit	The instruction does not need any appended operand, the opcode is sufficient for its execution. e.g. PUSH instruction will always push the top of stack into the Accumulator, and the instruction does not need any additional indication.
Immediate	The instruction is followed by a signed integer value. In the machine code representation the opcode is appended by a two's-complement signed 8-bit operand. e.g. ADD -5 will add Acc to -5.
Direct	The instruction is followed by the address of the value in memory. In the machine code representation the opcode is appended by an 8-bit address operand. e.g. ADD (9) will add Acc to the value in the memory cell pointed by the address 9.
Indirect	Used only with Jump instruction. Normally in the direct mode, Jump is followed by the address of the instruction to jump to, but in the indirect mode the Jump is followed by the address in memory that contains the address of the target instruction. e.g. JUMP (9) will cause the CPU to execute the instruction pointed by the value inside the cell memory with the address 9.

D. Addressing modes

SEP implements four of the most common existing addressing modes in real CPUs, the implicit, the immediate, the direct, and the indirect mode. Despite the fact that index mode is one of most used addressing modes out-there, it was not added to the CPU, due to the significantly increasing internal complexity of the CPU microarchitecture. The Table III describes each one of them.

E. Using flags

Some real-world CPUs use flags register primarily to save the states of the CPU after finishing executing one instruction for the main purpose to be potentially used by the next instruction. Normally, each 1-bit cell of the register save a particular state, for instance, they can save the sign of the instruction operation result, save the overflow state, save the carry bit... etc. Two binary cells were used as flags in SEP, NF for Negative Flag, indicating if the result is negative or not (1 indicating a negative result), and ZF for Zero Flag, indicating if the result is null or not (1 indicating a null result). The flags mechanism is primarily used by branch instructions, usually a CMP instruction should precede them to update the flags, the branch instructions would then rely on the flags to deduct the relational condition.

III. CPU DESIGN STEP BY STEP

This part describes the computer organization of the CPU, it responds to the question of how the CPU is built up inside, what components are involved and how they are connected to form the microarchitecture. The way and the steps to construct the hardware for a new CPU are relatively common in the literature and can be perceived in many works, like [9] [13]citeFeinberg. The first step generally consist of creating the ALU (Arithmetic and Logic Unit), in this step the designer should decide about the arithmetic and logic operations to be

used by the CPU, sometimes some additional operations are added to micro execute the instructions themselves, like the incrementation, which for instance could be used to increment the PC after each instruction. The ALU represent generally the easiest part to design amongst the three most important parts inside a CPU, the two others are the datapath (DP) and the control unit (CU).

The second step is the datapath, this part has a huge impact on the nonfunctional aspects of the CPU, it is possible to find different datapath implementations for the same processor but with different characteristics [21]. In this part, it is essential to sway between the two principal CPU design trade-offs, the performance with more components and higher price, and design simplicity with less components and less power consumption. Often the ISA which represents the software interface, is well defined before the CPU hardware design, like in RISC-V [22], but in some cases the datapath could influence the instruction set, especially when the ISA is semi-defined, which means that the complete definition is intentionally not totally finished, allowing some optimizations in the CPU design, essentially in the datapath. Still, it is always possible to adapt a datapath to a specific completely defined instruction set [22].

The third and the last is the control unit, different kinds of CU can be used, the simplistic one is for the single cycle processors, where an entire instruction is executed in one clock cycle, in this case a combinational circuit could be used for the CU, or an equivalent implementation using look-up table ROMs [19]. For the multicycle processor, in which an instruction needs many cycles to achieve, a finite state machine sequential circuit is used for the control unit, and often a more advanced controller is used in this case, called the sequencer, like in [9], which allows a more flexible microprogramming of instructions. More elaborate controllers should be used for more advanced processors, like the pipelined and out-of-order ones.

A. The ALU

A very simple method is often used to design ALUs, involving the implementation in combinational logic circuits of all the arithmetic and logic operations imagined by the CPU designer, they are arranged in parallel manner so that every circuit receives at the same time the two ALU operand and calculates its operation in parallel with other circuits, the results of all the circuits are then fed to an exit multiplexer, shown in Fig. 2, which will decide to output only one result and discard the others, depending on the operation/function selection code.

The SEP CPU aims to be simple, only two arithmetic operations were used, addition and subtraction, with two more logical operations, the bitwise NAND and the shift right by one bit operation, knowing that NAND is sufficient in software to implement the other remaining logical operations, and the counterpart left shift by one bit is also easily reproduced programmatically by adding the selfsame value, then $X+X$ will shift left X value by one bit. Four other operations were added, aimed to be used internally for the proper working of

Fig. 2. Generic way to design an ALU.

Fig. 3. SEP ALU circuit under Logisim.

the CPU, the function A, which returns only the value of A, the function B, the function B+1, and the function B-1. The Fig. 3 shows the ALU realization on Logisim, and the Table IV lists the function codes for each ALU operation.

We can also see on Fig. 3 the zero detector output from the ALU, this output is used to detect if the result is null or not, and thus this output is intended to feed the ZF flag. In the other hand, NF flag is directly tied-up to the *Output* last bit, which is the sign bit of the ALU result.

This method described there is very primitive to design efficient ALU, meanwhile it could be a good starting point to design real-world ALU. Mainly two types of optimizations could be done to enhance this kind of ALU, first by reducing

TABLE IV
SEP ALU COMMANDS/FUNCTIONS TABLE

Functions	Binary code	Equivalent number
A	000	0
B	001	1
A+B	010	2
A-B	011	3
B+1	100	4
B-1	101	5
A >> 1 (Shift Right by 1 bit)	110	6
A nand B	111	7

the overall complexity circuit by reducing the number of logic gates, and secondly, by reducing the longest signal path from the entrances toward the outputs and gaining in speed and frequency.

B. The Datapath

Designing a new CPU begins generally by conceiving the instruction set, or a part of it, but it is not uncommon to find a design mix of the two, ISA and datapath, knowing that instructions and DP are strongly dependent, in a sense where the DP is primarily the real effective means to execute an instruction inside the CPU. Formally, a datapath could be defined as an oriented graph with two kinds of nodes, memory nodes and processing nodes. Memory nodes are essentially registers, register files (a set of registers assembled all-together) or RAMs, their unique objective is to temporary stock the information while moving through the DP. It's worth mentioning that the memory circuits need a synchronization clock signal to save information, and on the other hand, the processing nodes are used to treat, process and operate over the information, usually they are combinational and do not need a clock to perform, the best example of them is the ALU.

An instruction after being decoded by the control unit, is executed by moving or traveling the information through a specific path across the DP mesh structured by memory nodes and processing nodes. Each instruction from the ISA pursue its unique path, performing a hop traverse. The information for one clock cycle starts from a memory node and finishes in another memory node passing across an intermediate processing node, thereby for each clock cycle the instruction is partially executed, and several clock cycles are required to accomplish one entire instruction. This kind of processor is called multicycle processor. For the single cycle processors the same principle applies, with only one hop passing through multiple processing nodes for the entire instruction.

The example in the Fig. 4 shows a part of a datapath involved in the ADD instruction, executed in one clock cycle. The memory nodes Acc and RAM are the starting point in the path instruction, the two operands stored in the nodes are provided to the ALU to be processed, the CU after decoding the ADD instruction must choose the addition function code (010) for the ALU, after adding the two numbers by the ALU the result will go back to Acc, and the CU again will commands the Acc to store the result value at the front edge of the clock ending the current cycle.

Fig. 4. Illustrative example of a portion in a datapath.

Fig. 5. SEP CPU microarchitectural datapath.

The big picture is that depending on the instruction decoded, the CU will send command signals to the nodes in the DP forming a specific traveling path for the information to go through, the traveling is done by hopping where at each clock cycle the information goes from one memory node to another passing through processing nodes. The multiplexers are a special type of nodes also used in DPs, they are used especially to irrigate the information flow inside the DP mesh, they are controlled by CU to choose a particular path for the information to follow, likewise a switchman in a train railroad. A golden rule [23] to design an efficient CPU is to make common and most used instructions run fast, and take the least cycle count possible. In counterpart, the least uncommon ones could run slow.

The SEP CPU datapath microarchitecture is shown in Fig. 5, the aim of its design was intended to be simple, we can see that only one multiplexer was used to decide which path should be taken to the ALU B operand. The CU gets information from IR, to be able to decode the instruction opcode, in addition from ZF and NF flip-flops allowing him to decide to jump or not in branch instructions. For simplicity, it was not drowned on the Fig. 5, but most of the nodes get a command signal from the CU. Five input/output ports are also shown in the Fig. 5, they are used by the CPU to communicate with the external world components, for example, the memory out data port is used to send data to RAM and the input memory port is used to bring back data from RAM, the memory address output is used to provide memory addresses to the RAM, the input and the output ports are used respectively to get data from the binary keyboard and to send data to the seven segment display device.

C. The Control Unit

Like mentioned before, different kinds of CU exist, we decided to use a sequencer mechanism to implement the SEP CU, the sequencer works for multicycle processors, and it has the benefit to be more flexible than his counterpart the finite state machine, it makes adding, removing and updating instructions much more easier, even the actual home PCs Intel and AMD processors use a similar mechanism [24]. The particularity of a sequencer is its ability to be programmable, the process is often called the microprogramming of a CPU. The Fig. 6 shows the internal implementation of the sequencer inside the SEP CU, it is realized by constructing a loop that executes one microinstruction per iteration, the three main elements constructing the loop are, the MicroProgram Counter register (MPC), the MicroProgram ROM (MP-ROM), and the Control Signals Register (CSR). Generally speaking the sequencer execute the microinstructions inside the MP-ROM one after another, but there are cases where the execution should jump to another nonsequential microinstruction, each microinstruction formalize a hops through the DP, which means the instruction is built by a sequence of microinstructions.

The loop begins with the MPC register that contains the address of the actual microinstruction stored in MP-ROM, thus it is directly attached to the address port of the MP-ROM, on the other side, the MP-ROM stores the list of 24 bits long microinstructions, this list is often called microprogram, then the MP-ROM sends the concerned microinstruction to the CSR register¹, the CSR stores the 24 bits long microinstruction that mainly contains command signals to be sent outside the CU to command the DP nodes, the MPC will get the next microinstruction at the end of the loop, and the loop continues so forth.

An instruction globally in its execution go through three ordered phases, the fetch phase in which the CPU looks for

¹In Fig. 6 The clock of the CSR register is inverted, making a half cycle delay to give enough time for the propagation of the signal coming from MPC through MP-ROM. And the same for the control signals, they have enough time to propagate to DP nodes before the next rising edge of the clock.

Fig. 6. SEP CU sequencer circuit diagram.

Fig. 7. Binary encoding of a microinstruction inside CSR.

the instruction in the RAM using its address inside the PC register, the next phase is about decoding the opcode of the instruction, this is done by the CU retrieving the pcode from IR register, and finally the execution phase, where the proper microinstruction sequence of the instruction is unrolled.

The internal binary encoding of a microinstruction is presented in Fig. 7, it shows three different fields, the most important is the command signals fields with thirteen bits, each one responsible for commanding one DP node, the description of the overall signals is written down in Table V, the sequencer control field decide of the next microinstruction sequence behavior, and the jump address field holds a potential jump-to microinstruction address.

The three bits used for the sequencer control inside CSR are used directly by the Sequencer Controller (SC) circuit, this circuit is responsible for filling the MPC register with the address of the next microinstruction to execute, four possible different address sources in destination to the MPC are available and the SC should choose one from them by using the multiplexer shown in Fig. 6. The MPC could be incremented by one using the +1 combinational circuit, in this scenario the sequential microinstruction on the MP-ROM is used. The second choice for MPC is the jump to the address stored by the actual microinstruction in CSR. The third is getting the instruction opcode coming from IR register, in this particular case the CPU is known as it is in the decoding phase of the instruction. The fourth case is to take the hardwired address (0x01) representing the beginning of the fetch microinstructions sequence, in this case CPU is known as it is in the fetch phase.

The table VI shows the binary encoding of the sequencer control bits field in CSR. Practically, the Sequencer Control circuit is implemented using a simple combinational logic, receiving two kinds of entrances, one for the 3 bits sequencer commands from the SCR, and the two flags ZF and NF.

TABLE V
CU COMMAND SIGNALS DESCRIPTION

Command signal type	Command signal name	Description
Registers Storing/Writing commands	Accumulator write (Accw)	Command signal to the register ordering the store/write of the incoming value.
	Stack Pointer write (SPw)	Command signal to the register ordering the store/write of the incoming value.
	Program Counter write (PCw)	Command signal to the register ordering the store/write of the incoming value.
	Instruction Register write (IRw)	Command signal to the register ordering the store/write of the incoming value.
	Address Bus write (ABw)	Command signal to the register ordering the store/write of the incoming value.
ALU operation/function bits	Function bit 0 (F0)	First bit of the selection/function of the ALU.
	Function bit 1 (F1)	Second bit of the selection/function of the ALU.
	Function bit 2 (F2)	Third bit of the function/function of the ALU.
Multiplexer selection bits	Multiplexer bit 0 (M0)	First bit of the selection command for the DP multiplexer.
	Multiplexer bit 1 (M1)	Second bit of the selection command for the DP multiplexer.
Memory and input/output devices commands (sent outside the CPU)	Memory Read Write bar (MRW _~)	Indicating a read or a write operation for the RAM, 1 to read from, and 0 to write-to.
	Output Write (OW)	Indicating the store/write to the output register inside the display device.
	Input Request (IR)	Sending a request to the binary keyboard device to send back the value entered by the user.

TABLE VI
BINARY ENCODING OF SC COMMANDS

The sequencer command code (binary format)	Functionality
0 (000)	Increment MPC address, and execute the next microinstruction inside MP-ROM.
1 (001)	Jump to the address stored in jump address field inside CSR.
2 (010)	Jump to opcode, and the beginning of the instruction execution sequence (decode phase).
3 (011)	Jump to fetch address (0x01) and beginning the fetch sequence for the new instruction.
4 (100)	Jump to the address stored in jump address field inside CSR if ZF flag is active (1).
5 (101)	Jump to the address stored in jump address field inside CSR if ZF flag is not active (0).
6 (110)	Jump to the address stored in jump address field inside CSR if NF flag is active (1).
7 (111)	Jump to the address stored in jump address field inside CSR if NF flag is not active (0).

And it outputs 2 control bits to the multiplexer in charge of choosing the next microinstruction address (Fig. 6). On Table VI, we can easily distinguish the last four sequencer commands, 4, 5, 6, and 7, they are different from the others in the way they are conditional, the SC has to check the flags to know if a jump to the address on CSR should be taken

TABLE VII
SNAP OF A MICROCODE CONTAINING THREE INSTRUCTIONS

Instruction	Microinstruction datapath	Microinstruction address	Jump address	Sequencer command	Command signals (in binary)	Microinstruction (in hex)
Fetch	$AB \leftarrow PC$	start=01	-	0	0001000110000	0x000230
	$IR \leftarrow AB, PC++$	02	-	0	0001010001100	0x00028C
	$MPC \leftarrow IR$	03	-	2	0000000000000	0x004000
LD (Load)	$AB \leftarrow PC$	OPcode=04	-	0	0001000110000	0x000230
	$PC++$	05	-	0	0001010000100	0x000284
	$AB \leftarrow Mem[AB]$	06	-	0	0000100110000	0x000130
	$Acc \leftarrow Mem[AB]$	07	-	3	0000100100001	0x006121
LDI (Load Immediate)	$AB \leftarrow PC$	OPcode=08	-	0	0001000110000	0x000230
	$PC++$	09	-	0	0001010000100	0x000284
	$Acc \leftarrow Mem[AB]$	0A	-	3	0000100100001	0x006121
STR (Store)	$AB \leftarrow PC$	OPcode=0B	-	0	0001000110000	0x000230
	$PC++$	0C	-	0	0001010000100	0x000284
	$AB \leftarrow Mem[AB]$	0D	-	0	0000100110000	0x000130
	$Mem[AB] \leftarrow Acc$	0E	-	3	0010000000000	0x006400

or not, this kind of commands is particularly useful for the branch instructions, where depending on the flags a jump to a nonsequential instruction should be done or not.

D. The Microprogramming

The microprogramming of the CU is about filling the MP-ROM with the list of the microinstructions in binary code of all the instructions supported by the CPU, each instruction is executed by executing its microcode formed by a limited number of microinstructions. The Table VII shows a snap of an example of three instructions, LD, LDI, CMP plus the fetch microcode.

On the Table VII, the first column concern the instruction name, although the fetch is not a proper instruction, but included in the beginning of every instruction. The second column is a representative data flow or the hop through the datapath occasioned by the actual microinstruction. The third is the address of the microinstruction inside the MP-ROM, actually the microinstructions are sequentially deployed inside the MP-ROM, the first one for each instruction represents its opcode. The next columns, Jmp address, Seq cmd (for Sequencer Commands), and Command signals represent the different encoding fields within 24 bits of a microinstruction (Fig. 7). The last column is about the hexadecimal representation of the overall 24 bits of the microinstruction. An example of the micro execution of the instruction LD is presented in the appendix.

1) *The fetch microcode:* This part of microcode shown on the first row of the VII VII is very important, it is unique in the way that it is not a proper instruction, but instead represent the fetch and decode phases of every instruction, and all SEP instructions should begin with this part of microcode and all of them should at the final microinstruction jump to this part to fetch and decode the next instruction. The fetch microcode contains three microinstructions, the first $AB \leftarrow PC$ like previously explained take the value of PC, which is the opcode of the instruction this time, to the AB register, now AB is pointing to the cell in memory containing the opcode, next the microinstruction $[IR \leftarrow AB, PC++]$ will at the same time increment PC and transfer the opcode to IR register, the last one $MPC \leftarrow IR$ is what we call the decoding phase, it is a jump

TABLE VIII
IMAGINED MAPPER MAKING ASSOCIATIONS

opcode	corresponding address inside the MP-ROM
0x25	0xF21
0xA1	0x23B
...	
...	

to the first microinstruction of the instruction decoded by CU. In fact the decoding mechanism in SEP CPU is rudimentary and straightforward and does not need any sort of mapping, this is done by jumping directly to the opcode, knowing that the opcode is actually the address of the first microinstruction for a given instruction inside MP-ROM.

2) *The mapper:* SEP processor does not use any sort of mapping, the opcodes of its instructions are the direct addresses of the beginning of the microcode for each instruction, and it is noticeable in Table VII (the entire table is accessible in [25]). Although many CPUs use the mapper inside the CU for the purpose of mapping their instructions, and it becomes the responsible actor in the decoding phase of an instruction. Generally the mapper is implemented using a simple combinational circuit reflecting the association table that associates each opcode with the first address of the related instruction microcode inside MP-ROM. A presumed example is shown in the Table VIII.

E. The Input/Output peripherals

The SEP architecture uses a simplistic input/output system, one device for the input represented by a binary keyboard, and one for the output in a form of a four seven segment displays.

1) *The binary keyboard device:* The binary keyboard is used by the user to write one byte, from the Fig. 1, eight switches are used for that purpose, in addition, another button called *Enter* button should also be used by the user to enter the value typed by the switches. An LED indicator shows if the keyboard buffer is full or empty, nevertheless even if the keyboard is full a new value could update the old one by modifying the switches and pushing *Enter* again. The keyboard on the other side, communicate with the processor through two lines, the *Input keyboard bus*, like the name suggest is used to

Fig. 8. Halt signal stopping mechanism circuit.

send the entered value to the CPU input port, and a signal command coming from the CU named IR (Input Request) seeking from the keyboard the permission to read the value from the *Input keyboard bus*, the keyboard could allow the processor to read if the user already filled the keyboard, if not the keyboard will through the *halt* signal to the clock circuit to stop the processor clock until the user enter his value, this pseudo-interrupt mechanism is solely used by the keyboard to stop the processor execution while reading, waiting for the user to enter his value, this halt mechanism was implemented by a simple circuit using only one AND gate, it is shown in Fig. 8.

For the Keyboard to achieve the behavior described above, it will need practically an 8 bits register like a buffer to store the user value, and a flip-flop cell to track the state of the buffer, if it's empty or full. The Fig. 9 depicts the internal circuit of the keyboard. When the user type a value using the 8 switches, he should push the enter button to validate, from the circuit the Enter button rises the clock signal of the register buffer, causing it to save the new value, at the same time, the Enter signal will set to 1 the cell indicator stating that the buffer is full and the LED is turned on. When the CPU needs to read from the keyboard, and programmatically it is only possible with the instruction IN, it will throw an IR (Input Request) signal, two possibilities are to be considered, the buffer is empty or it is full, for the first case, the Halt signal will be thrown by the AND gate between the IR signal and the inverse of the indicator cell (Fig. 9), causing the freeze of all the architecture until the user enter a value. In the second case, the buffer is full and no halt will be thrown and normally the CPU will read the value from the keyboard bus. When the IN instruction finishes the IR signal will fall to 0 causing a rising edge on the clock signal of the flip-flops indicator causing the saving of the constant 0 in its D entrance making the keyboard empty again.

2) *The display device*: The SEP uses 8 bits two's-complement integer representation to deal with numbers, that requires a display capability to show values in decimal format from -128 to +127, the display device uses four seven-segment components, one for the sign and the others for the three decimal digits. The display is shown in Fig. 10, it receives the value to be shown transmitted from the CPU using the output data bus, the CU also uses the OW (Output Write) command signal to tell the device the storing of the value to be shown from the output data bus in its buffer register.

Fig. 9. Keyboard internal circuit.

Fig. 10. Seven segment display device circuit.

The Fig. 10 shows how the buffer stores the value to be displayed by a write command to the register, the value stored is directly shown by the seven segment displays coming from the output register (or the buffer) passing through the three combinational circuits dedicated for each digit. To simplify the design, those combinational circuits are implemented using a 256 bytes look-up table ROMs. The sign display is easily implemented by connecting the middle segment LED of the even segment to the sign bit which is the last bit of the 8 bits two's-complement value, depicting doing so a minus typography.

IV. SEP ASSEMBLY

It is possible to program the SEP CPU using two different methods, writing instructions directly using the machine code format, conveniently with the hexadecimal format, the list of all the possible instructions is presented in the Table IX, it is called the ISA table. The second method is to use an assembly language which is more easy and less tedious to program than the machine language, it is then assembled to the machine language format. In any case the program should be submitted in its hexadecimal format to the RAM circuit before execution.

A. The ISA

Although the list of all the instructions was previously mentioned in section 2.3, the ISA in the Table IX lists the precise binary values and format of each instruction, each instruction is represented by its opcode on 8 bits, and potentially followed by one operand value on 8 bits too, the actual interpretation of the value depends on the addressing mode of that instruction.

TABLE IX
SEP ISA INSTRUCTIONS TABLE

Instructions	opcode	Addressing mode	Operand
TAS	0x55	Implicit	does not need operand.
TSA	0x56	Implicit	does not need operand.
LD	0x04	Direct	the address in RAM of the value to be loaded into ACC.
LDI	0x08	Immediate	The value to be loaded into ACC.
STR	0x0b	Direct	the address in RAM of the cell where the ACC would be stored.
ADD	0x0f	Direct	the address in RAM of the value to be added to ACC.
ADDI	0x13	Immediate	the value to be added to ACC.
SUB	0x16	Direct	the address in RAM of the value to be subtracted from ACC.
SUBI	0x1a	Immediate	the value to be subtracted from ACC.
LSR	0x24	Implicit	does not need operand.
NAND	0x21	Direct	the address in RAM of the value to be NAND bit-wised with ACC.
NANDI	0x1d	Immediate	the value to be NAND bit-wised with ACC.
CMP	0x3b	Direct	the address in RAM of the value to be subtracted from ACC.
CMPI	0x3f	Immediate	the value to be subtracted from ACC.
JMP	0x26	Direct	the address to jump-to.
JMPI ^a	0x28	Indirect	the address in RAM of the address to jump-to.
BLT	0x42	Direct	the address to jump-to if NF=1.
BGE	0x46	Direct	the address to jump-to if NF=0.
BEQ	0x4a	Direct	the address to jump-to if ZF=1.
BNE	0x4e	Direct	the address to jump-to if ZF=0.
PUSH	0x35	Implicit	does not need operand.
POP	0x38	Implicit	does not need operand.
JSR	0x2b	Direct	the address of the subroutine to jump-to.
RTS	0x32	Implicit	does not need operand, the PC is retrieved from the stack.
OUT	0x52	Implicit	does not need operand.
IN	0x53	Implicit	does not need operand.
NOP	0x25	Implicit	does not need operand.

^aThe I of JMPI mean indirect addressing mode, and it is different from other instructions where post-fixed by I means they use the Implicit addressing (like ADDI).

B. The assembly language

Like any assembler, the CustomAsm assembler [20] simplify significantly the programming process, it allows the student to easily create a custom assembler capable of generating machine code by defining formally each processor instruction. The assemble allows the use of 21 mnemonics for instructions instead of binary (or hexadecimal) opcodes representation, automatically converts negative numbers into two's-complement representation, or allowing programmers to use more complex arithmetic expressions in their code, like $((1+2)/3)*4$, and using labels and constants. The assembler also could decide by itself the appropriate addressing mode of an instruction relating to its syntax, for instance the two LOAD following instructions `ld 1` and `ld (1)` have two different opcodes, the first is the instruction LDI and the second is LD (Table IX), the assembler could deduct the difference by the parenthesis. It has also the ability to perform data declaration/reservation and initialization, supporting ASCII/UTF-8 characters and strings

encoding, reformatting the output executable file, and so on...

C. Coding examples

We will use a trivial code to demonstrate the use of both programming methods, assembly language and machine language, the listing 1 shows the assembly language code and the listing 2 shows its equivalent machine language program.

Listing 1. A trivial program using assembly language
loop :

```

out
add 1
add (1)
jmp loop
push

```

Listing 2. A trivial program using machine code language
addr | code

```

-----
10 |           ; loop address :
10 | 52       ; OUT
11 | 13 01    ; ADDI 1
13 | 0f 01    ; ADD 1
15 | 26 10    ; JMP loop
17 | 35       ; PUSH

```

Both codes are equivalent, nevertheless the assembly code is easier to program and understand due to the conveniences of the assembler, like the use of the label `loop` instead of the address and the use of the same `add` instruction for its two addressing modes instead of using two different instructions `ADD` and `ADDI` in machine code. Examples for more elaborate programs like Factorial and Fibonacci are available in GitHub [25].

V. COMPARISON WITH OTHER WORKS

We chose to compare our work with four other well known CPUs in academia, focusing essentially on the list of features and mechanisms provided by each CPU. Most of the CPUs used for comparison are hypothetical processor described in notorious COA and digital design books, this includes DLX described in [23] and [26], Mic-1 described in [27] and SAP described in [3], which led us to look for their corresponding implementations. DXL was partially described in [21], Mic-1 was implemented in this work [9] and SAP was implemented physically using TTL-74 logic chips [28]. The forth CPU called CHAMP was also designed using TTL-74 logic chips [19]. One well known teaching CPU known as MMIX [2] was not included for the comparison, due essentially to its architectural advancement, it was described by his author Donald Knuth as a CPU that can compete with actual modern CPUs [29]. Other CPUs were also discarded considered out of category since they were too advanced, like [30], or aiming to teach the CPU for several years, like [31] and [15]. The DXL was accepted for comparison yet it is considered advanced CPU, and practically full construction would be infeasible for a junior student in one academic semester or year, in practice, it is often partially implemented.

TABLE X
CPUS ABILITIES COMPARISON

Features	DLX	Mic-1	SAP	CHAMP	SEP
Architecture family (in bits)	32	8	8	4	8
Programmable registers	32	2	2	1	2
Instructions	64	9	11	14	21
Addressing modes	5	3	3	2	4
Arithmetic operations	4	2	2	2	2
Logic operations	3	2	0	0	1
Shift operation	3	0	0	0	1
Stack mechanism	yes	no	no	no	yes
Subroutines mechanism	yes	no	no	no	yes
flags bits	no	no	yes	no	yes
Branch instructions	2	0	2	1	4
Input/Output mechanism	both	both	output	no	both
Micro-programmability	-	yes	yes	no	yes
Assembler	yes	no	no	no	yes
Supportive architecture	no	no	no	no	yes
Interrupt mechanism	yes	no	no	no	pseudo

From analyzing the table X we can clearly denote the advancement in the DLX capabilities, actually, although it is an educational processor, it is considered equivalent to the well known commercial 32-bits processor MIPS, and it is even implemented through a 5-stage pipeline and ASIC proven [32], it would be unfair to compare it with SEP, nevertheless, it will demonstrate the actual place of the SEP CPU between high-end processor and the other oversimplified processors. This middle range position is well highlighted by the instruction numbers, the RISC real-world CPU DLX uses 64 instructions, SEP 20 instructions, and the others around ten instructions. The addressing modes are also of real interest, the simplistic processors implement often more simplistic modes, like implicit, immediate, and direct mode, SEP adds the indirect mode and DLX the more complex indexed mode (or based mode). Concerning the arithmetic operations, all the processors are quite equivalent by implementing the addition and subtraction, knowing that it is fairly easy to get subtraction from and adder utilizing two's-complement encoding, the DLX adds multiplication and division, a two heavy operations to integrate in a simplistic academic CPU. For the logic operations, almost all the simplistic CPUs do not own one, except the Mic-1, DLX implement tree operations and the SEP implements one, the reason behind implementing only one is that all logic operations are executed the same way and no teaching values are gained by adding more than one, and it's quite the same for the real-world processors, a small set of logical operations is sufficient to mimic the others not implemented, SEP implemented the universal NAND operation. Also, none of the simplistic CPUs implements the shift operation, making

it impossible to perform on those CPUs, SEP implements the logical right shift and the DLX adds two more flavors.

The stack is fully implemented in SEP and DLX, the other CPUs do not use the stack, and it is the same for the subroutines mechanisms, fully implemented by both SEP and DLX and not by the others, knowing that those two mechanisms are present in practically any real-world CPU. The flags put together in conjunction to form what sometimes called a status register, are very handy in certain situations, like been used for conditional instructions, extending the operations capabilities or tracking the execution correctness of a CPU, some real-world CPUs use them and some not, for example the DLX do not use the flags and uses others technics to get the same behavior. For the branch instructions, it is clear that SEP implement much more than the others, but in reality DLX is more rich in branch instructions than any CPU by using indirect combinations with other instructions used for that purpose.

A complete functional CPU needs to have an input/output mechanism, and from Table X, we note that only SAP and CHAMP lack a complete one. The microprogrammability is also an important feature that allows the CPU to be more flexible in adding/removing and modifying the instructions of a processor, except for CHAMP, which is a single cycle processor and uses a lookup ROM that procure him a certain degree of flexibility, each other CPU uses a microprogrammable ROM in some level of advancement. The DLX cannot be compared with, in this situation, because it is implemented in [21] by three different CUs and one of them is destined to a pipelined version, when Mic-2 and SAP hold a much primitive micro-programming mechanism compared to SEP. One important aspect of teaching COA is the use of assemblers, all the CPUs do not use assembler and are all programmed using machine languages, except the DLX, that can be considered a real CPU with a real full-stack development tools, whereas SEP is beneficiary by using a third-party easy to use customizable generic assembler called CustomAsm [20].

For the student to get the whole picture of a processor executing inside an entire architecture, a simplistic one was made for the processor SEP, and in this regard, it remains the only one that implement a complete architecture among the others, despite the fact that each one adds more or less some parts of an architecture to illustrate its execution. Regardless of the importance of an interrupt mechanism it was not integrated to SEP, only DLX includes one, adding a such mechanism would not be impossible, but it would require a significant amount of energy and knowledge from the student to assimilate this mechanism.

VI. CONCLUSION AND FUTURE WORKS

The elaboration of simplistic fictive processor for teaching COA is not new, one of the first was the Little Man Computer (LMC) in the 60s [33], they are supposed to teach junior students side-by-side computer organization and computer architecture. A common flaw arises from those kind of processors due to their inherent simplistic nature, is their disparity with the real full fledged processors, leading

to a steep learning curve for the real processors. In this perspective, we have tried to create a CPU that optimizes the equilibrium point for the two opposite constraints, seeking for the more simplistic processor possible and at the same time, the fullest real-world features one. The SEP processor contains most of the features found in real processors and not present in academic processors, like multi-addressing modes, stack, subroutine calls, status flags, input/output, microprogramming, proper assembler...etc. The design of the CPU was also supplemented by an integration within a complete Von Neumann architecture, using a small RAM and simplistic input/output devices.

The design of the CPU goes along with two complementary aspects, the architectural aspect which defines the hardware/software interface and its programmability, and the microarchitectural aspect which emphasis on the construction and the internal hardware organization inside the CPU. All along the paper, different aspects on how to design a CPU are highlighted, and many alternative design choices are mentioned even though they are not used for SEP, giving our work a learning foundation on how to design 8-bits like processors, and so this work is intended to be used for COA courses, books, labs, tutorials, projects...etc. Even though we tried to integrate the maximum number of real-world CPUs mechanisms, an important number of them could not be added due to exponential growth in the CPU design complexity, among the most important, we can list, the interrupt mechanism, still we used a pseudo-interrupt mechanism during the input operation, flags setter and getter instructions, indexed/based addressing mode, almost every current processor own a version similar to this mode, the carry flag, which was mandatory for real-world 8-bits CPUs allowing them to extend their computation to values greater than 8-bits, including 16, 24, 32-bits...etc, or more advanced architectural mechanism like pipelining, out-of-order, caching...etc. And finally it worth mentioning that an implementation of the CPU or the architecture on FPGA or using logic ICs could be really supportive for an academic point of view.

APPENDIX

EXECUTION EXAMPLE OF THE LD INSTRUCTION

We can see for instance that the LD (Load) instruction in Table VII is made-up with four consecutive microinstructions, the LD instruction use a direct addressing mode, in which involve two bytes of memory, one for the opcode followed by the address of the operand in memory. LD instruction works like any direct addressing mode instruction, it loads the value addressed by the second byte to Acc. Assuming the fetch and the decode phases of that instruction were done previously and PC now points to the second byte, the execution of LD begins with the first microinstruction $AB \leftarrow PC$ which will cause the CU to command the DP nodes to transfer the address saved in PC to AB (Address BUS) following a path crossing the multiplexer and the ALU, AB is now throwing out the CPU the address of the second byte directly to the RAM address port, which in return send back the actual address of the value to the CPU memory data input port. The

second microinstruction $PC++$ increment the PC using also a path crossing through the multiplexer and the ALU, which makes the PC pointing now to the next instruction opcode. The third microinstruction $AB \leftarrow Mem[AB]$ is responsible for making the CPU pointing to the actual value to load, by getting this address from memory and forward it to AB, going also through the multiplexer and the ALU. And the last one $Acc \leftarrow Mem[AB]$ which will do the same traversal as previously and getting the value from RAM and load it into ACC, and at the same time calls for a jump to the fetch microcode for the next instruction.

REFERENCES

- [1] D. E. Knuth, *The art of computer programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1968.
- [2] —, *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium (Art of Computer Programming)*. Addison-Wesley Professional, 2005.
- [3] A. P. Malvino and J. A. Brown, *Digital Computer Electronics*, 3rd ed. Mc Graw Hill publisher, 1993.
- [4] L. Null and J. Lobur, *Essentials of Computer Organization and Architecture*, 1st ed. USA: Jones and Bartlett Publishers, Inc., 2003.
- [5] H.-P. Gumm and M. Sommer, *Einführung in die Informatik (Introduction into computer science)*, 8th ed. Addison-Wesley, 2008.
- [6] J. Carpinelli, *Computer Systems Organization and Architecture*. Pearson Education, 2000.
- [7] F. García-Carballeira, A. Calderón-Mateos, S. Alonso-Monsalve, and J. Prieto-Cepeda, “Wepsim: An online interactive educational simulator integrating microdesign, microprogramming, and assembly language programming,” *IEEE Transactions on Learning Technologies*, vol. 13, no. 1, pp. 211–218, 2020.
- [8] C. Burch, “Logisim: A graphical system for logic circuit design and simulation,” *J. Educ. Resour. Comput.*, vol. 2, no. 1, p. 5–16, Mar. 2002.
- [9] D. C. Schuurman, “Step-by-step design and simulation of a simple cpu architecture,” in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 335–340.
- [10] M. H. H. Ichsan and W. Kurniawan, “Cpu implementation using only logisim simulator to achieve computer architecture learning outcome,” *Bulletin of Electrical Engineering and Informatics*, vol. 9, 04 2020.
- [11] —, “Design and implementation 8 bit cpu architecture on logisim for undergraduate learning support,” in *2017 International Conference on Sustainable Information Engineering and Technology (SIET)*, 2017, pp. 132–137.
- [12] C. Salazar and M. Bobby Birrer, “Instrumentation and extension of reduced, simulated single cycle mips architecture to improve student comprehension,” in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–5.
- [13] T. D. Stanley, G. Embrey, D. Prigmore, L. Fife, S. Mikolyski, and D. Colton, “Pedagogic value in understanding computer architecture of implementing the marie computer from null and lobur in the logic emulation software, multimedia logic,” in *Proceedings of the 2007 Workshop on Computer Architecture Education*, ser. WCAE ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 66–71.
- [14] C. W. Kann, “Implementing a one address cpu in logisim,” Open Textbooks, 07 2016. [Online]. Available: <https://cupola.gettysburg.edu/oer/3>
- [15] A. Yıldız, H. F. Ugurdag, B. Aktemur, D. İskender, and S. Gören, “Cpu design simplified,” in *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, 2018, pp. 630–632.
- [16] K. Nakano and Y. Ito, “Processor, assembler, and compiler design education using an fpga,” in *2008 14th IEEE International Conference on Parallel and Distributed Systems*, 2008, pp. 723–728.
- [17] X. Li and T. Li, “Ecomips: an economic mips cpu design on fpga,” in *4th IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2004, pp. 291–294.
- [18] M. V. Kervinck and W. Belgers. (2018) Gigatron, ttl microcomputer. [Online]. Available: <https://gigatron.io/>
- [19] D. Feinberg, “A simple and affordable ttl processor for the classroom,” *Computer Science Education*, vol. 17, pp. 107–116, 06 2007.
- [20] H. Lorenzi. (2016) Customasm. GitHub repository. [Online]. Available: <https://github.com/hlorenzi/customasm>

- [21] D. Harris and S. Harris, *Digital Design and Computer Architecture, Second Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.
- [22] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. Technical Report No. UCB/EECS-2014-146, 08 2014. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>
- [23] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [24] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *USENIX Security Symposium*, 2017, pp. 1163–1180.
- [25] A. Kara. (2021, 08) Sep cpu. GitHub repository. [Online]. Available: <https://github.com/kara-abdelaziz/SEP-CPU>
- [26] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- [27] A. S. Tanenbaum and J. R. Goodman, *Structured Computer Organization*, 4th ed. USA: Prentice Hall PTR, 1998.
- [28] B. Eater. (2016) Build an 8-bit computer from scratch. breadboard computer kit. [Online]. Available: <https://eater.net/8bit/>
- [29] D. Knuth, "Mmixware, a risc computer for the third millennium," in *Lecture Notes in Computer Science*, 1999.
- [30] A. A. S. Abdo and L. Yijun, "Design of a five stage pipeline cpu with interruption system," *Global Journal of Computer Science and Technology*, vol. 15, no. 2, 2015.
- [31] N. Bhardwaj, M. Senftleben, and K. Schneider, "Abacus: A processor family for education," in *Proceedings of the WESE'14: Workshop on Embedded and Cyber-Physical Systems Education*, ser. WESE'14. New York, NY, USA: Association for Computing Machinery, 2014.
- [32] M. Amde, I. Blunno, and C. P. Sotiriou, "Automating the design of an asynchronous dlx microprocessor," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 502–507.
- [33] W. Yurcik and H. Osborne, "A crowd of little man computers: Visual computer simulator teaching tools," in *Proceedings of the 33rd Conference on Winter Simulation*, ser. WSC '01. USA: IEEE Computer Society, 2001, p. 1632–1639.

and Computer Organization and Architecture.

Abdelaziz Kara was born in 1981 in Bordj Bou Arreridj, Algeria. He received a computer science engineering degree from the University of Sétif (Algeria) in 2006, he then got to study embedded systems and their use on p2p-TV and received magister degree from the University of Sétif in 2010. Now he is pursuing his Ph.D. in Hardware Architectures and CPUs design.

Since 2010, he has been an Assistant Professor with the department of computer science in the University of Sétif, teaching mostly Hardware Design

Messaoud Mostefai is a Professor at the computer science department of Bordj Bou Arreridj University. He is the head of the MSE Laboratory. His main research interests include reconfigurable architectures, pattern recognition, Real-time signal, image and video processing.